

# Руководство по реализации криптографии на эллиптических кривых

Санан Корняков

2024

# Содержание

<b>1</b>	<b>Введение</b>	<b>3</b>
1.1	Условия игры	3
1.2	База	3
1.3	Постановка задачи	4
<b>2</b>	<b>Длинная арифметика</b>	<b>4</b>
2.1	Каркас	4
2.2	Методы	5
2.3	Главный тип	17
<b>3</b>	<b>Поле</b>	<b>18</b>
3.1	Элемент поля	18
3.1.1	Каркас	18
3.1.2	Методы	18
3.2	Поле	20
<b>4</b>	<b>Эллиптические кривые</b>	<b>21</b>
4.1	Точка эллиптической кривой	21
4.1.1	Каркас	22
4.1.2	Методы	22
4.2	Эллиптическая кривая	27
<b>5</b>	<b>Эллиптический Эль-Гамаль</b>	<b>30</b>
5.1	Каркас	30
5.2	Методы	31
<b>6</b>	<b>ECDSA</b>	<b>33</b>
6.1	Каркас	35
6.2	Методы	35
<b>7</b>	<b>Тестирование</b>	<b>36</b>
7.1	Длинка	37
7.2	Эль-Гамаль	38
7.3	ECDSA	38
<b>8</b>	<b>Специализация</b>	<b>38</b>
8.1	Поле	39
8.1.1	Арифметика	39
8.1.2	Умножение	41
8.1.3	Инверсия	42
8.1.4	Пролом 4 стены	42
8.1.5	Конверсия	43
8.2	Точка Эллиптической кривой	44
8.3	Эль-Гамаль	44
8.4	Тестирование	45
<b>9</b>	<b>Заключение</b>	<b>46</b>
	<b>Список литературы</b>	<b>47</b>

## Аннотация

Работа является пошаговым руководством по реализации криптографии на эллиптических кривых. Реализованы объекты длинной арифметики, полей и эллиптических кривых. Изучены и имплементированы алгоритмы шифрования и дешифрования, электронной цифровой подписи, подсчёт количества точек на эллиптической кривой, быстрого умножения и деления длинных чисел. Протестированы объекты и алгоритмы по скорости, сравнивая с готовыми решениями. Руководство параллельно с имплементацией объясняет и рассказывает, что и зачем было реализовано.

*Ключевые слова:* эллиптические кривые, шифрование и дешифрование, криптография, ECDSA, ECC, длинная арифметика, C++, конечные поля, оптимизация

## 1 Введение

Современная криптография с нынешними вычислительными мощностями требует значительных усилий в шифровке сообщений, и шифрование с помощью эллиптических кривых - один из мощнейших инструментов. Но доступных и полных объяснений от начала до конца по шифрованию на них ничтожно мало, поэтому я решил сделать руководство для людей, которые хотят ознакомиться с данным видом криптографии.

### 1.1 Условия игры

Если вы искали данное руководство, то скорее всего где-то слышали/читали об эллиптических кривых и о возможности криптографии на них, поэтому я рассчитываю на базовое понимание математики и алгоритмов.

Здесь не будет дотошного доказательства теорем или строгости в описании математических объектов — в первую очередь акцент делается именно на имплементации (на языке C++). Данный язык был выбран в качестве общеизвестного языка среди программистов. Выберем C++20 для удобного использования шаблонов.

В данном руководстве мы будем стараться использовать как можно меньше готовых библиотек, чтобы не было огромных black box-ов в нашем коде. Это улучшит понимание и возможности алгоритмов.

### 1.2 База

Знаменитая формула

$$y^2 = x^3 + ax + b$$

обычно является самой первой, которую вы увидите при описании криптографии эллиптических кривых. Появляется несколько вопросов:

- Что такое  $x, y, a, b$ ? Где лежат данные числа?

Данные числа являются элементами некоего поля  $\mathbb{F}$ , над которым построена эллиптическая кривая, характеристики больше 3 (забьём на последние слова, так как мы будем работать с полями достаточно больших характеристик). Поле поддерживает все стандартные математические операции: сложение, вычитание, деление на ненулевой элемент, умножение, поэтому можно пока считать его  $\mathbb{R}$ .

- Что такое эллиптическая кривая?

Это группа точек в  $\mathbb{F}^2$ , координаты которых удовлетворяют данному уравнению, и ещё точка бесконечности  $\mathcal{O}$ , которая является своеобразным нулём группы. Сложение в группе происходит по специальным формулам на координаты, которые будут рассмотрены позже. Умножение точки на натуральное число приравнивается к сложению точки с собой это число раз.

- Как это используют для шифрования?

Обычно выбирается эллиптическая кривая  $\mathbb{E}$  над неким полем  $\mathbb{F}$ , точка  $P$  на ней и производится умножение точки на натуральное число  $k$ . Криптографическая стойкость достигается сложностью нахождения числа  $k$  по точкам  $P$  и  $kP$ .

- Чем это лучше других методов шифрования?

Тем, что данный способ шифрования можно реализовать так, что он будет выполняться быстрее других алгоритмов при аналогичной задаче и данных. Также, для одинаковых показателей криптографической стойкости, криптография на эллиптических кривых требует ключей (чисел для шифрования) меньшей длины, чем другие алгоритмы.

### 1.3 Постановка задачи

Начитавшись статей на хабре, мы воодушевились и решили написать свою криптографию на эллиптических кривых. Сначала надо определить, какие объекты нам надо реализовать:

- Нам надо реализовать эллиптическую кривую. Но эллиптическая кривая никто без поля, значит нам надо реализовать поле.
- Так как поля бывают бесконечными, а мы работаем на компьютере с числами, то ограничимся на простые поля  $\mathbb{F}_p$ , которые представим в виде вычетов по простому модулю  $p$ . Но этот простой модуль и числа в поле надо представить в виде целых чисел, а в криптографии обычно используются числа из более чем 200 битов. Целые числа такого размера не поддерживаются языком C++, поэтому нам надо реализовать класс целых чисел и длинную арифметику на них.

Итого 3 объекта: целые числа, поле, эллиптическая кривая. Приступим наконец к реализации!

## 2 Длинная арифметика

**Если не хотите запариваться и сделать основную рабочую лошадку блэббоксом, то можно просто установить длинку `boost::multiprecision` и скипнуть данную часть.**

Так как мы хотим реализовать вычеты по большому модулю, то достаточно реализовать беззнаковые длинные целые числа.

Основной приём для имплементации длинной арифметики - хранение чисел в основании  $2^{32}$  или  $2^{64}$ . То есть просто массив из целых чисел, которые представляют цифры данного числа в соответствующих основаниях. Есть несколько видов данного представления:

1. Количество цифр в числе меняется в зависимости от размера числа. Нет математического ограничения длины числа, только аппаратное.
2. Количество цифр в числе фиксировано и не меняется от числа.

Последний вариант можно видеть, например, в типе `uint64_t` — присутствуют сразу все 64 бита не зависимо от содержащихся данных. Данный вид целых чисел является наиболее удобным в реализации и использовании по назначению, поэтому будем имплементировать его.

### 2.1 Каркас

Так как количество бит в числе может разительно отличаться от задачи к задаче, то общим решением будет создать шаблонный класс по количеству содержащихся в нём бит:

```
1 template<size_t c_bits>
2 class uint_t {
3 };
```

Теперь надо определиться с представлением цифр в нашем классе. Из-за того, что для алгоритма деления, который будет позже, потребуется деление по две цифры, то возьмём за цифру `uint32_t`, чтобы можно было спокойно делить в `uint64_t`.

С помощью `constexpr` определим размеры и длину необходимого массива. Так как длина не изменяется во время жизни объекта, то возьмём `std::array` за контейнер. Он будет гарантировать, что длина массива сохраняет свой инвариант. Итого получилось:

```
1 template<size_t c_bits>
2 class uint_t {
3     using digit_t = uint32_t;
4     using double_digit_t = uint64_t;
5
6     static constexpr size_t c_bits_in_byte = 8;
7     static constexpr size_t c_digit_size = sizeof(digit_t) * c_bits_in_byte;
8     static constexpr size_t c_digit_number = c_bits / c_digit_size;
9     static constexpr size_t c_double_digit_size = sizeof(double_digit_t) * c_bits_in_byte;
10    static constexpr size_t c_double_digit_number = c_bits / c_double_digit_size;
11
12    template<size_t V>
```

```

13     friend class uint_t;
14
15     using digits = std::array<digit_t, c_digit_number>;
16     digits m_digits = {};
17 };

```

Дали псевдонимы используемым типам, чтобы улучшить читаемость и не менять все типы, если вдруг захотим использовать за цифру `uint16_t` и какой-то другой тип. В `m_digits` храним число в основании  $2^{32}$  в little-endian. 12-13 строчкой мы подружили все шаблоны друг с другом для общего взаимодействия.

## 2.2 Методы

Теперь мы хотим как-то общаться с нашими данными. Желательно имплементировать все операции, которые можно применять к обычным типам, таким как `uint32_t`, `uint64_t`.

- Конструирование: Есть несколько сценариев:

1. Хотим сконструироваться от целочисленного типа. Заметим, что количество бит в нём может быть как больше 32, так и меньше, поэтому ограничиться назначением `m_blocks[0]` нельзя. Тогда определим вспомогательный приватный метод, который будет определять через концепты отношения количества бит:

```

1  template<typename From, typename To>
2  concept is_upcastable_to = sizeof(From) <= sizeof(To) && is_convertible_to<From, To>;
3
4  template<typename From, typename To>
5  concept is_downcastable_to = sizeof(From) > sizeof(To) && is_convertible_to<From, To>;
6
7  template<typename T>
8  requires std::numeric_limits<T>::is_integer && is_upcastable_to<T, digit_t>
9  static constexpr digits split_into_digits(T value) {
10     return {static_cast<digit_t>(value)};
11 }
12
13 template<typename T>
14 requires std::numeric_limits<T>::is_integer && is_downcastable_to<T, digit_t>
15 static constexpr digits split_into_digits(T value) {
16     digits result = {};
17
18     for (size_t i = 0; i < c_digit_number; ++i) {
19         result[i] = static_cast<digit_t>(value);
20         value >>= c_digit_size;
21
22         if (value == 0) {
23             break;
24         }
25     }
26
27     return result;
28 }

```

2. Хотим сконструироваться от контейнера целых чисел, например как наш `uint_t`. Для этого пишем концепт, который определяет, что данный тип действительно является контейнером целых чисел, и копируем его данные:

```

1  template<typename Container, typename T>
2  concept is_convertible_container = requires(Container t, size_t i) {
3      { t[i] } -> is_convertible_to<T>;
4      { t.size() } -> std::same_as<size_t>;
5  };
6
7  template<typename T>
8  requires is_convertible_container<T, digit_t> || requires(T x) {
9      { uint_t {x} } -> std::same_as<T>;
10 }
11
12 static constexpr digits split_into_digits(const T& other) {
13     const size_t min_size = std::min(size(), other.size());
14     digits result = {};

```

```

14     for (size_t i = 0; i < min_size; i++) {
15         result[i] = static_cast<digit_t>(other[i]);
16     }
17
18     return result;
19 }

```

3. Хотим сконструироваться от строк (С-строк). Действительно, это единственный удобный способ задать необходимое нам число с количеством бит больше чем у `size_t`.

Для этого нам нужно сначала написать парсер строки, которая представляет число в двоичном, шестнадцатеричном и десятичном форматах. Так как мы хотим использовать методы `uint_t`, то нам нужно парсить в любой класс, который удовлетворяет критериям целочисленных классов:

```

1  template<typename T>
2  concept is_integral = std::is_integral_v<T> || requires(T t, T* p, void (*f)(T)) {
3      f(0);
4      p + t;
5  };
6  template<typename T>
7  requires is_integral<T>
8  constexpr T parse_into_uint(const char* str) {
9      assert(str != nullptr && "parse_into got nullptr");
10
11     T value = 0;
12     uint16_t radix = 10;
13
14     if (str[0] == '0' && str[1] == 'x') {
15         radix = 16;
16         str += 2;
17     } else if (str[0] == '0' && str[1] == 'b') {
18         radix = 2;
19         str += 2;
20     } else if (str[0] == '0') {
21         radix = 8;
22         ++str;
23     }
24
25     while (*str != '\0') {
26         value *= static_cast<T>(radix);
27         uint16_t symbol_value = radix + 1;
28
29         if (*str >= '0' && *str <= '9') {
30             symbol_value = static_cast<uint16_t>(*str - '0');
31         } else if (*str >= 'a' && *str <= 'f') {
32             symbol_value = static_cast<uint16_t>(*str - 'a') + 10;
33         } else if (*str >= 'A' && *str <= 'F') {
34             symbol_value = static_cast<uint16_t>(*str - 'A') + 10;
35         }
36
37         if (symbol_value >= radix) {
38             assert(false && "parse_into got incorrect string");
39         }
40
41         value += static_cast<T>(symbol_value);
42         ++str;
43     }
44
45     return value;
46 }

```

Теперь мы готовы определить конструкторы класса:

```

1  constexpr uint_t() = default;
2
3  template<typename T>
4  constexpr uint_t(const T& value) : m_digits(split_into_digits<T>(value)) {}
5
6  constexpr uint_t(const char* str) : m_digits(parse_into_uint<uint_t>(str).m_digits) {};
7

```

```
8 constexpr uint_t& operator=(const uint_t& value) = default;
```

Специально не делаем их explicit для неявных конвертаций. С помощью шаблонного конструктора мы можем конструироваться от других экземпляров нашего класса, например:

```
1 uint_t<128> a = ...;
2 uint_t<160> b(a);
```

- Сложение: Самое простое, но тем не менее лучшее решение - это сложение в столбик. Используем стандартизированное переполнение беззнаковых типов в C++ для определения, есть ли остаток от сложения наших 32-битных чисел:

```
1 constexpr uint_t& operator+=(const uint_t& other) {
2     digit_t carry = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         digit_t sum = carry + other[i];
6         m_digits[i] += sum;
7         carry = (m_digits[i] < sum) || (sum < carry);
8     }
9
10    return *this;
11 }
```

Используем ключевое слово constexpr для вычисления значения некоторых констант во время компиляции.

