Правительство Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
"Национальный исследовательский университет
"Высшая школа экономики"
Московский институт электроники и математики им. А.Н.Тихонова
Департамент прикладной математики

ОТЧЕТ

по дисциплине «Теоретико-числовые методы в криптографии»

Программная реализация алгоритма вычисления квадратного корня в поле $\mathbb{F}_q$ для $q \equiv 2^s + 1 \pmod{2^{s+1}}$

Выполнили студенты гр. СКБ181
Васильева Анастасия Андреевна,
Попов Дмитрий Александрович

Москва, 2022 г.

1 Введение

В данной работе рассматриваются алгоритмы вычисления квадратного корня в конечном поле $\mathbb{F}_q$ для $q \equiv 2^s \pmod{2^{s+1}}$. В общем случае эта задача решается с помощью алгоритмов Тонелли-Шенкса и Чиполлы-Лемера, однако в случае, когда показатель s мал, существуют более эффективные методы решения.

Пусть c – квадратичный вычет в поле $\mathbb{F}_q$. Рассмотрим следующие алгоритмы нахождения квадратного корня из c при малых s:

- При $s = 1$ квадратный корень из c задается как $c^{\frac{(q+1)}{4}}$, так как по критерию Эйлера $\left(c^{\frac{(q+1)}{4}}\right)^2 = c^{\frac{(q+1)}{2}} = c \cdot c^{\frac{(q-1)}{2}} = c$
- При $s = 2$ квадратный корень из c может быть найден с помощью алгоритма Аткина
- При $s = 3$ квадратный корень из c может быть найден с помощью алгоритма Мюллера, а также алгоритма Конга и др.

Далее будет представлен новый метод нахождения квадратного корня в конечном поле, основанный на представленных выше алгоритмах. При малых s он является более эффективным, чем алгоритмы Тонелли-Шенкса и Чиполлы-Лемера, так как он требует только одного возведения в степень. Особенностью данного алгоритма является предварительное вычисление примитивного корня ξ степени 2^s из единицы в $\mathbb{F}_q$. Данное значение будет зафиксировано для конкретного q, за счет чего увеличивается эффективность вычисления квадратных корней в рамках одного поля $\mathbb{F}_q$.

2 Существующие алгоритмы извлечения квадратного корня для малых s

2.1 Алгоритм Аткина ($s = 2$)

Ранее было показано, что для $s = 1$, то есть в случае $q \equiv 3 \pmod 4$, квадратным корнем из a в $\mathbb{F}_q$ будет являться $a^{\frac{(q+1)}{4}}$. Но не существует аналогичного алгоритма с одним возведением в степень для случая $q \equiv 1 \pmod 4$. Однако для такого q есть алгоритм в частном случае, когда $q \equiv 5 \pmod 8$ – это алгоритм Аткина, который также использует одно возведение в степень.

Этот алгоритм использует тот факт, что число 2 является квадратичным невычетом в $\mathbb{F}_q$ в случае $q \equiv 5 \pmod 8$, и, следовательно, $2a$ также является квадратичным невычетом. Таким образом, можно видеть, что по критерию Эйлера $(2a)^{\frac{q-1}{2}} = -1$ и $(2a)^{\frac{q-1}{4}} = \sqrt{-1}$. Этот алгоритм также использует тот факт, что $\left(\sqrt{-1} - 1\right)^2 = -2\sqrt{-1}$.

Доказать правильность алгоритма можно, выполнив все шаги и получив искомый квадратный корень x. Затем возводим x в квадрат и, используя замечания выше, действительно можно будет убедиться, что получится a.

Алгоритм 1 (Аткин) $q \equiv 5 \pmod 8$

Вход: q – порядок конечного поля
a – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = a$ в $\mathbb{F}_q$

1: $b \leftarrow (2a)^{\frac{q-5}{8}}$
2: $i \leftarrow 2ab^2$
3: $x \leftarrow ab(i-1)$
4: **return** x

Здесь и далее приводится реализация алгоритма с использованием SageMath:

```
def algoritm_1(a, q):
    """q = 5 (mod 8)"""
    b = powmod(2 * a, (q - 5) // 8, q)
    i = (2 * a * (b ** 2)) % q
    x = (a * b * (i - 1)) % q
    return x
```

Проверка работы этого и других алгоритмов представлена в разделе Тесты.

2.2 Алгоритм Мюллера ($s = 3$)

Данный алгоритм является частным случаем при $q \equiv 1 \pmod 8$. Здесь будем рассматривать q, удовлетворющие $q \equiv 9 \pmod{16}$.

В этом случае число 2 больше не является квадратичным невычетом в $\mathbb{F}_q$. Однако, поскольку $(2a)^{\frac{q-1}{4}}$ равно 1 или -1, идея Аткина может быть расширена, если у нас есть другой параметр – квадратичный невычет в случае $(2a)^{\frac{q-1}{4}} = 1$ и квадратичный вычет в случае $(2a)^{\frac{q-1}{4}} = -1$.

Алгоритм Мюллера является вероятностным и требует 2 возведения в степень. Вероятностный шаг находит $d \in \mathbb{F}_q$, удовлетворяющий $\eta(d) = -b$, где η такая, что $\eta(d) = 1$, если d является квадратом в $\mathbb{F}_q$, и $\eta(d) = -1$, если нет.

Алгоритм 2 (Мюллер) $q \equiv 9 \pmod{16}$

Вход: q – порядок конечного поля
a – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = a$ в $\mathbb{F}_q$

1: $b \leftarrow (2a)^{\frac{q-1}{4}}$
2: Поиск d: $-b = \eta(d)$
3: $u \leftarrow (2ad^2)^{\frac{q-9}{16}}$
4: $i \leftarrow 2u^2d^2a$
5: $x \leftarrow uda(i-1)$
6: **return** x

```python
def algoritm_2(a, q):
    """q = 9 (mod 16)"""
    b = powmod(2 * a, (q - 1) // 4, q)
    if b == q - 1:
        b = -1

    d = 0
    for j_d in range(2, q):
        if -b == kronecker(j_d, q):
            d = j_d
            break
    if d == 0:
        sys.exit("Can not find d")

    u = powmod(2 * a * (d ** 2), (q - 9) // 16, q)
    i = (2 * (u ** 2) * (d ** 2) * a) % q
    x = (u * d * a * (i - 1)) % q
    return x
```

2.3 Алгоритм Конга

Алгоритм Мюллера требует 2 возведения в степень, однако заметим, что при $(2a)^{\frac{q-1}{4}} = -1$ он будет аналогичен алгоритму Аткина, то есть будет требоваться всего 1 возведение в степень. На этом замечании основан алгоритм Конга, который является усовершенствованной версией алгоритма Мюллера.

Алгоритму Конга требуется в среднем 1,5 возведения в степень. Этот алгоритм также является вероятностным в общем случае.

Алгоритм 3 (Конг) $q \equiv 9 \pmod{16}$

Вход: q – порядок конечного поля
a – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = a$ в $\mathbb{F}_q$

1: $b \leftarrow (2a)^{\frac{q-9}{16}}$
2: $i \leftarrow 2ab^2,\ r \leftarrow i^2$
3: **if** $r = -1$ **then**
4: $\quad x \leftarrow ab(i-1)$
5: **else**
6: $\quad$ Поиск d – невычет в $\mathbb{F}_q$
7: $\quad u \leftarrow bd^{\frac{q-9}{8}}$
8: $\quad i \leftarrow 2u^2d^2a$
9: $\quad x \leftarrow uda(i-1)$
10: **return** x

```python
def algoritm_3(a, q):
    """q = 9 (mod 16)"""
    b = powmod(2 * a, (q - 9) // 16, q)
    i = (2 * a * (b ** 2)) % q
    r = (i ** 2) % q
    if r == q - 1:
        x = (a * b * (i - 1)) % q
    else:
        d = 0
        for j_d in range(2, q):
            if kronecker(j_d, q) == -1:
                d = j_d
                break
        if d == 0:
            sys.exit("Can not find d")

        u = b * powmod(d, (q - 9) // 8, q) % q
        i = (2 * (u ** 2) * (d ** 2) * a) % q
        x = (u * d * a * (i - 1)) % q
    return x
```

3 Новый метод нахождения квадратного корня в поле $\mathbb{F}_q$ при $q \equiv 2^s + 1 \pmod{2^{s+1}}$

Пусть q – степень нечетного простого числа, а s – наибольшее положительное целое число, удовлетворяющее $2^s | q - 1$. Отсюда, так как $\frac{q-1}{2^s} \equiv 1 \pmod 2$, имеем $q \equiv 2^s + 1 \pmod{2^{s+1}}$. То есть для любой нечетной простой степени q существует единственное положительное целое число s, удовлетворяющее $q \equiv 2^s + 1 \pmod{2^{s+1}}$, где s – наибольшее положительное целое число, удовлетворяющее $2^s | q - 1$.