Теперь мы хотим определить простое сложение, т.е. operator+. Его можно было бы сделать через

```
1 constexpr uint_t operator+(const uint_t& other) const {
2     uint_t result = *this;
3     result += other;
4     return result;
5 }
```

Обратите внимание, что мы не пишем:

```
1 constexpr uint_t operator+(const uint_t& other) const {
2     uint_t result = *this;
3     return result += other;
4 }
```

так как тогда мы будем возвращать ссылку uint\_t&, что не затриггерит NRVO.

Вместо определения метода класса, мы напишем 4 дружественных функции:

```
1 friend constexpr uint_t operator+(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = lhs;
3     result += rhs;
4     return result;
5 }
6
7 friend constexpr uint_t operator+(uint_t&& lhs, const uint_t& rhs) {
8     lhs += rhs;
9     return lhs;
10 }
11
12 friend constexpr uint_t operator+(const uint_t& lhs, uint_t&& rhs) {
13     rhs += lhs;
14     return rhs;
15 }
16
17 friend constexpr uint_t operator+(uint_t&& lhs, uint_t&& rhs) {
18     lhs += rhs;
19     return lhs;
20 }
```

Тут есть два оптимизационных момента:

1. Эффективно используется то, что нам передали `r` — value, и не копируем данные. Обычно это возникает при многократном сложении или в других сложных формулах:

```
1 uint_t a,b,c = ...;
2 uint_t result = a + b + c;
```

2. Теперь можно неявно заапкастить другие типы к `uint_t`, чтобы применить данное сложение. Это позволяет писать:

```
1 uint_t a = ...;
2 uint_t b = 3 + a;
```

что было бы невозможно при внутреннем методе. Везде далее будем использовать по возможности внешние `friend` функции для возможности неявного апкаста других типов.

- Вычитание: Оно абсолютно аналогично делается через вычитание в столбик:

```
1 constexpr uint_t& operator--(const uint_t& other) {
2     digit_t remainder = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         digit_t prev = m_digits[i];
6         digit_t sum = other[i] + remainder;
7         m_digits[i] -= sum;
8         remainder = (m_digits[i] > prev) || (sum < remainder);
9     }
10
11     return *this;
12 }
```

Так как мы хотим писать такие конструкции как:

```
1 uint_t a = ...;
2 uint_t b = -a;
```

то нужно определить отрицание. В компьютерах отрицательные целые числа представляются как флиппнутые биты + 1. Рассмотрим на примере:

0000000000000000000101

+

111111111111111111011

=

00000000000000000000

Определим `inplace` отрицание как приватный метод:

```
1 constexpr void negative() {
2     for (size_t i = 0; i < c_digit_number; ++i) {
3         m_digits[i] = ~(m_digits[i]);
4     }
5
6     ++*this;
7 }
```

Значит отрицанием будет:

```
1 constexpr uint_t operator-() const {
2     uint_t result = *this;
3     result.negative();
4     return result;
5 }
```

Определяем внешние `friend` для вычитания:



```

1  friend constexpr uint_t operator-(const uint_t& lhs, const uint_t& rhs) {
2      uint_t result = lhs;
3      result -= rhs;
4      return result;
5  }
6
7  friend constexpr uint_t operator-(uint_t&& lhs, const uint_t& rhs) {
8      lhs -= rhs;
9      return lhs;
10 }
11
12 friend constexpr uint_t operator-(const uint_t& lhs, uint_t&& rhs) {
13     rhs -= lhs;
14     rhs.negative();
15     return rhs;
16 }
17
18 friend constexpr uint_t operator-(uint_t&& lhs, uint_t&& rhs) {
19     lhs -= rhs;
20     return lhs;
21 }

```

- Умножение: Самым лёгким и относительно быстрым способом будет метод сканирования операнда (или построчное умножение). Заключается в построчном обновлении цифр ответа и накоплении прежних результатов умножения для новой строки. Получается такое динамическое программирование:

```

1  friend constexpr uint_t operator*(const uint_t& lhs, const uint_t& rhs) {
2      uint_t result;
3
4      for (size_t i = 0; i < c_digit_number; ++i) {
5          uint64_t u = 0;
6
7          for (size_t j = 0; i + j < c_digit_number; ++j) {
8              u = static_cast<uint64_t>(result[i + j])
9                  + static_cast<uint64_t>(lhs[i]) * static_cast<uint64_t>(rhs[j])
10                 + (u >> c_digit_size);
11              result[i + j] = static_cast<uint32_t>(u);
12          }
13      }
14
15      return result;
16 }

```

Заметим, что мы отбрасываем цифры, чей номер не помещается в `c_digit_number`. Улучшения этого алгоритма можно найти в [1].

Также в реализации имплементировано [нерекурсивное FFT](#) умножение двух комплексных полиномов, но из-за большой ошибки при конвертации обратно в целочисленные цифры он выдаёт неверный ответ уже при длине числа в две цифры, что непозволительно для умножения. Поэтому он не используется и не рассматривается в этом гайде. Если вы смогли реализовать более точный FFT или NTT, то можете попробовать использовать их.

- Деление: Чтобы поделить два целочисленных длинных числа используем алгоритм-D Кнута [2]:

Задача - поделить два длинных числа, представленных цифрами с основанием  $b$ , где  $b$  в имплементации  $2^{32}$  или  $2^{64}$ .

Рассмотрим сначала  $u = (u_n u_{n-1} \dots, u_0)_b$  и  $v = (v_{n-1} \dots v_0)_b$ , где  $u/v < b$ . Найдём алгоритм для вычисления  $q := \lfloor u/v \rfloor$ :

Заметим, что  $u/v < b \Leftrightarrow u/b < v \Leftrightarrow \lfloor u/b \rfloor < v$ , а это условие того, что

$$(u_n u_{n-1} \dots, u_1)_b < (v_{n-1} \dots v_0)_b$$

Если обозначить  $r := u - qv$ , то  $q$  - это уникальное число, такое что  $0 \leq r < v$ . Пусть

$$\hat{q} := \min \left( \left\lfloor \frac{u_n b + u_{n-1}}{v_{n-1}} \right\rfloor, b - 1 \right)$$

Т.е. мы получаем гипотетическое значение  $q$ , поделив первые две цифры  $u$  на первую цифру  $v$ , а если результат деления больше или равен  $b$ , то берём  $b - 1$ . Для такого  $\hat{q}$  выполняются две теоремы:

**Теорема 1.**  $\hat{q} \geq q$

**Теорема 2.** Если  $v_{n-1} \geq \lfloor b/2 \rfloor$ , то  $\hat{q} - 2 \leq q \leq \hat{q}$ .

Существенно ограничили нашу гипотезу. Умножив  $u$  и  $v$  на  $\lfloor b/(v_{n-1} + 1) \rfloor$ , мы не изменим длину числа  $v$  и результат деления. После этого умножения станет выполняться вторая из данных теорем.

**Алгоритм D:** Дано неотрицательное целое число  $u = (u_{m+n-1}, \dots, u_1, u_0)_b$  и  $v = (v_{n-1}, \dots, v_1, v_0)_b$ , где  $v_{n-1} \neq 0$  и  $n > 1$ . Мы хотим посчитать  $\lfloor u/v \rfloor = (q_m, q_{m-1}, \dots, q_0)_b$  и остаток  $u \bmod v = (r_{n-1}, \dots, r_0)_b$ :

1.  $d := \lfloor b/(v_{n-1} + 1) \rfloor$ . Тогда пусть  $(u_{m+n}u_{m+n-1} \dots u_1u_0)_b := (u_{m+n-1} \dots u_1u_0)_b \cdot d$ , аналогично,  $(v_{n-1}, \dots, v_1, v_0)_b = (v_{n-1}, \dots, v_1, v_0)_b \cdot d$ . Заметим, что новая цифра могла появиться только у  $u$
2. Итерироваться будем по  $j$ , которая в начале равна  $m$  (Делить в следующих шагах будем  $(u_{j+n} \dots u_{j+1}u_j)_b$  на  $(v_{n-1} \dots v_1v_0)_b$  чтобы получить цифру  $q_j$ )
3.  $\hat{q} := \left\lfloor \frac{u_{j+n}b + u_{j+n-1}}{v_{n-1}} \right\rfloor$  и пусть  $\hat{r}$  будет остатком, т.е.  $\hat{r} := u_{j+n}b + u_{j+n-1} \pmod{v_{n-1}}$
4. Если  $\hat{q} \geq b$  или  $\hat{q}v_{n-2} > b\hat{r} + u_{j+n-2}$ , то уменьшаем  $\hat{q}$  на 1 и увеличиваем  $\hat{r}$  на  $v_{n-1}$ . Если  $\hat{r} < b$ , то повторяем данный шаг
5. Заменим  $(u_{j+n} \dots u_{j+1}u_j)_b$  на

$$(u_{j+n} \dots u_{j+1}u_j)_b - \hat{q}(0v_{n-1} \dots v_1v_0)_b$$

6. Назначаем  $q_j = \hat{q}$
7. Если число  $u$  на 5 шаге получилось отрицательным, то добавляем к нему  $b^{n+1}$  и переходим к шагу 8, иначе переходим к шагу 9.
8. (Вероятность данного шага крайне мала, за счёт чего достигается асимптотическая быстрота алгоритма) Уменьшаем  $q_j$  на 1 и добавляем  $(0v_{n-1} \dots v_1v_0)_b$  к  $(u_{j+n} \dots u_{j+1}u_j)_b$  (при сложении появится цифра  $u_{j+n+1}$ , её следует проигнорировать)
9. Уменьшаем  $j$  на 1. Если  $j \geq 0$ , то возвращаемся на шаг 3
10. Теперь  $q = (q_m \dots q_1q_0)$  - это искомое частное, а искомый остаток можно получить, поделив  $(u_{n-1} \dots u_1u_0)$  на  $d$  наивным способом
11. Возвращаем  $(q, r)$

Алгоритм сверху применяется только при размере делителя больше 1 цифры и не больше количества цифр в делимом, так как при меньших размерах есть более быстрые оптимизации, значит нам понадобится приватный метод определения количества цифр в числе:

```

1 constexpr size_t actual_size() const {
2     size_t result = c_digit_number;
3
4     while (result > 0 && m_digits[result - 1] == 0) {
5         --result;
6     }
7
8     return result;
9 }
```

Для удобной работы с `m_digits` определим приватные методы для `operator[]`, которые будут проталкивать его внутрь:

```

1 constexpr const digit_t& operator[](size_t pos) const {
2     return m_digits[pos];
3 }
4
5 constexpr digit_t& operator[](size_t pos) {
6     return m_digits[pos];
7 }
```

Наконец, определим приватный метод divide, который будет вычислять, в каком случае мы находимся.

```
1  static constexpr uint_t divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
2      size_t dividend_size = lhs.actual_size();
3      size_t divisor_size = rhs.actual_size();
4
5      // CASE 0:
6      if (dividend_size < divisor_size) {
7          if (remainder != nullptr) {
8              *remainder = lhs;
9          }
10
11         return uint_t(0);
12     }
13
14     // CASE 1:
15     if (divisor_size == 1) {
16         return divide(lhs, rhs[0], remainder);
17     }
18
19     // CASE 2:
20     return d_divide(lhs, rhs, remainder);
21 }
22
23 static constexpr uint_t divide(const uint_t& lhs, const digit_t& rhs, uint_t* remainder = nullptr) {
24     uint_t result;
25     double_digit_t part = 0;
26
27     for (size_t i = c_digit_number; i > 0; --i) {
28         part = (part << (c_digit_size)) + static_cast<double_digit_t>(lhs[i - 1]);
29
30         if (part < rhs) {
31             continue;
32         }
33
34         result[i - 1] = static_cast<digit_t>(part / rhs);
35         part %= rhs;
36     }
37
38     if (remainder != nullptr) {
39         *remainder = uint_t(static_cast<digit_t>(part));
40     }
41
42     return result;
43 }
44
45 static constexpr uint_t d_divide(const uint_t& lhs, const uint_t& rhs, uint_t* remainder = nullptr) {
46     size_t dividend_size = lhs.actual_size();
47     size_t divisor_size = rhs.actual_size();
48
49     uint_t<c_bits + c_digit_size> dividend(lhs);
50     uint_t divisor(rhs);
51     uint_t quotient;
52
53     size_t shift_size = 0;
54     digit_t divisor_head = divisor[divisor_size - 1];
55     static constexpr double_digit_t c_HalfBlock = static_cast<double_digit_t>(1)
56                                                 << (c_digit_size - 1);
57
58     while (divisor_head < c_HalfBlock) {
59         ++shift_size;
60         divisor_head <<= 1;
61     }
62
63     dividend <<= shift_size;
64     divisor <<= shift_size;
65
66     double_digit_t divisor_ = divisor[divisor_size - 1];
67     static constexpr double_digit_t c_Block = static_cast<double_digit_t>(1) << c_digit_size;
68
69     for (size_t i = dividend_size - divisor_size + 1; i > 0; --i) {
70         double_digit_t part =
```

```

71         (static_cast<double_digit_t>(dividend[i + divisor_size - 1]) << c_digit_size)
72         + static_cast<double_digit_t>(dividend[i + divisor_size - 2]));
73     double_digit_t quotient_temp = part / divisor_;
74     part %= divisor_;
75
76     if (quotient_temp == c_Block) {
77         --quotient_temp;
78         part += divisor_;
79     }
80
81     while (part < c_Block
82            && (quotient_temp * divisor[divisor_size - 2]
83               > (part << c_digit_size) + dividend[i + divisor_size - 3])) {
84         --quotient_temp;
85         part += divisor_;
86     }
87
88     int64_t carry = 0;
89     int64_t widedigit = 0;
90
91     for (size_t j = 0; j < divisor_size; ++j) {
92         double_digit_t product =
93             static_cast<digit_t>(quotient_temp) * static_cast<double_digit_t>(divisor[j]);
94         widedigit = (static_cast<int64_t>(dividend[i + j - 1]) + carry) - (product & UINT32_MAX);
95         dividend[i + j - 1] = static_cast<digit_t>(widedigit);
96         carry = (widedigit >> c_digit_size) - static_cast<double_digit_t>(product >> c_digit_size);
97     }
98
99     widedigit = static_cast<int64_t>(dividend[i + divisor_size - 1]) + carry;
100    dividend[i + divisor_size - 1] = static_cast<digit_t>(widedigit);
101
102    quotient[i - 1] = static_cast<digit_t>(quotient_temp);
103
104    if (widedigit < 0) {
105        --quotient[i - 1];
106        widedigit = 0;
107
108        for (size_t j = 0; j < divisor_size; ++j) {
109            widedigit += static_cast<double_digit_t>(dividend[i + j - 1]) + divisor[j];
110            dividend[i + j - 1] = static_cast<digit_t>(widedigit);
111            widedigit >>= 32;
112        }
113    }
114 }
115
116 if (remainder != nullptr) {
117     *remainder = uint_t(0);
118
119     for (size_t i = 0; i < divisor_size - 1; ++i) {
120         (*remainder)[i] =
121             (dividend[i] >> shift_size)
122             | (static_cast<double_digit_t>(dividend[i + 1]) << (c_digit_size - shift_size));
123     }
124
125     (*remainder)[divisor_size - 1] = dividend[divisor_size - 1] >> shift_size;
126 }
127
128 return quotient;
129 }

```

Теперь можем определить операторы деления и остатка:

```

1 friend constexpr uint_t operator/(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = divide(lhs, rhs);
3     uint_t less = result * rhs;
4     uint_t greater = (result + 1) * rhs;
5     if (less > lhs || greater <= lhs) {
6         result = 0;
7     }
8     return result;
9 }

```

```

10
11 friend constexpr uint_t operator%(const uint_t& lhs, const uint_t& rhs) {
12     uint_t remainder;
13     divide(lhs, rhs, &remainder);
14     return remainder;
15 }
16
17 constexpr uint_t& operator*=(const uint_t& other) {
18     return *this = *this * other;
19 }
20
21 constexpr uint_t& operator/=(const uint_t& other) {
22     return *this = *this / other;
23 }

```

Так как нам не нужны r-value при делении, то не пишем оптимизации на них.

- Битовые сдвиги: Нам поступает запрос на сдвиг на `size_t shift` бит влево или вправо. Для высокой производительности выполнение операции нужно разбить на два этапа: сдвиг цифр внутри числа, сдвиг битов внутри цифр:

```

1  constexpr uint_t& operator>>=(size_t shift_size) {
2      size_t digit_shift = shift_size >> 5;
3
4      if (digit_shift > 0) {
5          for (size_t i = 0; i < c_digit_number; ++i) {
6              if (i + digit_shift < c_digit_number) {
7                  m_digits[i] = m_digits[i + digit_shift];
8              } else {
9                  m_digits[i] = 0;
10             }
11         }
12     }
13
14     shift_size %= c_digit_size;
15
16     if (shift_size == 0) {
17         return *this;
18     }
19
20     for (size_t i = 0; i + digit_shift < c_digit_number; ++i) {
21         m_digits[i] >>= shift_size;
22
23         if (i + 1 < c_digit_number) {
24             m_digits[i] |= m_digits[i + 1] << (c_digit_size - shift_size);
25         }
26     }
27
28     return *this;
29 }
30
31 constexpr uint_t& operator<<=(size_t shift_size) {
32     size_t digit_shift = shift_size >> 5;
33
34     if (digit_shift > 0) {
35         for (size_t i = c_digit_number; i > 0; --i) {
36             if (i > digit_shift) {
37                 m_digits[i - 1] = m_digits[i - digit_shift - 1];
38             } else {
39                 m_digits[i - 1] = 0;
40             }
41         }
42     }
43
44     shift_size %= c_digit_size;
45
46     if (shift_size == 0) {
47         return *this;
48     }
49
50     for (size_t i = c_digit_number; i > digit_shift; --i) {

```

```

51         m_digits[i - 1] <= shift_size;
52
53         if (i - 1 > 0) {
54             m_digits[i - 1] |= m_digits[i - 2] >> (c_digit_size - shift_size);
55         }
56     }
57
58     return *this;
59 }

```

В обоих методах в концах двигаем недостающие биты из соседней цифры, если она существует. Определяем внешние friend:

```

1  friend constexpr uint_t operator>>(const uint_t& lhs, const size_t& rhs) {
2      uint_t result = lhs;
3      return result >>= rhs;
4  }
5
6  friend constexpr uint_t operator>>(uint_t&& lhs, const size_t& rhs) {
7      return lhs >>= rhs;
8  }
9
10 friend constexpr uint_t operator<<(const uint_t& lhs, const size_t& rhs) {
11     uint_t result = lhs;
12     return result <<= rhs;
13 }
14
15 friend constexpr uint_t operator<<(uint_t&& lhs, const size_t& rhs) {
16     return lhs <<= rhs;
17 }

```

- Сравнение: Так как используются 20 плюсы, то можно определить оператор <=>, но мы не можем использовать = default, так как тогда сравнение будет с 0 индекса, а не с последнего:

```

1  friend constexpr std::strong_ordering operator<=>(const uint_t& lhs, const uint_t& rhs) {
2      for (size_t i = c_digit_number; i > 0; --i) {
3          if (lhs[i - 1] != rhs[i - 1]) {
4              return lhs[i - 1] <=> rhs[i - 1];
5          }
6      }
7
8      return std::strong_ordering::equal;
9  }

```

Так как мы не определили через default, нам придётся написать и оператор равенства, но он очевиден:

```

1  friend constexpr bool operator==(const uint_t& lhs, const uint_t& rhs) {
2      return lhs.m_digits == rhs.m_digits;
3  }

```

- Битовые операции: Наше число является по сути большой последовательностью бит одного числа, поэтому битовые операции выполняются поэлементно:

```

1  constexpr uint_t& operator^=(const uint_t& other) {
2      for (size_t i = 0; i < c_digit_number; ++i) {
3          m_digits[i] ^= other[i];
4      }
5
6      return *this;
7  }
8
9  constexpr uint_t& operator|=(const uint_t& other) {
10     for (size_t i = 0; i < c_digit_number; ++i) {
11         m_digits[i] |= other[i];
12     }
13
14     return *this;
15 }

```

```

16
17 constexpr uint_t& operator&=(const uint_t& other) {
18     for (size_t i = 0; i < c_digit_number; ++i) {
19         m_digits[i] &= other[i];
20     }
21
22     return *this;
23 }

```

Определяем внешние friend:

```

1 friend constexpr uint_t operator^(const uint_t& lhs, const uint_t& rhs) {
2     uint_t result = lhs;
3     return result ^= rhs;
4 }
5
6 friend constexpr uint_t operator^(uint_t&& lhs, const uint_t& rhs) {
7     return lhs ^= rhs;
8 }
9
10 friend constexpr uint_t operator^(const uint_t& lhs, uint_t&& rhs) {
11     return rhs ^= lhs;
12 }
13
14 friend constexpr uint_t operator^(uint_t&& lhs, uint_t&& rhs) {
15     return lhs ^= rhs;
16 }
17
18 friend constexpr uint_t operator|(const uint_t& lhs, const uint_t& rhs) {
19     uint_t result = lhs;
20     result |= rhs;
21     return result;
22 }
23
24 friend constexpr uint_t operator|(uint_t&& lhs, const uint_t& rhs) {
25     lhs |= rhs;
26     return lhs;
27 }
28
29 friend constexpr uint_t operator|(const uint_t& lhs, uint_t&& rhs) {
30     rhs |= lhs;
31     return rhs;
32 }
33
34 friend constexpr uint_t operator|(uint_t&& lhs, uint_t&& rhs) {
35     lhs |= rhs;
36     return lhs;
37 }
38
39 friend constexpr uint_t operator&(const uint_t& lhs, const uint_t& rhs) {
40     uint_t result = lhs;
41     result &= rhs;
42     return result;
43 }
44
45 friend constexpr uint_t operator&(uint_t&& lhs, const uint_t& rhs) {
46     lhs &= rhs;
47     return lhs;
48 }
49
50 friend constexpr uint_t operator&(const uint_t& lhs, uint_t&& rhs) {
51     rhs &= lhs;
52     return rhs;
53 }
54
55 friend constexpr uint_t operator&(uint_t&& lhs, uint_t&& rhs) {
56     lhs &= rhs;
57     return lhs;
58 }

```

- Унарные инкремент и декремент: Определим вспомогательные приватные методы для увеличения/уменьшения числа на 1, скопировав код `y += /-=` соответственно:

```

1 constexpr void increment() {
2     for (size_t i = 0; i < c_digit_number; ++i) {
3         m_digits[i] += 1;
4
5         if (m_digits[i] != 0) {
6             break;
7         }
8     }
9 }
10
11 constexpr void decrement() {
12     for (size_t i = 0; i < c_digit_number; ++i) {
13         digit_t temp = m_digits[i];
14         m_digits[i] -= 1;
15
16         if (temp >= m_digits[i]) {
17             break;
18         }
19     }
20 }

```

То есть мы пытаемся прибавить/вычесть остаток, пока не найдём хотя бы одну цифру, которая не переполнится от этой операции.

Теперь сами унарные операции. В C++ они бывают двух видов: префиксные и постфиксные. Они разделяются типом `int` в аргументе:

```

1 [[nodiscard("Optimize unary operator usage")]]
2 constexpr uint_t
3     operator++(int) {
4         uint_t result = *this;
5         increment();
6         return result;
7 }
8
9 constexpr uint_t& operator++() {
10     increment();
11     return *this;
12 }
13
14 [[nodiscard("Optimize unary operator usage")]]
15 constexpr uint_t
16     operator--(int) {
17         uint_t result = *this;
18         decrement();
19         return result;
20 }
21
22 constexpr uint_t& operator--() {
23     decrement();
24     return *this;
25 }

```

Дали атрибуты `nodiscard` постфиксным операторам, чтобы пользователь эффективно использовал унарные операции.

- Конвертация в стандартные типы: Два варианта:

1. Хотим обрезать наш тип до стандартных целочисленных типов. Тогда заполняем требуемый тип битами из `m_digits`:

```

1 template<typename T>
2 requires is_convertible_to<T, digit_t>
3 constexpr T convert_to() const {
4     size_t shift_size = sizeof(T) * c_bits_in_byte;
5     size_t digits_number = shift_size / c_digit_size;
6

```



```

7     if (digits_number == 0) {
8         return static_cast<T>(m_digits[0]);
9     }
10
11     T result = 0;
12
13     for (size_t i = 0; i < c_digit_number && i < digits_number; ++i) {
14         result |= static_cast<T>(m_digits[i]) << (i * c_digit_size);
15     }
16
17     return result;
18 }

```

2. Хотим получить всё число. Так как ни один стандартный тип такого размера не поддерживает, то будем переводить наше число в строку обычным делением:

```

1  template<typename T>
2  constexpr T convert_to() const;
3
4  template<>
5  constexpr std::string convert_to() const {
6      std::string result;
7      uint_t clone_of_this = *this;
8
9      do {
10         uint_t remainder;
11         clone_of_this = divide(clone_of_this, 10, &remainder);
12         result.push_back(remainder.m_digits[0] + '0');
13     } while (clone_of_this > 0);
14
15     std::reverse(result.begin(), result.end());
16     return result;
17 }

```

Заметим, что мы специально сделали шаблон, а потом его специализировали, чтобы можно было использовать два варианта одинаково:

```

1  uint_t a = ...;
2  size_t b = a.convert_to<size_t>();
3  std::string s = a.convert_to<std::string>();

```

## 2.3 Главный тип

Независимо от того, выбрали мы писать свою длинку или использовать `boost::multiprecision`, нам нужно задать главный целочисленный тип, на котором мы будем в дальнейшем работать.

Так как обычно для алгоритмов шифрования на эллиптических кривых мы будем использовать эллиптические кривые, предложенные NIST, то нам нужно выбрать количество бит, которое в два раза больше количества бит используемых чисел, чтобы числа не переполнялись при умножении. В данном гайде будем ориентироваться на NIST P-256, поэтому нам понадобится 512 бит в нашей длинке.

Два варианта:

1. Для бустовской длинки:

```

1  #include "boost/multiprecision/cpp_int.hpp"
2  #include "boost/multiprecision/fwd.hpp"
3
4  using uint = boost::multiprecision::uint512_t;

```

2. Для самописной:

```

1  #include "long-arithmetic.h"
2
3  using uint = uint_t<512>;

```

## 3 Поле

Наши поля - это простые конечные поля, значит можно представить их в виде вычетов по простому модулю. Но чем мы будем оперировать? Элементами поля. То есть нам нужен класс элементов поля. Но они должны как-то знать свой простой модуль, который будет одинаков у большинства элементов этого поля, поэтому надо как-то ввести общий элемент для множества объектов. Главная идея: создать класс поля, который сам будет оркестрировать элементами поля: создавать элементы поля и давать им общий простой модуль.

Итого нужно реализовать два класса:

### 3.1 Элемент поля

Так как нужно раздавать простой модуль на множество объектов, то одним из лучших решений будет делать это через умный указатель на константный объект. так как мы не хотим, чтобы наш класс создавали извне, кроме класса поля, то сделаем класс поля другим, а конструктор приватным.