Для заданного квадрата c из $\mathbb{F}_q$ определим

$$b = c^{\frac{q-(2^s+1)}{2^{s+1}}}$$

и

$$\zeta = c^{\frac{q-1}{2^s}} = c \cdot c^{\frac{q-(2^s+1)}{2^s}} = c \cdot \left(c^{\frac{q-(2^s+1)}{2^{s+1}}}\right)^2 = cb^2$$

Поскольку c – квадрат в $\mathbb{F}_q$, то ζ – это примитивный корень из единицы в $\mathbb{F}_q$ степени 2^t для некоторого однозначно определенного $t \leq s - 1$, то есть существует $t < s$ такой, что

$$\zeta^{2^t} = 1, \qquad \zeta^{2^{t-1}} = -1$$

Пусть ξ – примитивный корень из единицы в $\mathbb{F}_q$ степени 2^s, который будет вычислен один раз и зафиксирован. ξ можно вычислить, полагая $\xi = d^{\frac{q-1}{2^s}}$, где d – квадратичный невычет в $\mathbb{F}_q$. Таким образом, данный метод также является вероятностным.

Поскольку $\xi^{2^{s-t}}$ является примитивным корнем из единицы в $\mathbb{F}_q$ степени 2^t, то существуют единственные i и j, определенные по mod 2^t, такие, что

$$\xi^{2^{s-t}} = \zeta^i, \qquad \left(\xi^{2^{s-t}}\right)^j = \zeta$$

Из $\xi^{2^{s-t}} = \zeta^i = \left(\xi^{2^{s-t}}\right)^{ij}$ имеем

$$ij \equiv 1 \pmod{2^t} \quad (*)$$

Теперь рассмотрим новую теорему, которая утверждает, что квадратный корень можно найти, используя одно возведение в степень при заданных условиях.

Теорема 1

Определим u: $u \equiv j(2^t - 1)2^{s-t-1} \pmod{2^{s-1}}$

Тогда квадратный корень из c в $\mathbb{F}_q$ задается как $cb\xi^u$

Доказательство

Положим $x = cb\xi^u$. Тогда

$$x^2 = c \cdot cb^2 \cdot \xi^{2u} = c \cdot \zeta \cdot \xi^{2u}$$

Поскольку по условию $u = j(2^t - 1)2^{s-t-1} + 2^{s-1}k$ для некоторого целого k, то, используя $\xi^{2^s} = 1$, получим

$$\zeta \cdot \xi^{2u} = \left(\xi^{2^{s-t}}\right)^j \cdot \xi^{2u} = \xi^{j \cdot 2^{s-t} + 2u} =$$

$$= \xi^{j \cdot 2^{s-t} + j(2^t - 1)2^{s-t} + 2^s k} =$$

$$= \xi^{j \cdot 2^{s-t} + j(2^t - 1)2^{s-t}} = \xi^{j \cdot 2^{s-t}(1 + 2^t - 1)} =$$

$$= \xi^{j \cdot 2^s} = 1$$

Отсюда $x^2 = c \cdot \zeta \cdot \xi^{2u} = c$

■

Нахождение i или j из уравнения (∗) сложно, если 2^t велико. Таким образом, рассмотренный метод полезен только тогда, когда значение t относительно небольшое.

Далее рассмотрим примеры применения данного метода при малых s.

4 Примеры

4.1 Пример 1 ($s = 1$)

Рассмотрим нахождение квадратного корня из c в $\mathbb{F}_q$ при $q \equiv 3 \pmod 4$. В этом случае:

- $\zeta = c^{\frac{q-1}{2^s}} = c^{\frac{q-1}{2}} = 1$ (по критерию Эйлера)
- $\xi = d^{\frac{q-1}{2^s}} = \xi = d^{\frac{q-1}{2}} = 1$ (по критерию Эйлера)
- $t = 0$, так как $\zeta = 1$
- $u = 0$, так как $t = 0$
- $b = c^{\frac{q-(2^s+1)}{2^{s+1}}} = c^{\frac{q-3}{4}}$

Таким образом, $x = cb\xi^u = c \cdot b = c \cdot c^{\frac{q-3}{4}} = c^{\frac{q+1}{4}}$. Этот случай рассматривался во введении к отчету.

4.2 Пример 2 ($s = 2$)

Рассмотрим нахождение квадратного корня из c в $\mathbb{F}_q$ при $q \equiv 5 \pmod 8$. В этом случае:

- $\zeta = c^{\frac{q-1}{2^s}} = c^{\frac{q-1}{4}} = \pm 1$
- При $\zeta = 1$:
  - $t = 0;\ \ u = 0$
  - $x = cb = c \cdot c^{\frac{q-5}{8}} = c^{\frac{q+3}{8}}$
- При $\zeta = -1$:
  - $t = 1;\ \ i = j = 1;\ \ u = j(2^t - 1)2^{s-t-1} = 1$
  - $x = cb\xi^u = cb\xi$

Приведем алгоритм поиска x на основе этого примера.