#### 3.1.1 Каркас

```
1 class FieldElement {
2     friend class Field;
3
4     FieldElement(const uint& value, std::shared_ptr<const uint> modulus);
5     uint m_value;
6     std::shared_ptr<const uint> m_modulus;
7 };
```

Так как поступившее число может быть не меньше модуля, то определим приватный метод нормализации:

```
1 uint FieldElement::normalize(const uint& value, std::shared_ptr<const uint> modulus) {
2     return value % *modulus;
3 }
```

Итого:

```
1 class FieldElement {
2     friend class Field;
3
4     FieldElement(const uint& value, std::shared_ptr<const uint> modulus) :
5         m_value(normalize(value, modulus)), m_modulus(std::move(modulus)) {};
6     uint m_value;
7     std::shared_ptr<const uint> m_modulus;
8 };
9
```

#### 3.1.2 Методы

- Стандартным образом определяем по 4 дружественных оператора на каждую операцию  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<<$  через их  $+$  версии. Заметим, что нельзя определить операторы  $>>$ ,  $>>=$ , так как это нарушает арифметику простого поля, потому что деление в поле - это умножение на обратный к делителю.
- Метод отрицания — это просто модуль - значение:

```
1 FieldElement FieldElement::operator-() const {
2     return FieldElement(*m_modulus - m_value, m_modulus);
3 }
```

- Возведение в степень uint — стандартный double-and-add:

```
1 template<typename T>
2 T fast_pow(const T& value, const uint& power) {
3     if ((power & 1) != 0) {
4         if (power == 1) {
5             return value;
6         }
7     }
```

```

8         return value * fast_pow<T>(value, power - 1);
9     }
10
11     T temp = fast_pow<T>(value, power >> 1);
12     return temp * temp;
13 }
14
15 FieldElement FieldElement::pow(const FieldElement& element, const uint& power) {
16     return fast_pow<FieldElement>(element, power);
17 }

```

- Обратный элемент по простому модулю можно найти по расширенному алгоритму Евклида. Не буду себя утруждать объяснениями такого базового алгоритма:

```

1  template<typename T>
2  static T extended_modular_gcd(const T& a, const T& b, T& x, T& y, const T& modulus) {
3      if (b == 0) {
4          x = 1;
5          y = 0;
6          return a;
7      }
8
9      T x1, y1;
10     T d = extended_modular_gcd<T>(b, a % b, x1, y1, modulus);
11     x = y1;
12     T temp = y1 * (a / b);
13
14     while (x1 < temp) {
15         x1 += modulus;
16     }
17
18     y = x1 - temp;
19     return d;
20 }
21
22 template<typename T>
23 T inverse_modulo(const T& value, const T& modulus) {
24     T result, temp;
25     extended_modular_gcd<T>(value, modulus, result, temp, modulus);
26     return result;
27 }
28
29 void FieldElement::inverse() {
30     m_value = inverse_modulo<uint>(m_value, *m_modulus);
31 }

```

- Заметим, что для сложения и вычитания достаточно не деления, а простого сравнения для поддержания инварианта:

```

1  FieldElement& FieldElement::operator+=(const FieldElement& other) {
2      m_value += other.m_value;
3
4      if (m_value >= *m_modulus) {
5          m_value -= *m_modulus;
6      }
7
8      return *this;
9  }
10
11 FieldElement& FieldElement::operator-=(const FieldElement& other) {
12     if (m_value < other.m_value) {
13         m_value += *m_modulus;
14     }
15
16     m_value -= other.m_value;
17
18     return *this;
19 }

```

- Битовый сдвиг влево производим по одному, чтобы случайно не получить переполнение:

```

1 FieldElement& FieldElement::operator<<=(const uint& shift) {
2     for (size_t i = 0; i < shift; ++i) {
3         m_value <<= 1;
4
5         if (m_value >= *m_modulus) {
6             m_value -= *m_modulus;
7         }
8     }
9
10    return *this;
11 }

```

- Сравнение будет в двух вариантах, так как длинка от буста не поддерживает оператор <=>:

```

1 friend bool operator==(const FieldElement& lhs, const FieldElement& rhs) {
2     return lhs.m_value == rhs.m_value;
3 }
4
5 #ifdef ECG_USE_BOOST
6 friend bool operator<(const FieldElement& lhs, const FieldElement& rhs) {
7     return lhs.m_value < rhs.m_value;
8 }
9
10 friend bool operator>(const FieldElement& lhs, const FieldElement& rhs) {
11     return lhs.m_value > rhs.m_value;
12 }
13
14 friend bool operator<=(const FieldElement& lhs, const FieldElement& rhs) {
15     return lhs.m_value <= rhs.m_value;
16 }
17
18 friend bool operator>=(const FieldElement& lhs, const FieldElement& rhs) {
19     return lhs.m_value >= rhs.m_value;
20 }
21
22 friend bool operator!=(const FieldElement& lhs, const FieldElement& rhs) {
23     return lhs.m_value != rhs.m_value;
24 }
25 #else
26 friend std::strong_ordering operator<=>(const FieldElement& lhs, const FieldElement& rhs) {
27     return lhs.m_value <=> rhs.m_value;
28 }
29 #endif

```

- Делаем публичный метод is\_invertible, который проверяет, обратим ли элемент в поле. Так как в простом поле любой ненулевой элемент обратим, то

```

1 bool FieldElement::is_invertible() const {
2     return m_value != 0;
3 }

```

- Прописываем геттеры для модуля и значения и идём дальше.

## 3.2 Поле

Поле будем создавать от простого модуля в uint или в виде строки:

```

1 class Field {
2 public:
3     Field(const uint& modulus);
4
5 private:
6     std::shared_ptr<const uint> m_modulus;
7 };

```

Заметим, что конвертация от строки к uint произойдёт автоматически, если у uint есть не explicit конструктор от строки, и можно будет писать так:

```
1 Field F("0xFFFFFFFF123");
```

Но у бустовской длинки нет конструктора от строки, поэтому заведём отдельный дефайн ECG\_USE\_BOOST, если мы использовали длинку от буста, и определим новый конструктор:

```
1 class Field {
2 public:
3 #ifdef ECG_USE_BOOST
4     Field(const char* str);
5 #endif
6     Field(const uint& modulus);
7
8 private:
9     std::shared_ptr<const uint> m_modulus;
10 };
```

Теперь мы бы хотели создавать элементы поля. Из-за буста опять в двух вариантах:

```
1 class Field {
2 public:
3 #ifdef ECG_USE_BOOST
4     Field(const char* str);
5     FieldElement element(const char* str) const;
6 #endif
7     Field(const uint& modulus);
8     FieldElement element(const uint& value) const;
9     const uint& modulus() const;
10
11 private:
12     std::shared_ptr<const uint> m_modulus;
13 };
```

где element реализуем как:

```
1 #ifdef ECG_USE_BOOST
2 FieldElement Field::element(const char* str) const {
3     return FieldElement(uint(str), m_modulus);
4 }
5 #endif
6 FieldElement Field::element(const uint& value) const {
7     return FieldElement(value, m_modulus);
8 }
```

Даём оператор сравнения на равенство для поля и геттер поля:

```
1 const uint& Field::modulus() const {
2     return *m_modulus;
3 }
4
5 bool Field::operator==(const Field& other) const {
6     return *m_modulus == *other.m_modulus;
7 }
```

Идём дальше.

## 4 Эллиптические кривые

Задаём те же вопросы, бла бла бла.... Короче, нам снова нужны классы эллиптической кривой, точки эллиптической кривой и умные указатели на общие данные.

### 4.1 Точка эллиптической кривой

В реализованном классе используются некие разные виды координат эллиптической кривой и всё такое, но по замерам тестов, они не дают никакого улучшения по времени, а только замедляют, поэтому здесь будет описан только класс эллиптической кривой с нормальными координатами.

### 4.1.1 Каркас

Точка должна содержать свои координаты, числа  $a$ ,  $b$  из уравнения эллиптической кривой:

$$y^2 = x^3 + ax + b$$

Все эти числа лежат в некотором поле  $\mathbb{F}_p$ , поэтому добавим и его в поля через `shared_ptr`. Она должна знать, является ли она точкой  $\mathcal{O}$ , поэтому добавим поле `is_null`. Также, только у класса эллиптической кривой должен быть доступ к конструктору, а значит каркасом будет:

```
1 class EllipticCurvePoint {
2     using Element = FieldElement;
3
4     friend class EllipticCurve;
5
6     std::shared_ptr<const Element> m_a;
7     std::shared_ptr<const Element> m_b;
8     std::shared_ptr<const Field> m_field;
9
10    Element m_x;
11    Element m_y;
12    bool m_is_null;
13 };
```

### 4.1.2 Методы

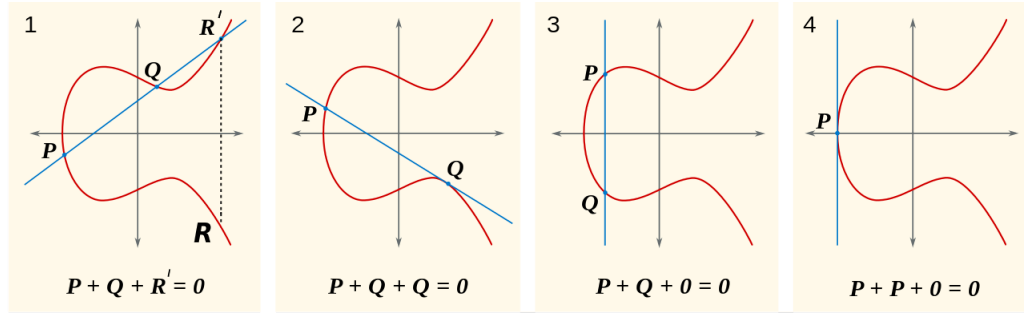
Что можно делать с точкой?

- Создавать: Для создания точки надо передать ей все внутренние поля. Чтобы немного оптимизировать создание, сделаем 4 конструктора в зависимости от r-value-ности координат  $x$  и  $y$ :

```
1 EllipticCurvePoint(const Element& x, const Element& y, std::shared_ptr<const Element> a,
2                   std::shared_ptr<const Element> b, std::shared_ptr<const Field> F,
3                   bool is_null = false) :
4     m_x(x), m_y(y), m_a(std::move(a)),
5     m_b(std::move(b)), m_(std::move(F)), m_is_null(is_null) {};
6
7 EllipticCurvePoint(Element&& x, const Element& y, std::shared_ptr<const Element> a,
8                   std::shared_ptr<const Element> b, std::shared_ptr<const Field> F,
9                   bool is_null = false) :
10    m_x(std::move(x)), m_y(y), m_a(std::move(a)),
11    m_b(std::move(b)), m_(std::move(F)), m_is_null(is_null) {};
12
13 EllipticCurvePoint(const Element& x, Element&& y, std::shared_ptr<const Element> a,
14                   std::shared_ptr<const Element> b, std::shared_ptr<const Field> F,
15                   bool is_null = false) :
16    m_x(x), m_y(std::move(y)), m_a(std::move(a)),
17    m_b(std::move(b)), m_(std::move(F)), m_is_null(is_null) {};
18
19 EllipticCurvePoint(Element&& x, Element&& y, std::shared_ptr<const Element> a,
20                   std::shared_ptr<const Element> b, std::shared_ptr<const Field> F,
21                   bool is_null = false) :
22    m_x(std::move(x)), m_y(std::move(y)), m_a(std::move(a)),
23    m_b(std::move(b)), m_(std::move(F)), m_is_null(is_null) {};
```

Заметим, что мы муваем умные указатели, чтобы атомики внутри них не производили лишнюю работу. Для точки бесконечности мы не поддерживаем никакой специальный вид координат для ускорения работы. То есть имеем "ленивое" обновление координат.

- Складывать: Чтобы сложить две различные точки, надо провести через них прямую на плоскости и посмотреть, какую точку на эллиптической кривой она пересекла. Тогда симметричная ей точка, относительно оси  $Ox$  и будет результатом сложения. Если прямая не пересекла никакой точку на кривой, то считаем, что сумма равна точке  $\infty$ . Чтобы сложить точку с собой, надо провести касательную через неё и повторить алгоритм выше. Точка  $\infty$  будет нулём в данной группе.



Заметим, что при таком задании операции '+' обратная точка в группе - это точка с противоположной координатой  $y$ . С помощью уравнений выразим операцию сложения в нормальных координатах:  $A, B, C \in \mathbb{E}(\mathbb{F}_p)$ ,  $A = (x_1, y_1)$ ,  $B = (x_2, y_2)$ ,  $C = (x_3, y_3)$  и  $A + B = C$ , рассмотрим 3 случая:

1.  $x_1 \neq x_2$ . Обозначим  $k := \frac{y_1 - y_2}{x_1 - x_2}$ . Пусть  $C := A + B$  и  $C = (x_3, y_3)$ . Тогда

$$x_3 = k^2 - x_1 - x_2$$

$$y_3 = k(x_1 - x_3) - y_1$$

2.  $x_1 = x_2$ ,  $y_1 = -y_2$ . Тогда  $A + B = \infty$ .