Алгоритм 4 $q \equiv 5 \pmod 8$

Вход: q – порядок конечного поля
c – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = c$ в $\mathbb{F}_q$

1: $b \leftarrow c^{\frac{q-5}{8}}$
2: $\zeta \leftarrow cb^2$
3: **if** $\zeta = 1$ **then** $x \leftarrow cb$
4: **else** $x \leftarrow cb\xi$
5: **return** x

```python
def algoritm_4(c, q, xi):
    """q = 5 (mod 8); s = 2"""
    b = powmod(c, (q - 5) // 8, q)
    zeta = (c * (b ** 2)) % q
    if zeta == 1:
        x = (c * b) % q
    else:
        x = (c * b * xi) % q
    return x
```

В этой реализации алгоритма на вход функции подается также ξ, которое вычисляется заранее для заданного q. Алгоритм вычисления ξ и тесты описаны в разделе Тесты.

4.3 Пример 3 ($s = 3$)

Рассмотрим нахождение квадратного корня из c в $\mathbb{F}_q$ при $q \equiv 9 \pmod{16}$:

- $\zeta = c^{\frac{q-1}{2^3}}$; $t = 0, 1, 2$
- ξ – примитивный корень степени 2^3; $\xi^{2^{3-t}} = \zeta^i$ где $i \pmod{2^t}$ при $ij \equiv 1 \pmod{2^t}$
- $u \equiv j(2^t - 1)2^{2-t} \pmod{2^2}$
- $x = cb\xi^u$

Имеем следующие случаи при разных t:

- При $t = 0$ получим $u = 0$, а значит $x = cb$
- При $t = 1$ получим $i = j = 1$ и $u \equiv 2 \pmod{2^2}$, а значит $x = cb\xi^2$
- При $t = 2$ получим из $\xi^2 = \zeta^i$ две пары $(i, j) = (1, 1),\ (3, 3) \pmod{2^2}$ и $u \equiv 3j \pmod{2^2}$
  - При $\xi^2 = \zeta$ получим $x = cb\xi^3$
  - При $\xi^2 = \zeta^3$ (то есть $\zeta = -\xi^2$) получим $x = cb\xi$

Алгоритм 5 $q \equiv 9 \pmod{16}$

Вход: q – порядок конечного поля
c – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = c$ в $\mathbb{F}_q$

1: $b \leftarrow c^{\frac{q-9}{16}}$
2: $\zeta \leftarrow cb^2$
3: **if** $\zeta = 1$ **then** $x \leftarrow cb$
4: **else if** $\zeta = -1$ **then** $x \leftarrow cb\xi^2$
5: **else if** $\zeta = \xi^2$ **then** $x \leftarrow cb\xi^3$
6: **else** $x \leftarrow cb\xi$
7: **return** x

```python
def algoritm_5(c, q, xi):
    """q = 9 (mod 16); s = 3"""
    b = powmod(c, (q - 9) // 16, q)
    zeta = (c * (b ** 2)) % q
    if zeta == 1:
        x = (c * b) % q
    elif zeta == q - 1:
        x = (c * b * (xi ** 2)) % q
    elif zeta == (xi ** 2) % q:
        x = (c * b * (xi ** 3)) % q
    else:
        x = (c * b * xi) % q
    return x
```

4.4 Пример 4 ($s = 4$)

Рассмотрим нахождение квадратного корня из c в $\mathbb{F}_q$ при $q \equiv 17 \pmod{32}$:

- $\zeta = c^{\frac{q-1}{2^4}}$; $t = 0, 1, 2, 3$
- ξ – примитивный корень степени 2^4; $\xi^{2^{4-t}} = \zeta^i$ где $i \pmod{2^t}$ при $ij \equiv 1 \pmod{2^t}$
- $u \equiv j(2^t - 1)2^{3-t} \pmod{2^3}$
- $x = cb\xi^u$

Имеем следующие случаи при разных t:

- При $t = 0$ получим $u = 0$, а значит $x = cb$
- При $t = 1$ получим $\xi^8 = \zeta$ при $\zeta^2 = 1$, то есть $i = j = 1$ и $u \equiv 4 \pmod{2^3}$, а значит $x = cb\xi^4$
- При $t = 2$ получим из $\xi^4 = \zeta^i$ две пары $(i, j) = (1, 1),\ (3, 3) \pmod{2^2}$ и $u \equiv 6j \pmod{2^3}$
    - При $\xi^4 = \zeta$ получим $x = cb\xi^6$
    - При $\xi^4 = \zeta^3$ (то есть $\zeta = -\xi^4$) получим $x = cb\xi^2$
- При $t = 3$ получим из $\xi^2 = \zeta^i$ четыре пары $(i, j) = (1, 1), (3, 3), (5, 5), (7, 7) \pmod{2^3}$ и $u \equiv 7j \pmod{2^3}$
    - При $\xi^2 = \zeta$ получим $x = cb\xi^7$
    - При $\xi^2 = \zeta^3$ (то есть $\zeta = \xi^6$) получим $x = cb\xi^5$
    - При $\xi^2 = \zeta^5$ (то есть $\zeta = -\xi^2$) получим $x = cb\xi^3$
    - При $\xi^2 = \zeta^7$ (то есть $\zeta = -\xi^6$) получим $x = cb\xi$

Алгоритм 6 $q \equiv 17 \pmod{32}$

Вход: q – порядок конечного поля
c – квадрат в $\mathbb{F}_q$

Выход: x – решение для $x^2 = c$ в $\mathbb{F}_q$

1: $b \leftarrow c^{\frac{q-17}{32}}$
2: $X \leftarrow cb,\ \zeta \leftarrow Xb,\ A \leftarrow \xi,\ B \leftarrow A^2,\ C \leftarrow B^2,\ D \leftarrow BC$
3: **if** $\zeta = 1$ **then** $x \leftarrow X$
4: **else if** $\zeta = -1$ **then** $x \leftarrow XC$
5: **else if** $\zeta = B$ **then** $x \leftarrow XAD$
6: **else if** $\zeta = -B$ **then** $x \leftarrow XAB$
7: **else if** $\zeta = C$ **then** $x \leftarrow XD$
8: **else if** $\zeta = -C$ **then** $x \leftarrow XB$
9: **else if** $\zeta = D$ **then** $x \leftarrow XAC$
10: **else if** $\zeta = -D$ **then** $x \leftarrow XA$
11: **return** x

```python
def algoritm_6(c, q, xi):
    """q = 17 (mod 32); s = 4"""
    b = powmod(c, (q - 17) // 32, q)
    X = (c * b) % q
    zeta = (X * b) % q
    A = xi
    B = (A ** 2) % q
    C = (B ** 2) % q
    D = (B * C) % q

    if zeta == 1:
        x = X
    elif zeta == q - 1:
        x = (X * C) % q
    elif zeta == B:
        x = (X * A * D) % q
    elif zeta == q - B:
        x = (X * A * B) % q
    elif zeta == C:
        x = (X * D) % q
    elif zeta == q - C:
        x = (X * B) % q
    elif zeta == D:
        x = (X * A * C) % q
    elif zeta == q - D:
        x = (X * A) % q
    else:
        x = 0
    return x
```

5 Сравнение алгоритмов

Далее приведем сравнение алгоритмов 1-6, а также наглядно покажем время работы каждого из них. Имеем следующие алгоритмы:

- Алгоритм 1 (Аткин) – для $q \equiv 5 \pmod 8$; 1 возведение в степень
- Алгоритм 2 (Мюллер) – для $q \equiv 9 \pmod{16}$; 2 возведения в степень
- Алгоритм 3 (Конг) – для $q \equiv 9 \pmod{16}$; в среднем 1,5 возведения в степень
- Алгоритм 4 – для $q \equiv 5 \pmod 8$; 1 возведение в степень
- Алгоритм 5 – для $q \equiv 9 \pmod{16}$; 1 возведение в степень
- Алгоритм 6 – для $q \equiv 17 \pmod{32}$; 1 возведение в степень

Отдельно сравним работу алгоритмов без функции powmod() и с ее использованием.

5.1 Сравнение без powmod()

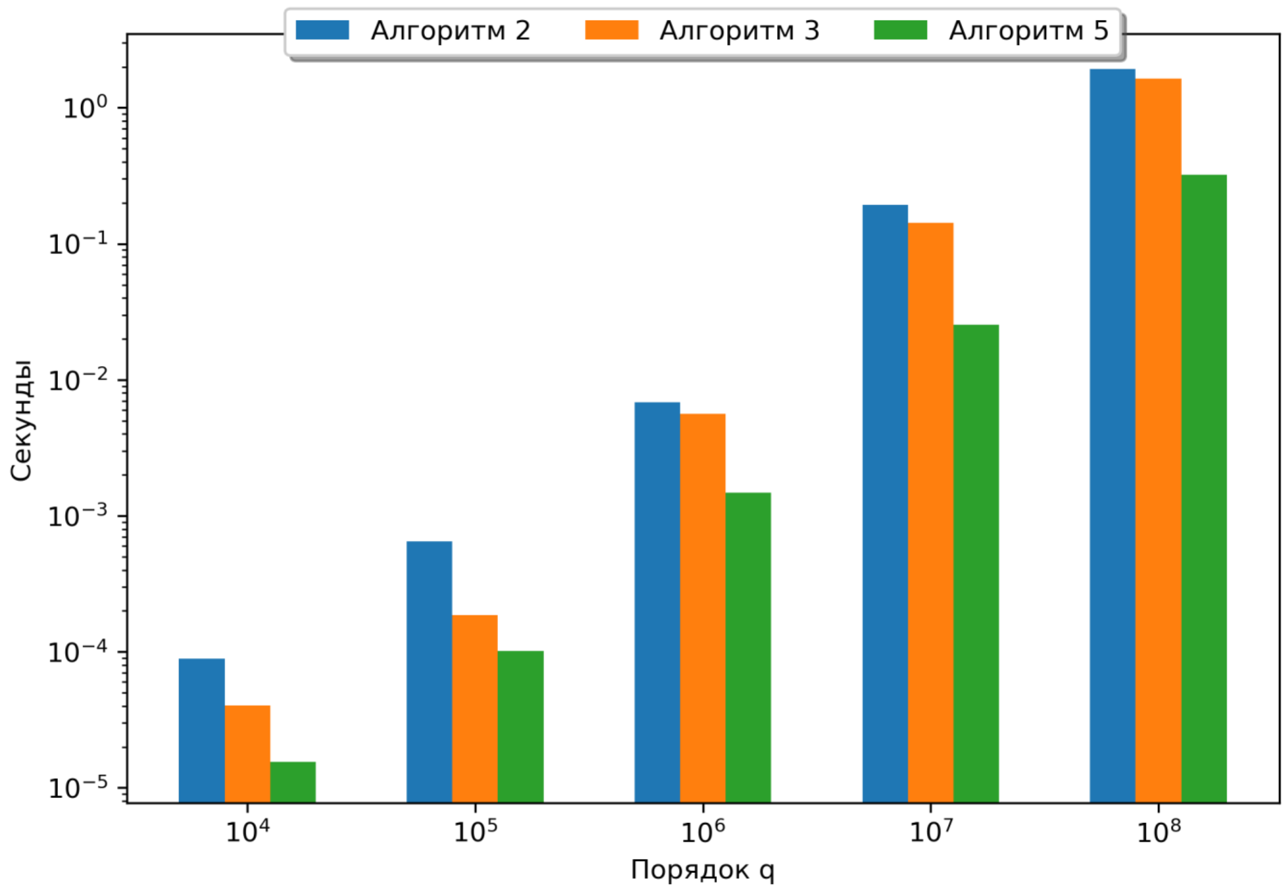
В этом случае выполняется сначала возведение в степень, а затем приведение по модулю, из-за чего значительно возрастает время работы алгоритмов.

Используем следующий метод расчета времени работы алгоритмов и их сравнения. Будем вычислять время работы алгоритмов для простых чисел q, которые ищем случайным образом на 5 промежутках: q порядка $10^4, 10^5, 10^6, 10^7, 10^8$. Выбираем q, удовлетворяющие условию $q \equiv 2^s + 1 \pmod{2^{s+1}}$. Будем брать по 5 чисел на каждый $s = 2, 3, 4$. Для каждого q подбираем квадратичный вычет c, квадратный корень которого будут находить алгоритмы. Также для каждого q заранее вычисляем значение ξ для алгоритмов 4, 5, 6. В итоге получаем набор q, c, ξ. Для одинаковых s берем одинаковые значения: алгоритмы 1 и 4 сравниваются на одних данных, аналогично алгоритмы 2, 3, 5.

5.1.1 Сравнение 2, 3 и 5 алгоритмов без powmod()

Проведем сравнение Алгоритма 5 с алгоритмами Мюллера и Конга. Так как они осуществляются для одного s, то возьмем для них одинаковые наборы q и s.