3.  $A = B$ . Обозначим  $k := \frac{3x_1^2 + a}{2y_1}$ . Тогда

$$x_3 = k^2 - 2x_1$$

$$y_3 = k(x_1 - x_3) - y_1$$

Определим приватный метод `twice` для удвоения точки. Заметим, что если  $y = 0$ , то удвоение точки даст ноль:

```

1 void twice() {
2     if (m_is_null) {
3         return;
4     }
5
6     if (!m_y.is_invertible()) {
7         m_is_null = true;
8         return;
9     }
10
11     const Element k = (m_field->element(3) * Element::pow(m_x, 2) + *m_a) / (m_y << 1);
12     const Element x = Element::pow(k, 2) - (m_x << 1);
13     m_y = k * (m_x - x) - m_y;
14     m_x = x;
15 }

```

Значит сложение точки будет:

```

1 EllipticCurvePoint& operator+=(const EllipticCurvePoint& other) {
2     if (m_is_null) {
3         return *this = other;
4     } else if (other.m_is_null) {
5         return *this;
6     }
7
8     if (m_x == other.m_x) {
9         if (m_y != other.m_y) {
10             m_is_null = true;
11         } else {
12             twice();
13         }
14
15         return *this;
16     }
17 }

```

```

18     const Element k = (other.m_y - m_y) / (other.m_x - m_x);
19     const Element x = Element::pow(k, 2) - m_x - other.m_x;
20     m_y = k * (m_x - x) - m_y;
21     m_x = x;
22
23     return *this;
24 }

```

- Вычитать. Так как мы поняли, что

```

1 void negative() {
2     m_y = -m_y;
3 }

```

то отрицанием и вычитанием будут:

```

1 EllipticCurvePoint operator-() const {
2     EllipticCurvePoint result = *this;
3     result.negative();
4     return result;
5 }
6
7 EllipticCurvePoint& operator--(const EllipticCurvePoint& other) {
8     EllipticCurvePoint temp = other;
9     temp.negative();
10    return *this += temp;
11 }
12
13 EllipticCurvePoint& operator--(EllipticCurvePoint&& other) {
14     other.negative();
15     return *this += other;
16 }

```

- Умножаться на натуральное число.

Умножение на число  $n$  - это сложение  $n$  раз точки с собой. Есть несколько этапов:

1. Представить число  $n$  в *non-adjacent form* (NAF) форме - это уникальное представление числа в двоичном основании, где есть цифра  $-1$ . Такое представление помогает достичь минимального веса Хэмминга (количество цифр, отличных от 0), что заметно ускорит умножение точки на число, если удвоение точки будет быстрее сложения. Алгоритм перевода из двоичной системы исчисления в NAF:

---

**Algorithm 1:** NAF from binary

---

**Data:**  $n = (n_{m-1}n_{m-2} \dots n_0)_2$   
**Result:**  $z = (z_m z_{m-1} \dots z_1 z_0)_{NAF}$   
 $i \leftarrow 0$   
**while**  $n > 0$  **do**  
    **if**  $n$  is odd **then**  
         $z_i \leftarrow 2 - (n \bmod 4)$   $n \leftarrow n - z_i$   
    **else**  
         $z_i \leftarrow 0$   
    **end**  
     $n \leftarrow n/2$   
     $i \leftarrow i + 1$   
**end**

---

Есть версия данного алгоритма, которая основана на битовых операциях:



---

**Algorithm 2:** NAF from binary by bit operations

---

**Data:**  $x = (x_{m-1}x_{m-2} \dots n_0)_2$   
**Result:**  $z = (z_m z_{m-1} \dots z_1 z_0)_{NAF}$   
 $xh \leftarrow x \gg 1$   
 $x3 \leftarrow x + xh$   
 $c \leftarrow xh \text{ XOR } x3$   
 $pb \leftarrow x3 \text{ AND } c$   
 $nb \leftarrow xh \text{ AND } c$   
 $z \leftarrow pb - nb$

---

Есть улучшенная версия данного представления:

**Определение 3.** Пусть  $w \geq 2$  - положительное целое число. Тогда wNAF представление положительного числа  $k$  - это представление  $k = \sum_{i=0}^{l-1} k_i 2^i$ , где каждый ненулевой коэффициент  $k_i$  нечётный,  $|k_i| < 2^{w-1}$ ,  $k_{l-1} \neq 0$  и хотя бы одна ненулевая цифра есть в каждой подпоследовательности из  $w$  цифр. Длина такого представления равна  $l$ .

**Теорема 4** (Свойства wNAF). Пусть  $k$  - положительное целое число, тогда

- (a)  $k$  имеет уникальное представление wNAF, которое обозначается  $NAF_w(k)$
- (b)  $NAF_2(k) = NAF(k)$ , где  $NAF(k)$  - это классическое бинарное представление в NAF
- (c) Длина  $NAF_w(k)$  больше длины  $(k)_2$  хотя бы на 1 цифру.
- (d) Средняя плотность ненулевых цифр в wNAF представлении длины  $l$  примерно  $\frac{1}{w+1}$ .

Алгоритм вычисления wNAF формы:

---

**Algorithm 3:** Computing wNAF

---

**Data:**  $w, k \in \mathbb{N}$   
**Result:**  $NAF_w(k)$   
 $i \leftarrow 0$   
**while**  $k \geq 1$  **do**  
    **if**  $k$  is odd **then**  
         $k_i \leftarrow k \pmod{2^w}$   $k \leftarrow k - k_i$   
    **else**  
         $k_i \leftarrow 0$   
    **end**  
     $k_i \leftarrow k/2$   
     $i \leftarrow i + 1$   
**end**  
**return**  $(k_{i-1}, \dots, k_1, k_0)$

---

2. Теперь напомним алгоритм вычисления кратной точки:

---

**Algorithm 4:** wNAF for computing kP

---

**Data:**  $w, k \in \mathbb{N}, P \in \mathbb{E}(\mathbb{F}_p)$   
**Result:**  $Q := kP$   
 $[k_i] \leftarrow NAF_w(k)$   
Compute  $P_i := iP$  for  $i \in \{1, 3, 5, \dots, 2^{w-1} - 1\}$   
 $Q \leftarrow \mathcal{O}$   
**for**  $i$  from  $l - 1$  to 0 **do**  
     $Q \leftarrow 2Q$   
    **if**  $k_i \neq 0$  **then**  
        **if**  $k_i > 0$  **then**  
             $Q \leftarrow Q + P_{k_i}$   
        **else**  
             $Q \leftarrow Q - P_{-k_i}$   
        **end**  
    **end**  
**end**  
**return**  $Q$

---

Имплементируем NAF как шаблон и подружим его с классом точки для быстрого зануления и удвоения:

```
1 constexpr size_t c_width = 3;
2
3 struct Coefficient {
4     uint16_t value;
5     bool is_negative;
6 };
7
8 using WnafForm = std::vector<Coefficient>;
9
10 constexpr uint16_t c_mask_modulo_2_pow_w = (1 << c_width) - 1;
11
12 WnafForm get_wnaf(uint value) {
13     WnafForm result;
14
15     while (value > 0) {
16         if ((value & 0b1) == 1) {
17             uint16_t coef_value = value.convert_to<uint16_t>() & c_mask_modulo_2_pow_w;
18
19             if (coef_value >= (1 << (c_width - 1))) {
20                 coef_value = (1 << c_width) - coef_value;
21                 result.push_back({.value = coef_value, .is_negative = true});
22                 value += coef_value;
23             } else {
24                 result.push_back({.value = coef_value, .is_negative = false});
25                 value -= coef_value;
26             }
27         } else {
28             result.push_back({.value = 0, .is_negative = false});
29         }
30
31         value >>= 1;
32     }
33
34     return result;
35 }
36
37 constexpr size_t c_k_number = static_cast<size_t>(1) << (c_width - 2);
38
39 template<typename T>
40 T wnaf_addition(T value, const uint& n) {
41     WnafForm wnaf_form = get_wnaf(n);
42     T two_value = value + value;
43     std::vector<T> k_values = {value};
44
45     for (size_t i = 1; i < c_k_number; ++i) {
46         k_values.emplace_back(k_values.back() + two_value);
47     }
48
49     value.nullify();
50
51     for (size_t i = wnaf_form.size(); i > 0; --i) {
52         value.twice();
53
54         if (wnaf_form[i - 1].value != 0) {
55             if (!wnaf_form[i - 1].is_negative) {
56                 value += k_values[wnaf_form[i - 1].value >> 1];
57             } else {
58                 value -= k_values[wnaf_form[i - 1].value >> 1];
59             }
60         }
61     }
62
63     return value;
64 }
```

где

```
1 void nullify() {
2     m_is_null = true;
```

```
3 }
```

Тогда умножение точки на число будет:

```
1 EllipticCurvePoint& operator==(const uint& value) {
2     *this = wnaf_addition<EllipticCurvePoint>(*this, value);
3     return *this;
4 }
5
6 EllipticCurvePoint& operator==(const Element& element) {
7     *this = wnaf_addition<EllipticCurvePoint>(*this, element.value());
8     return *this;
9 }
```

- Сравнивать. Точки равны, если они обе нули или если у них совпадают координаты:

```
1 friend bool operator==(const EllipticCurvePoint& lhs, const EllipticCurvePoint& rhs) {
2     return (lhs.m_is_null && rhs.m_is_null) || (lhs.m_x == rhs.m_x && lhs.m_y == rhs.m_y);
3 }
```

- Приватный статический метод создания нулевой точки. За нулевую точку принято считать точку с координатами (0,1):

```
1 static EllipticCurvePoint null_point(const std::shared_ptr<const Element>& a,
2 const std::shared_ptr<const Element>& b, const std::shared_ptr<const Field>& F) {
3     return EllipticCurvePoint(F->element(0), F->element(1), a, b, F, true);
4 }
```

- Стандартным образом определяем по 4 дружественных оператора на каждую операцию  $+$ ,  $-$ ,  $*$  через их  $+$  = версии.
- Делаем геттеры на  $x, y$ .

## 4.2 Эллиптическая кривая

По сути мы повторяем функционал класса поля:

```
1 class EllipticCurve {
2     using Field = field::Field;
3     using Element = field::FieldElement;
4
5 public:
6     EllipticCurve(const Element& a, const Element& b, Field F);
7     EllipticCurve(Element&& a, const Element& b, Field F);
8     EllipticCurve(const Element& a, Element&& b, Field F);
9     EllipticCurve(Element&& a, Element&& b, Field F);
10
11     const Field& get_field() const;
12     const Element& get_a() const;
13     const Element& get_b() const;
14
15 private:
16     std::shared_ptr<const Element> m_a;
17     std::shared_ptr<const Element> m_b;
18     std::shared_ptr<const Field> m_field;
19 };
```

Этот класс должен поддерживать 3 вида создания точек:

- Создание нулевой точки:

```
1 EllipticCurvePoint null_point() const {
2     return EllipticCurvePoint::null_point(m_a, m_b, m_field);
3 }
```

- Создание точки от  $x$  координаты. Но на эллиптической кривой может и не лежать данная координата, поэтому надо проверить, существует ли корень от  $x^3 + ax + b$  в  $\mathbb{F}_p$ . Есть два варианта:

1.  $p \equiv 3 \pmod{4}$
2.  $p \equiv 1 \pmod{4}$

Почти все простые числа от NIST обладают первым свойством, поэтому рассмотрим только его. Но в реализации имплементирован и протестирован и второй случай.

1. Если число  $a$  имеет корень  $x \in \mathbb{F}_p: x^2 = a$ , то  $a$  либо равен 0, тогда корень понятен 0, либо лежит в  $2\mathbb{Z}_{p-1}$ , так как  $\mathbb{F}_p^* \cong \mathbb{Z}_{p-1}$ .

Пусть  $p - 1 = 2k$ . Значит  $k = \frac{p-1}{2}$ . Заметим, что тогда

$$\forall y \in 2\mathbb{Z}_{p-1} \quad \frac{p-1}{2}y = 0$$

Отсюда вытекает критерий:  $\exists x \in \mathbb{F}_p: x^2 = a \Leftrightarrow a^{\frac{p-1}{2}} = 1$ .

2. Пусть  $a^{\frac{p-1}{2}} = 1$ . Так как  $p \equiv 3 \pmod{4}$ , то  $\frac{p-1}{2}$  нечётно, т.е.

$$\frac{p-1}{2} = 2k - 1 \implies \frac{p+1}{2} = 2k$$

Тогда получаем, что

$$\begin{aligned} a^{\frac{p-1}{2}} = 1 &\implies a^{\frac{p-1}{2}} \cdot a = a \\ a = a^{\frac{p+1}{2}} &= a^{2k} = (a^k)^2 \end{aligned}$$

Значит в качестве корня подойдёт  $x := a^k = a^{\frac{p+1}{4}}$ .

Итого:

---

```

1  std::optional<FieldElement> find_root(const FieldElement& value, const Field& field) {
2      if (!value.is_invertible()) {
3          return std::nullopt;
4      }
5
6      const uint& p = field.modulus();
7      const FieldElement one = field.element(1);
8
9      if (FieldElement::pow(value, (p - 1) >> 1) != one) {
10         return std::nullopt;
11     }
12
13     return FieldElement::pow(value, (p + 1) >> 2);
14 }

```

---

Используем std::optional для передачи того, можно ли найти корень или нет. Тогда методами эллиптической кривой будут:

---

```

1  private:
2  std::optional<EllipticCurve::Element> EllipticCurve::find_y(const Element& x) const {
3      Element value = Element::pow(x, 3) + *m_a * x + *m_b;
4      return find_root(value, *m_field);
5  }
6
7  public:
8  std::optional<EllipticCurvePoint> point_with_x_equal_to(const Element& x) const {
9      if (!x.is_invertible()) {
10         return null_point();
11     }
12
13     std::optional<Element> y = find_y(x);
14
15     if (!y.has_value()) {
16         return std::nullopt;
17     }
18
19     return EllipticCurvePoint(x, std::move(y.value()), m_a, m_b, m_field);
20 }
21

```

---

```

22 std::optional<EllipticCurvePoint> point_with_x_equal_to(Element&& x) const {
23     if (!x.is_invertible()) {
24         return null_point();
25     }
26
27     std::optional<Element> y = find_y(x);
28
29     if (!y.has_value()) {
30         return std::nullopt;
31     }
32
33     return EllipticCurvePoint(std::move(x), std::move(y.value()), m_a, m_b, m_field);
34 }

```

3. Создание точки по двум координатам. Надо проверить только, что координаты удовлетворяют уравнению кривой:

```

1 private:
2 bool EllipticCurve::is_valid_coordinates(const Element& x, const Element& y) const {
3     const Element lhs = Element::pow(y, 2);
4     const Element rhs = Element::pow(x, 3) + *m_a * x + *m_b;
5     return lhs == rhs;
6 }
7
8 bool EllipticCurve::is_null_coordinates(const Element& x, const Element& y) const {
9     return x.value() == 0 && y.value() == 1;
10 }
11
12 public:
13 std::optional<EllipticCurvePoint> point(const Element& x, const Element& y) const {
14     if (is_null_coordinates(x, y)) {
15         return null_point();
16     }
17
18     if (!is_valid_coordinates(x, y)) {
19         return std::nullopt;
20     }
21
22     return EllipticCurvePoint(x, y, m_a, m_b, m_field);
23 }
24
25 std::optional<EllipticCurvePoint> point(Element&& x, const Element& y) const {
26     if (is_null_coordinates(x, y)) {
27         return null_point();
28     }
29
30     if (!is_valid_coordinates(x, y)) {
31         return std::nullopt;
32     }
33
34     return EllipticCurvePoint(std::move(x), y, m_a, m_b, m_field);
35 }
36
37 std::optional<EllipticCurvePoint> point(const Element& x, Element&& y) const {
38     if (is_null_coordinates(x, y)) {
39         return null_point();
40     }
41
42     if (!is_valid_coordinates(x, y)) {
43         return std::nullopt;
44     }
45
46     return EllipticCurvePoint(x, std::move(y), m_a, m_b, m_field);
47 }
48
49 std::optional<EllipticCurvePoint> point(Element&& x, Element&& y) const {
50     if (is_null_coordinates(x, y)) {
51         return null_point();
52     }
53
54     if (!is_valid_coordinates(x, y)) {

```

```

55         return std::nullopt;
56     }
57
58     return EllipticCurvePoint(std::move(x), std::move(y), m_a, m_b, m_field);
59 }

```

Теперь, когда мы определили все нужные методы, можем имплементировать алгоритмы шифрования на эллиптических кривых. Главные алгоритмы, которые мы рассмотрим: Эль-Гамаль и ECDSA. Остальные алгоритмы используют реализованные объекты аналогично тому, как мы будем использовать их в предстоящих алгоритмах.