Получаем следующие результаты замеров (для времени используем логарифмическую шкалу):



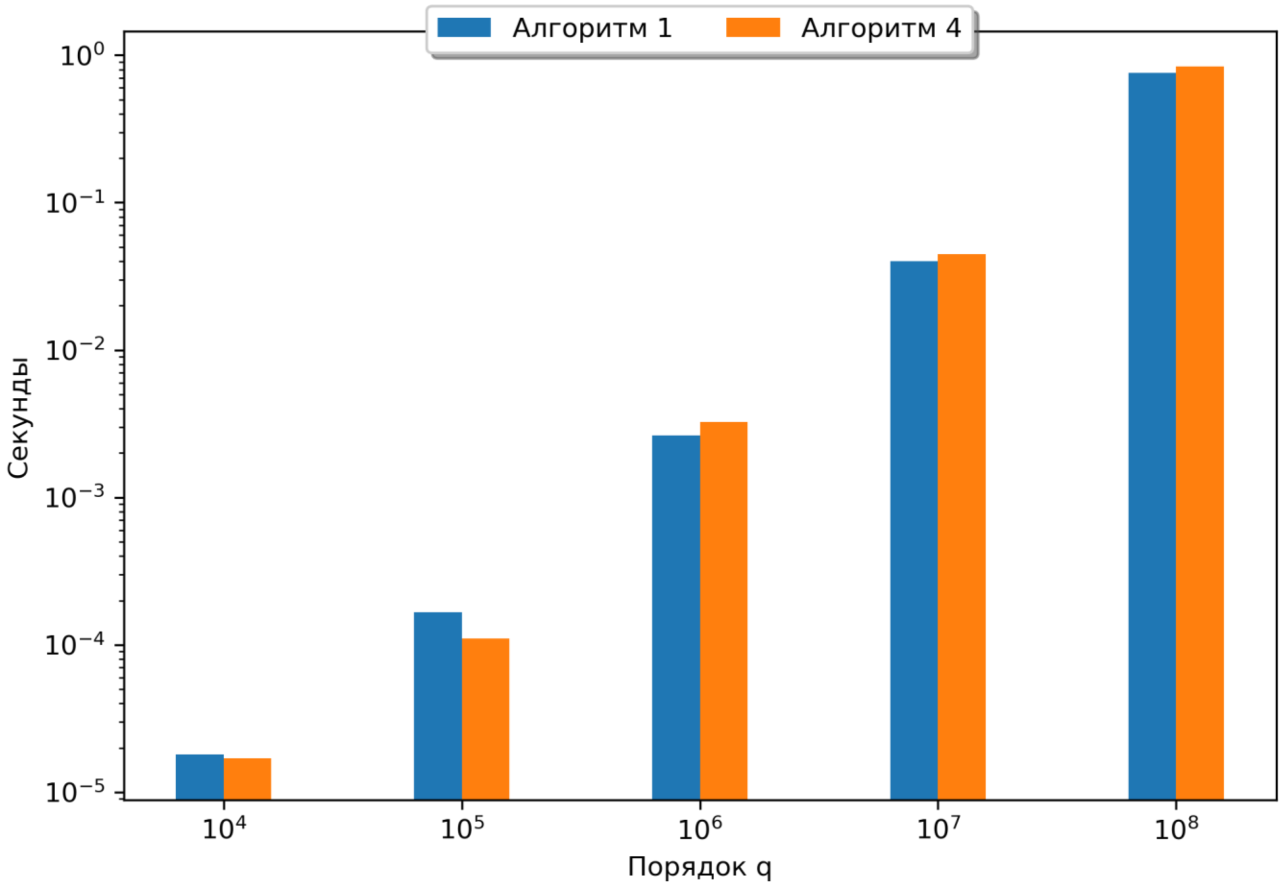
На графике видна разница в эффективности алгоритмов:

- Алгоритм 3 (Конг) эффективнее Алгоритма 2 (Мюллер) за счет привнесенного усовершенствования. В Алгоритме 2 производится 2 возведения в степень, а в Алгоритме 3 в среднем 1,5 возведения в степень
- Алгоритм 5 эффективнее Алгоритма 3 (Конг) за счет заранее вычисленного значения ξ. В Алгоритме 5 производится только 1 возведение в степень

5.1.2 Сравнение 1 и 4 алгоритмов без powmod()

Проведем сравнение Алгоритма 4 с алгоритмом Аткина. Так как они осуществляются для одного s, то возьмем для них одинаковые наборы q и s.