## 5 Эллиптический Эль-Гамаль

Общие данные: простое число  $p$ , эллиптическая кривая  $\mathbb{E}(\mathbb{F}_p)$ , точка на этой кривой  $P \in \mathbb{E}(\mathbb{F}_p)$ , её порядок  $q$ , отображение сообщения на эллиптическую кривую и обратное слева к нему отображение.

Допустим Боб хочет отправить сообщение  $m$  Алисе. Тогда:

1. Алиса генерирует секретный ключ  $n_\alpha: 1 \leq n_\alpha < q$ , публичный ключ:  $A := n_\alpha P$ . Отправляет публичный ключ Бобу.
2. Боб обратимо отображает сообщение  $m$  на эллиптическую кривую:  $M \in \mathbb{E}(\mathbb{F}_p)$ , генерирует одноразовый секретный ключ  $k: 1 \leq k < q$ , вычисляет  $C_1 := kP$ ,  $C_2 := kA + M$  и отправляет пару  $(C_1, C_2)$  Алисе.
3. Алиса вычисляет исходную точку по формуле:

$$\begin{aligned}
 C_2 - n_\alpha C_1 &= \\
 &= kA + M - n_\alpha kP = n_\alpha kP + M - n_\alpha kP = M
 \end{aligned}$$

и отображает её обратно в  $m$

Отображать сообщение, представленное в виде двоичного числа, на эллиптическую кривую можно следующим образом:

1. Пусть наше поле  $\mathbb{F}_p$ , и  $p$  - это достаточно большое число. Обозначим за  $q$  - длину числа  $p$  в битовом представлении.
2. Выберем и зафиксируем число  $l \in (0, \frac{q}{2})$ , и будем в первые  $l$  бит числа записывать наше сообщение  $m$  в двоичном представлении.
3. Заполним оставшиеся биты случайно. Тогда высока вероятность, что полученное число - это координата  $x$  какой-то точки на эллиптической кривой. Если нет, то повторяем этот шаг, пока не получим точку на кривой.
4. Когда Алиса посчитала точку  $M$ , она берёт координату  $x$  этой точки. Тогда первые  $l$  бит значения этой координаты и будут сообщением  $m$ .

Заметим, что отображать сообщения на кривую и обратно не является удобным. В вариации Эль-Гамалья с хешированием, выбирается хеш-функция  $H: \mathbb{E}(\mathbb{F}_p) \rightarrow \{0, 1\}^n$  и  $(C_1, c_2) := (kP, m \oplus H(kA))$ , т.е. сообщение в двоичном виде длины  $\leq n$  XORится с хешем от точки  $kA$ . Тогда Алиса получает исходное сообщение через  $m = c_2 \oplus H(n_\alpha C_1)$ .

Реализуем данный функционал через класс:

### 5.1 Каркас

Запрашиваем нужные общие данные:

```

1  class ElGamal {
2      using Curve = EllipticCurve;
3      using Point = EllipticCurvePoint;
4
5  public:
6      ElGamal(const Curve& curve, const Point& generator, const uint& generator_order);

```

```

7
8 private:
9
10     Curve m_curve;
11     Point m_generator;
12     uint m_generator_order;
13 };

```

## 5.2 Методы

Нам нужны методы

1. Генерации ключей.

Для этого нужен генератор случайных чисел. Обычный мерсен не подойдёт, так как он не предназначен для криптографии. Используем [библиотеку CSPRNG](#) для этих целей. Она использует внутренние генераторы операционной системы. Напишем генератор случайных uint:

```

1 #include "csprng/csprng.hpp"
2
3 constexpr size_t c_size = 512 / sizeof(uint32_t);
4
5 uint generate_random_uint() {
6     duthomhas::csprng rng;
7     std::seed_seq sseq {228};
8     rng.seed(sseq);
9     uint result = 0;
10    uint32_t x = rng(uint32_t());
11    std::vector<uint32_t> values = rng(std::vector<uint32_t>(c_size));
12
13    for (size_t i = 0; i < c_size; ++i) {
14        result <<= 32;
15        result += values[i];
16    }
17
18    return result;
19 }
20
21 uint generate_random_uint_modulo(const uint& modulus) {
22     return generate_random_uint() % modulus;
23 }
24
25 uint generate_random_non_zero_uint_modulo(const uint& modulus) {
26     return generate_random_uint_modulo(modulus - 1) + 1;
27 }

```

Тогда ключи получаем через:

```

1 struct Keys {
2     uint private_key;
3     Point public_key;
4 };
5
6 Keys ElGamal::generate_keys() const {
7     uint private_key = generate_random_non_zero_uint_modulo(m_generator_order);
8     Point public_key = private_key * m_generator;
9     return Keys {.private_key = private_key, .public_key = public_key};
10 }

```

2. Шифрование сообщения. Есть два варианта - нам предоставили уже готовое сообщение  $M \in \mathbb{E}(\mathbb{F}_p)$ , либо надо самим отобразить  $m$  на кривую. Напишем отображение uint на кривую, через вероятностный алгоритм:

```

1 size_t actual_bit_size(uint value) {
2     size_t result = 0;
3
4     while (value != 0) {
5         ++result;

```

```

6         value >>= 1;
7     }
8
9     return result;
10 }
11
12 static std::map<uint, uint> p_zero_mask;
13 static constexpr uint c_full_bits = uint(0) - 1;
14
15 ElGamal::Point ElGamal::map_to_curve(const uint& message) const {
16     const Field& F = m_curve.get_field();
17     const uint& p = F.modulus();
18
19     if (!p_zero_mask.contains(p)) {
20         const size_t l = actual_bit_size(p) >> 1;
21         uint zero_mask = (c_full_bits >> l) << l;
22         p_zero_mask.insert({p, zero_mask});
23     }
24
25     const uint& zero_mask = p_zero_mask.at(p);
26
27     for (;;) {
28         uint x = random::generate_random_uint_modulo(p);
29         x &= zero_mask;
30         x |= message ^ (message & zero_mask);
31         auto opt = m_curve.point_with_x_equal_to(F.element(x));
32
33         if (opt.has_value()) {
34             return opt.value();
35         }
36     }
37 }

```

Так как нам нужно быстро записывать сообщение в половину длины простого числа, то нам нужна нулевая маска для этой половины. Заводим статическую мапу, которая будет хранить предподсчитанные маски, и создаём новые маски, если такого простого числа  $p$  не встречалось за время выполнения.

Отображение обратно с точки на `uint` происходит аналогично:

```

1 uint ElGamal::map_to_uint(const Point& message) const {
2     const uint& p = m_curve.get_field().modulus();
3
4     if (!p_zero_mask.contains(p)) {
5         const size_t l = actual_bit_size(p) >> 1;
6         uint zero_mask = (c_full_bits >> l) << l;
7         p_zero_mask.insert({p, zero_mask});
8     }
9
10    const uint& zero_mask = p_zero_mask.at(p);
11    uint x = message.get_x().value();
12    x ^= (x & zero_mask);
13    return x;
14 }

```

Заметим, что мы обрезаем сообщение, если количество бит в нём больше, чем половина бит от простого модуля. С высокой долей вероятности, этот алгоритм будет работать относительно быстро и при большей заполненности бит простого модуля, поэтому вы можете поэкспериментировать над длиной нулевой маски.

Имея оба отображения, можем имплементировать алгоритм шифрования, описанный ранее:

```

1 struct EncryptedMessage {
2     Point generator_degree;
3     Point message_with_salt;
4 };
5
6 EncryptedMessage encrypt(const Point& message, const Point& public_key) const {
7     const uint k = generate_random_non_zero_uint_modulo(m_generator_order);
8     const Point generator_degree = k * m_generator;
9     const Point message_with_salt = message + k * public_key;

```



```

10     return {.generator_degree = generator_degree, .message_with_salt = message_with_salt};
11 }
12
13 EncryptedMessage encrypt(const uint& message, const Point& public_key) const {
14     Point point_message = map_to_curve(message);
15     return encrypt(point_message, public_key);
16 }

```

3. Дешифрование. По алгоритму, описанному ранее:

```

1 Point decrypt_to_point(const EncryptedMessage& encrypted_message, const uint& private_key) const {
2     return encrypted_message.message_with_salt - private_key * encrypted_message.generator_degree;
3 }
4
5 uint decrypt_to_uint(const EncryptedMessage& encrypted_message, const uint& private_key) const {
6     const Point M = encrypted_message.message_with_salt
7         - private_key * encrypted_message.generator_degree;
8     return map_to_uint(M);
9 }
10

```

Итого сигнатура класса:

```

1 class ElGamal {
2     using Curve = EllipticCurve;
3     using Point = EllipticCurvePoint;
4
5 public:
6     struct Keys {
7         uint private_key;
8         Point public_key;
9     };
10
11     struct EncryptedMessage {
12         Point generator_degree;
13         Point message_with_salt;
14     };
15
16     ElGamal(const Curve& curve, const Point& generator, const uint& generator_order);
17
18     Keys generate_keys() const;
19
20     EncryptedMessage encrypt(const uint& message, const Point& public_key) const;
21     EncryptedMessage encrypt(const Point& message, const Point& public_key) const;
22     Point decrypt_to_point(const EncryptedMessage& encrypted_message, const uint& private_key) const;
23     uint decrypt_to_uint(const EncryptedMessage& encrypted_message, const uint& private_key) const;
24
25 private:
26     Point map_to_curve(const uint& message) const;
27     uint map_to_uint(const Point& message) const;
28
29     Curve m_curve;
30     Point m_generator;
31     uint m_generator_order;
32 };

```

## 6 ECDSA

Реализация вдохновлена [3]

У этого алгоритма есть несколько основных подалгоритмов:

1. Алгоритм генерации основных параметров:

Пользователь выбирает простое число  $p$  и уровень безопасности  $L$ :  $160 \leq L \leq \lceil \log_2 p \rceil$  и  $2^L \geq 4\sqrt{p}$ . На выходе получаем основные параметры эллиптической кривой.

- (a) Выбираем верифицировано случайным образом  $a, b \in \mathbb{F}_p: 4a^3 + 27b^2 \neq 0$ , чтобы они были параметрами эллиптической кривой. Назовём её  $\mathbb{E}(\mathbb{F}_p)$
- (b) Находим  $N := \#\mathbb{E}(\mathbb{F}_p)$
- (c) Проверяем, что существует простое число  $n \geq 2^L: N \equiv 0 \pmod{n}$ , т.е. что у  $N$  в делителях есть большое простое число. Если это условие неверно, то переходим на первый шаг
- (d) Проверяем, что для этого простого числа  $p^k - 1 \not\equiv 0 \pmod{n} \forall k \in \{1, 2, \dots, 20\}$
- (e) Проверим, что  $p \neq n$ , иначе переходим на шаг 1
- (f) Пусть  $h := \frac{N}{n}$
- (g) Генерируем случайную точку  $G' \in \mathbb{E}(\mathbb{F}_p)$  и задаём  $G := hG'$ . Если  $G = \mathcal{O}$ , то повторяем данный шаг.
- (h) Возвращаем  $D := (p, a, b, F, E, G, n, h)$

## 2. Алгоритм генерации ключей:

Пользователь передаёт основные параметры  $D$ . На выходе получаем открытый и закрытый ключи.

- (a) Выбираем случайное число  $d \in \{1, \dots, n-1\}$
- (b) Вычисляем  $Q := dG$
- (c) Возвращаем  $(Q, d)$ , где точка на эллиптической кривой  $Q$  - открытый ключ, а  $d \in \mathbb{N}$  - закрытый ключ

## 3. Алгоритм генерации цифровой подписи:

Пользователь, который имеет основные параметры  $D$  и ключи  $(Q, d)$ , хочет подписать сообщение  $m$ . Пусть  $H$  - криптографическая хеш-функция, результат которой даёт число, битовое представление которого имеет длину не более  $n$ . На выходе получаем подпись  $(r, s)$ :

- (a) Выбираем случайное число  $k \in \{1, \dots, n-1\}$
- (b) Вычисляем точку  $(x_1, y_1) = kG$
- (c) Вычисляем  $r := x_1 \pmod{n}$ . Если  $r = 0$ , то переходим к шагу 1
- (d) Вычисляем  $e := H(m)$
- (e) Вычисляем  $s := k^{-1}(e + dr) \pmod{n}$ . Если  $s = 0$ , то переходим к шагу 1
- (f) Возвращаем  $(r, s)$

## 4. Алгоритм проверки цифровой подписи:

Другой пользователь получает основные параметры  $D$ , хеш-функцию  $H$ , сообщение  $m$ , подпись  $(r, s)$  и открытый ключ  $Q$  от первого пользователя и хочет проверить подпись. На выходе получаем решение о принятии или отклонении подписи:

- (a) Если  $r, s$  - это не целые числа, принадлежащие  $[1, n-1]$ , то отклоняем
- (b) Вычисляем  $e := H(m)$
- (c) Вычисляем  $w := s^{-1} = k(e + dr)^{-1} \pmod{n}$
- (d) Вычисляем  $u_1 := ew \pmod{n}$  и  $u_2 := rw \pmod{n}$
- (e) Вычисляем координаты точки  $X = (x_1, y_1) := u_1G + u_2Q$
- (f) Если  $X = \mathcal{O}$ , то отклоняем
- (g) Вычисляем  $v := x_1 \pmod{n}$
- (h) Если  $v = r$ , то принимаем, иначе отклоняем

Алгоритм генерации основных параметров не нужен при использовании кривых от NIST, но всё же имплементирован в реализации. Рассмотрим ECDSA без генерации основных параметров:

## 6.1 Каркас

Аналогично Эль-Гамалю, мы хотим общие данные, т.е. основные параметры:

```
1 class ECDSA {
2     using Element = FieldElement;
3     using Curve = EllipticCurve;
4     using Point = EllipticCurvePoint;
5
6 public:
7     ECDSA(const Curve& elliptic_curve, const Point& generator, const uint& n, const uint& h);
8
9 private:
10    Curve m_elliptic_curve;
11    Point m_generator;
12    uint m_n;
13    uint m_h;
14 };
```

## 6.2 Методы

Так как это алгоритм для генерации и проверки цифровой подписи, то у него всего 3 метода:

- Генерация ключей: аналогична Эль-Гамалю:

```
1 struct Keys {
2     Point public_key;
3     uint private_key;
4 };
5
6 Keys ECDSA::generate_keys() const {
7     uint d = generate_random_non_zero_uint_modulo(m_n);
8     Point Q = d * m_generator;
9     return {.public_key = Q, .private_key = d};
10 }
```

- Генерация цифровой подписи: это вероятностный алгоритм, поэтому используем конструкцию for(;;) для удобных прыжков:

```
1 struct Signature {
2     uint r;
3     uint s;
4 };
5
6 ECDSA::Signature ECDSA::generate_signature(const uint& private_key, const uint& message) const {
7     const Field F(m_n);
8
9     for (;;) {
10         const Element k = random::generate_random_non_zero_field_element(F);
11
12         const Point P = k * m_generator;
13         const uint& r = P.get_x().value();
14
15         if (r == 0) {
16             continue;
17         }
18
19         const Element edr = F.element(message) + F.element(private_key) * F.element(r);
20         const uint& s = (Element::inverse(k) * edr).value();
21
22         if (s == 0) {
23             continue;
24         }
25
26         return {.r = r, .s = s};
27     }
28
29     return {};
30 }
```

- Проверка цифровой подписи: ничего нового, просто следуем алгоритму:

```

1  bool ECDSA::is_correct_signature(const Point& public_key, const uint& message,
2                                  const Signature& signature) const {
3      const uint& r = signature.r;
4      const uint& s = signature.s;
5
6      if (r == 0 || s == 0) {
7          return false;
8      }
9
10     if (r >= m_n || s >= m_n) {
11         return false;
12     }
13
14     const Field F = Field(m_n);
15     const Element w = Element::inverse(F.element(s));
16     const Element u1 = F.element(message) * w;
17     const Element u2 = F.element(r) * w;
18     const Point X = u1 * m_generator + u2 * public_key;
19
20     if (X.is_zero()) {
21         return false;
22     }
23
24     const uint& v = X.get_x().value();
25     return v == r;
26 }

```

Итого сигнатура класса:

```

1  class ECDSA {
2      using Field = field::Field;
3      using Element = field::FieldElement;
4      using Curve = elliptic_curve::EllipticCurve;
5      using Point = elliptic_curve::EllipticCurvePoint<elliptic_curve::CoordinatesType::Normal>;
6
7  public:
8      struct Keys {
9          Point public_key;
10         uint private_key;
11     };
12
13     struct Signature {
14         uint r;
15         uint s;
16     };
17
18     ECDSA(const Field& field, const Curve& elliptic_curve, const Point& generator, const uint& n,
19           const uint& h);
20
21     Keys generate_keys() const;
22     Signature generate_signature(const uint& private_key, const uint& message) const;
23     bool is_correct_signature(const Point& public_key, const uint& message,
24                              const Signature& signature) const;
25
26 private:
27     Field m_field;
28     Curve m_elliptic_curve;
29     Point m_generator;
30     uint m_n;
31     uint m_h;
32 };

```

## 7 Тестирование

Проведено тестирование на корректность и на устойчивость для всех реализованных в данном руководстве классов, поэтому все имплементированные классы являются корректными. Тесты, как и способы их запустить, находятся в этом же репозитории.

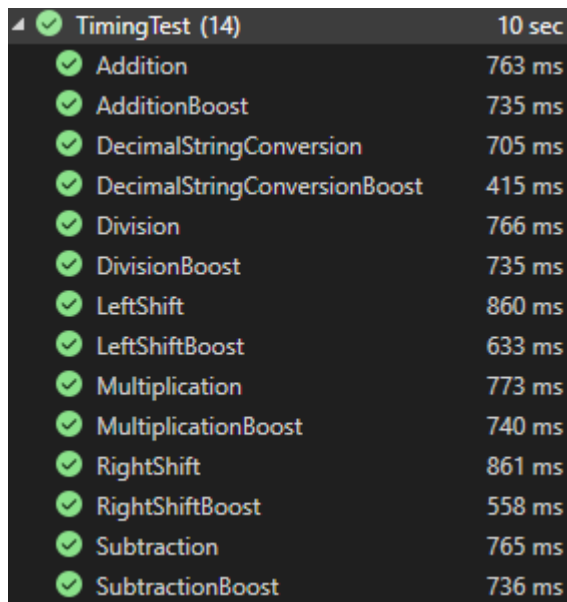
## 7.1 Длинка

Отдельно хочется сравнить скорость выполнения операций у самописного класса `uint_t` относительно скорости длинки от буста. Характеристики машины:

- Процессор: 12th Gen Intel(R) Core(TM) i7-12700K
- Видеокарта: GeForce RTX 3060 Ti Lite Hash Rate
- Оперативная память: DDR4, 8 ГБx2, 2400 МГц,

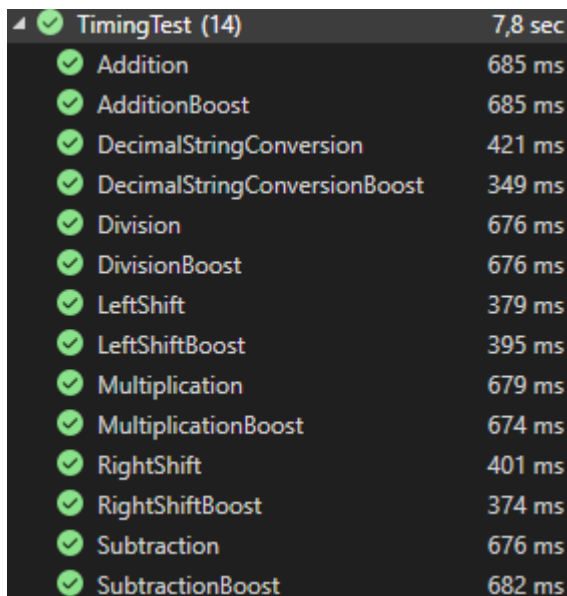
Вот результаты замеров времени работы при одинаковой нагрузке:

- Debug сборка:



TimingTest (14)	10 sec
Addition	763 ms
AdditionBoost	735 ms
DecimalStringConversion	705 ms
DecimalStringConversionBoost	415 ms
Division	766 ms
DivisionBoost	735 ms
LeftShift	860 ms
LeftShiftBoost	633 ms
Multiplication	773 ms
MultiplicationBoost	740 ms
RightShift	861 ms
RightShiftBoost	558 ms
Subtraction	765 ms
SubtractionBoost	736 ms

- Release сборка:



TimingTest (14)	7,8 sec
Addition	685 ms
AdditionBoost	685 ms
DecimalStringConversion	421 ms
DecimalStringConversionBoost	349 ms
Division	676 ms
DivisionBoost	676 ms
LeftShift	379 ms
LeftShiftBoost	395 ms
Multiplication	679 ms
MultiplicationBoost	674 ms
RightShift	401 ms
RightShiftBoost	374 ms
Subtraction	676 ms
SubtractionBoost	682 ms

Мы сравнялись почти во всех операциях. Это достаточно неплохой результат, поэтому можно продолжать использование данного класса, не забывая о возможности дальнейших оптимизаций данного класса.

## 7.2 Эль-Гамаль

Результаты времени работы Эль-Гамала для 1000 сообщений из 128 бит над NIST P-256:

- Debug сборка:

✓ Encryption 1,3 min

- Release сборка:

✓ Encryption 3,6 sec

Так как никто из нормальных людей не использует Эллиптического Эль-Гамала, то придётся сравнить его с [Эллиптическим Диффи-Хеллманом](#), который написан на расте:

```
running 1 test
test tests::bench_cipher ... bench: 374,617,626 ns/iter (+/- 9,080,345)
```

То есть он выполнился за  $\sim 375$  мс ( $\pm 9$  мс). При одинаковых условиях код выполнился в 10 раз медленнее. Это наверно связано с тем, что в библиотеке растовская имплементация, заточенная под P-256, а у нас C++ и общая имплементация. Поэтому есть возможности для улучшения.

## 7.3 ECDSA

Результаты времени работы Эль-Гамала для 1000 сообщений из 128 бит над NIST P-256:

- Debug сборка:

✓ Verification 1,2 min

- Release сборка:

✓ Verification 3,4 sec

Раз уж мы использовали [эту библиотеку](#), то можно сравнить именно с её ECDSA:

```
running 1 test
test tests::bench_cipher ... bench: 372,761,879 ns/iter (+/- 26,586,917)
```

Как видим, результаты  $\sim 372$  мс ( $\pm 26$  мс). При одинаковых условиях аналогично предыдущему случаю: в 10 раз медленнее.

## 8 Специализация

Остановимся на конкретной кривой и генераторе, а именно [NIST P-256](#). Будем писать однофайловое решение шифрования эллиптическим Эль-Гамалем. Начнём!

## 8.1 Поле

Так как нам нужно одно конкретное поле, то можно соединить классы `FieldElement` и `uint_t` в одно целое, забыв о классе `Field`:

```
1 class F_256 {
2     static constexpr size_t c_bits = 512;
3     static constexpr size_t c_bits_in_byte = 8;
4     static constexpr size_t c_digit_size = 32;
5     static constexpr size_t c_digit_number = 16;
6     static constexpr size_t c_half_digit_number = 8;
7
8     using digits = std::array<uint32_t, c_digit_number>;
9     digits m_digits = {};
10 }
```

### 8.1.1 Арифметика

Так как наше простое число имеет 256 бит, то достаточно 512 бит для сдерживания переполнения. Для нормализации числа нам нужно определить само простое число  $p$ , но вот загвоздка — оно тоже 256 битное, а мы не можем определить статическое поле класса самим классом. Поэтому идём на хитрость: расписываем через `constexpr` только цифры простого числа:

```
1 static constexpr digits p_values = {4294967295U, 4294967295U, 4294967295U, 0U, 0U, 0U, 1U, 4294967295U};
```

Тогда для прибавления и вычитания простого числа при нормализации числа определим приватные методы, которые не проверяют инвариант:

```
1 constexpr void add_p_uncheck() {
2     uint32_t carry = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         uint32_t sum = carry + p_values[i];
6         m_digits[i] += sum;
7         carry = (m_digits[i] < sum) || (sum < carry);
8     }
9 }
10
11 constexpr void subtract_p_uncheck() {
12     uint32_t remainder = 0;
13
14     for (size_t i = 0; i < c_digit_number; ++i) {
15         uint32_t prev = m_digits[i];
16         uint32_t sum = p_values[i] + remainder;
17         m_digits[i] -= sum;
18         remainder = (m_digits[i] > prev) || (sum < remainder);
19     }
20 }
```

Теперь модифицируем инкремент и декремент: так как мы считаем, что при инкременте число поддерживало инвариант, то максимум кем оно могло стать — это самым простым модулем  $p$ , поэтому:

```
1 constexpr void increment() {
2     for (size_t i = 0; i < c_digit_number; ++i) {
3         m_digits[i] += 1;
4
5         if (m_digits[i] != 0) {
6             break;
7         }
8     }
9
10    if (m_digits == p_values) {
11        m_digits = {};
12    }
13 }
```

С декрементом немного посложней. Единственный особый переполнение — это декремент от 0. Но мы знаем, чему число тогда станет равно:  $p - 1$ , поэтому заранее определим его:

```

1 static constexpr digits max_mod_p = {4294967294U, 4294967295U, 4294967295U, 0U, 0U, 0U, 1U, 4294967295U};
2
3 constexpr void decrement() {
4     if (*this == 0) {
5         m_digits = max_mod_p;
6         return;
7     }
8
9     for (size_t i = 0; i < c_digit_number; ++i) {
10         uint32_t temp = m_digits[i];
11         m_digits[i] -= 1;
12
13         if (temp >= m_digits[i]) {
14             break;
15         }
16     }
17 }

```

Оператором отрицания будет как обычно вычитание из модуля или 0, если текущее значение 0:

```

1 constexpr F_256 operator-() const {
2     if (!is_invertible()) {
3         return *this;
4     }
5
6     digits result = p_values;
7     uint32_t remainder = 0;
8
9     for (size_t i = 0; i < c_digit_number; ++i) {
10         uint32_t prev = result[i];
11         uint32_t sum = m_digits[i] + remainder;
12         result[i] -= sum;
13         remainder = (result[i] > prev) || (sum < remainder);
14     }
15
16     return F_256(result);
17 }

```

Теперь можно определить операции сложения и вычитания:

```

1 constexpr F_256& operator+=(const F_256& other) {
2     uint32_t carry = 0;
3
4     for (size_t i = 0; i < c_digit_number; ++i) {
5         uint32_t sum = carry + other[i];
6         m_digits[i] += sum;
7         carry = (m_digits[i] < sum) || (sum < carry);
8     }
9
10    if (!is_valid()) {
11        subtract_p_uncheck();
12    }
13
14    return *this;
15 }
16
17 constexpr F_256& operator--(const F_256& other) {
18     uint32_t remainder = 0;
19
20     for (size_t i = 0; i < c_digit_number; ++i) {
21         uint32_t prev = m_digits[i];
22         uint32_t sum = other[i] + remainder;
23         m_digits[i] -= sum;
24         remainder = (m_digits[i] > prev) || (sum < remainder);
25     }
26
27    if (!is_valid()) {
28        add_p_uncheck();
29    }
30
31    return *this;

```



### 8.1.2 Умножение

Чтобы редуцировать число, которое получится после умножения двух чисел из поля, нам надо научиться брать по модулю число  $c$ :  $0 \leq c < p^2$ . У P-256 есть специальный ритуал для этого:

```

1  constexpr void reduce() {
2      F_256 s1({m_digits[0],
3               m_digits[1],
4               m_digits[2],
5               m_digits[3],
6               m_digits[4],
7               m_digits[5],
8               m_digits[6],
9               m_digits[7]});
10     F_256 s2({0, 0, 0, m_digits[11], m_digits[12], m_digits[13], m_digits[14], m_digits[15]});
11     F_256 s3({0, 0, 0, m_digits[12], m_digits[13], m_digits[14], m_digits[15], 0});
12     F_256 s4({m_digits[8], m_digits[9], m_digits[10], 0, 0, 0, m_digits[14], m_digits[15]});
13     F_256 s5({m_digits[9],
14              m_digits[10],
15              m_digits[11],
16              m_digits[13],
17              m_digits[14],
18              m_digits[15],
19              m_digits[13],
20              m_digits[8]});
21     F_256 s6({m_digits[11], m_digits[12], m_digits[13], 0, 0, 0, m_digits[8], m_digits[10]});
22     F_256 s7({m_digits[12], m_digits[13], m_digits[14], m_digits[15], 0, 0, m_digits[9], m_digits[11]});
23     F_256 s8({m_digits[13],
24              m_digits[14],
25              m_digits[15],
26              m_digits[8],
27              m_digits[9],
28              m_digits[10],
29              0,
30              m_digits[12]});
31     F_256 s9({m_digits[14], m_digits[15], 0, m_digits[9], m_digits[10], m_digits[11], 0, m_digits[13]});
32     *this = s1 + s2 + s2 + s3 + s3 + s4 + s5 - s6 - s7 - s8 - s9;
33 }
```

Умные дяди написали, что нужно именно так, поэтому мы как макаки повторяем. Значит умножение мы крадём у `uint_t` и в конце редуцируем:

```

1  friend constexpr F_256 operator*(const F_256& lhs, const F_256& rhs) {
2      F_256 result;
3
4      for (size_t i = 0; i < c_half_digit_number; ++i) {
5          uint64_t u = 0;
6
7          for (size_t j = 0; j < c_half_digit_number; ++j) {
8              u = static_cast<uint64_t>(result[i + j])
9                  + static_cast<uint64_t>(lhs[i]) * static_cast<uint64_t>(rhs[j]) + (u >> c_digit_size);
10             result[i + j] = static_cast<uint32_t>(u);
11         }
12
13         result[i + c_half_digit_number] = static_cast<uint32_t>(u >> c_digit_size);
14     }
15
16     result.reduce();
17     return result;
18 }
```

Заметим, что используется только половины длины числа, так как мы считаем, что `lhs` и `rhs` удовлетворяют инварианту: `lhs, rhs < p`, а количество цифр в  $p$  — это  $8 = c\_half\_digit\_number$ .

### 8.1.3 Инверсия

Так как мы хотим всё и быстро, то у нас нет целочисленного деления. Но как же реализовать инверсию без деления? На помощь приходит алгоритм бинарного расширенного евклида. В его основе лежит житейская мудрость: не можешь использовать обычную арифметику — используй бинарную. Так и поступил автор данного алгоритма. Пруфов работы не будет (точнее, объяснение есть в [4]):

```
1  constexpr void inverse() {
2      constexpr F_256 zero;
3      constexpr F_256 one = 1;
4
5      F_256 u = *this;
6      F_256 v(p_values);
7
8      F_256 x_1 = one;
9      F_256 x_2;
10
11     while (u != one && v != one) {
12         while (u.is_even()) {
13             u >>= 1;
14
15             if (x_1.is_even()) {
16                 x_1 >>= 1;
17             } else {
18                 x_1.add_p_uncheck();
19                 x_1 >>= 1;
20                 assert(x_1.is_valid());
21             }
22         }
23
24         while (v.is_even()) {
25             v >>= 1;
26
27             if (x_2.is_even()) {
28                 x_2 >>= 1;
29             } else {
30                 x_2.add_p_uncheck();
31                 x_2 >>= 1;
32                 assert(x_2.is_valid());
33             }
34         }
35
36         if (u >= v) {
37             u -= v;
38             x_1 -= x_2;
39         } else {
40             v -= u;
41             x_2 -= x_1;
42         }
43     }
44
45     if (u == 1) {
46         *this = x_1;
47     } else {
48         *this = x_2;
49     }
50
51     assert(is_valid());
52 }
```

### 8.1.4 Пролом 4 стены

Нам нужен быстрый доступ к данным, поэтому открываем пользователю методы:

```
1  const uint32_t& first_digit() const {
2      return m_digits[0];
3  }
4
5  constexpr const uint32_t& operator[](size_t pos) const {
6      return m_digits[pos];
7  }
```

```

7 }
8
9 constexpr uint32_t& operator[](size_t pos) {
10     return m_digits[pos];
11 }

```

Да, это не безопасно, зато быстро.

### 8.1.5 Конверсия

У нас нет деления, значит мы можем привести только к 16,8,2-ичным системам счисления. Ограничимся для нашей специализации 16-ричной системой.

Основная сложность - каждая цифра содержит 8 16-ричных символов, поэтому нам понадобятся некоторые лямбды, которые будут упрощать деление числа на хекс-символы.

Так как мы начинаем заполнять со старших битов, то нам нужно пропустить верхние нули. Для цифр это несложно, но вот для скапа нулей внутри цифры нам придётся передать флаг в лямбды.

Итого:

```

1  constexpr std::string convert_to_hex_string() const {
2      std::string result;
3      size_t pos = c_digit_number;
4
5      while (pos > 0 && m_digits[pos - 1] == 0) {
6          --pos;
7      }
8
9      if (pos == 0) {
10         return "0x0";
11     }
12
13     result += "0x";
14
15     auto mini_push = [&](const uint8_t& value) {
16         if (value < 10) {
17             result.push_back(value + '0');
18         } else {
19             result.push_back(value - 10 + 'A');
20         }
21     };
22
23     auto push = [&](const uint32_t& value, bool first_time = false) {
24         size_t shift = 32;
25         uint8_t temp = 0;
26
27         if (first_time) {
28             while (shift > 0 && temp == 0) {
29                 shift -= 4;
30                 temp = (value >> shift) & 0xF;
31             }
32
33             do {
34                 mini_push(temp);
35
36                 if (shift == 0) {
37                     break;
38                 }
39
40                 shift -= 4;
41                 temp = (value >> shift) & 0xF;
42             } while (shift >= 0);
43
44             return;
45         }
46
47         while (shift > 0) {
48             shift -= 4;
49             temp = (value >> shift) & 0xF;
50             mini_push(temp);
51         }

```

Остальные методы без изменений.

Забываем о классе эллиптической кривой и захардкоживаем внутрь класса точки основные параметры:

где  $p_1$  и  $p_2$  - те самые значения, которые мы считали при нахождении корня в поле:

$$p_2 := \frac{p+1}{4}$$

Так как нам больше не нужно держать общие данные, то класс Эль-Гамаль распадается на обычный namespace:

```

7         F_256("0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5"));
8     static constexpr F_256 m_generator_order =
9         "0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551";
10
11     Point map_to_curve(const F_256& message) {
12         for (;;) {
13             F_256 x = generate_random_non_zero_value_modulo(m_generator_order);
14
15             for (size_t i = 0; i < 6; ++i) {
16                 x[i] = message[i];
17             }
18
19             auto opt = Point::point_with_x_equal_to(x);
20
21             if (opt.has_value()) {
22                 return opt.value();
23             }
24         }
25     }
26
27     F_256 map_to_uint(const Point& message) {
28         F_256 result;
29         const F_256& x = message.get_x();
30
31         for (size_t i = 0; i < 6; ++i) {
32             result[i] = x[i];
33         }
34
35         return result;
36     }
37 } // namespace
38
39 struct Keys {
40     F_256 private_key;
41     Point public_key;
42 };
43
44 struct EncryptedMessage {
45     Point generator_degree;
46     Point message_with_salt;
47 };
48
49 constexpr Keys generate_keys() {
50     F_256 private_key = generate_random_non_zero_value_modulo(m_generator_order);
51     Point public_key = private_key * m_generator;
52     return Keys {.private_key = private_key, .public_key = public_key};
53 }
54
55 EncryptedMessage encrypt(const F_256& message, const Point& public_key) {
56     const F_256 k = generate_random_non_zero_value_modulo(m_generator_order);
57     const Point generator_degree = k * m_generator;
58     const Point message_with_salt = map_to_curve(message) + k * public_key;
59     return {.generator_degree = generator_degree, .message_with_salt = message_with_salt};
60 }
61
62 F_256 decrypt(const EncryptedMessage& encrypted_message, const F_256& private_key) {
63     Point M = encrypted_message.message_with_salt - private_key * encrypted_message.generator_degree;
64     return map_to_uint(M);
65 }
66 }; // namespace ElGamal

```

Имплементация функций остаётся прежней, кроме того факта, что мы используем 6 бит из 8, вместо 4 бит, во время отображения сообщения на кривую и обратно. Это позволяет значительно увеличить передаваемое сообщение.

## 8.4 Тестирование

Наконец мы можем побаловаться с шифрованием сообщений через консоль:

```
1 int main() {
```

```

2      std::cout << "Enter hexadecimal message:\n";
3      std::string msg;
4      std::cin >> msg;
5      F_256 message = msg.c_str();
6      std::cout << "Generating keys...\n";
7      auto keys = ElGamal::generate_keys();
8      std::cout << "Encrypting message...\n";
9      auto encrypted_message = ElGamal::encrypt(message, keys.public_key);
10     std::cout << "Decrypting message...\n";
11     auto decrypted_message = ElGamal::decrypt(encrypted_message, keys.private_key);
12     std::cout << "Decrypted message is:\n";
13     std::string decrypted_msg = decrypted_message.convert_to_hex_string();
14     std::cout << decrypted_msg << '\n';
15     std::cout << "Checking correctness...\n";
16
17     if (message != decrypted_message) {
18         std::cout << "Fail ";
19
20         if (msg.ends_with(decrypted_msg.substr(2))) {
21             std::cout << "due to insufficient number of encryption bits\n";
22         } else {
23             std::cout << "due to implementation problems\n";
24         }
25     } else {
26         std::cout << "Success!\n";
27     }
28 }

```

Результаты замеров тестов на 1000 сообщений по 128 бит:

- Debug сборка:

✓ Encryption 3,4 min

- Release сборка:

✓ Encryption 20,8 sec

Шифрование стало в 5.7 раз медленнее, чем было у шаблонного варианта. Я уверен, что это из-за того, что код является однофайловым решением и не может эффективно скомпилироваться и слинковаться. Но всё равно, это был интересный опыт специализации.

## 9 Заключение

За время этого гайда вы ознакомились с основными проблемами при попытке реализации криптографии на эллиптических кривых и узнали способы их решения. Это руководство является всего лишь вершиной айсберга такого вида криптографии, но я надеюсь, что оно поможет сделать первый шаг в данный удивительный мир, где тесно переплетается информатика и математика.

С помощью этой базы уже можно строить протоколы шифрования бинарных сообщений, написав конвертацию к бинарной строке. Или написать хеш-функции ограниченной длины, чтобы подписывать сообщения. В общем, далее можно развивать часть, которая не относится непосредственно к шифрованию на эллиптических кривых, или наоборот ускорять часть, которая непосредственно к этому шифрованию относится.

Также можно рассматривать разные атаки на этот способ шифрования и способы защиты от них.

Сим откланиваюсь.

## Список литературы

- [1] Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. Technical report, Institute for Applied Information Processing and Communications, Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria, 2011.
- [2] Donald E. Knuth. *Art of computer programming*, volume 2. Addison-Wesley Professional, 1985.
- [3] Scott Vanstone Darrel Hakerson, Alfred Menezes. *Guide to Elliptic Curve Cryptography*, chapter Cryptographic Protocols, pages 183–195. Springer-Verlag, 2004.
- [4] Scott Vanstone Darrel Hakerson, Alfred Menezes. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.