Получаем следующие результаты замеров (для времени используем логарифмическую шкалу):



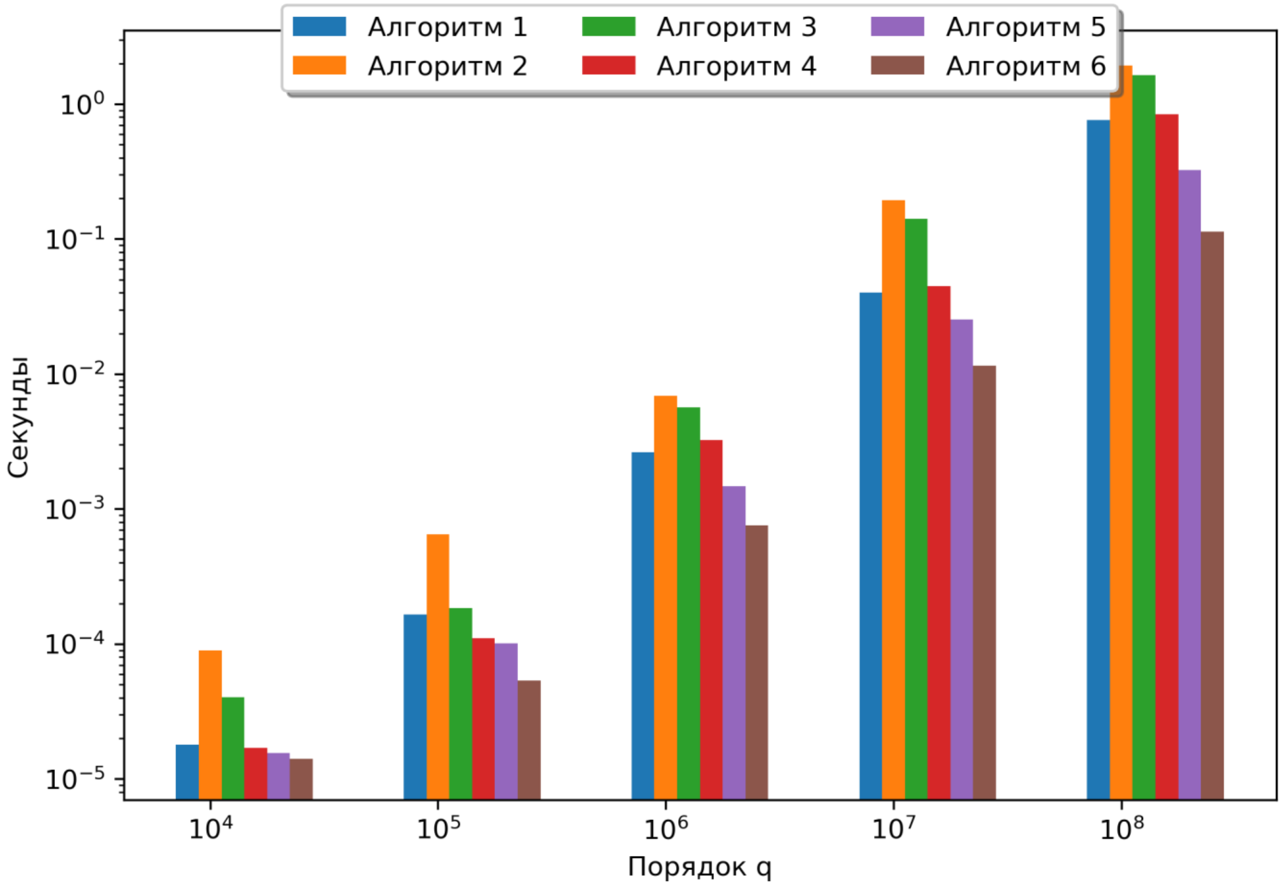
По полученным результатам можно сделать следующие выводы:

- Алгоритм 1 и Алгоритм 4 работают примерно за одинаковое время, так как они оба используют 1 возведение в степень $\frac{q-5}{8}$

5.1.3 Сравнение всех алгоритмов без powmod()

Проведем сравнение всех алгоритмов между собой. Покажем на одном графике все результаты подсчета времени работы алгоритмов.

Получаем следующие результаты замеров (для времени используем логарифмическую шкалу):



Итоговые выводы при сравнении алгоритмов:

- Алгоритм 1 и Алгоритм 4 работают примерно за одинаковое время, так как они оба используют 1 возведение в степень $\frac{q-5}{8}$
- Алгоритм 3 (Конг) работает быстрее Алгоритма 2 (Мюллер), так как в алгоритме Конга используется в среднем 1,5 возведения в степень (в отличие от 2 возведений у Мюллера)
- Алгоритм 5 работает быстрее Алгоритма 3 (Конг), так как в Алгоритме 5 используется только 1 возведение в степень (в отличие от в среднем 1,5 возведения у Конга)
- Алгоритм 6 работает быстрее Алгоритма 5, а Алгоритм 5 работает быстрее Алгоритма 4, хотя в каждом из них используется 1 возведение в степень. Это может объясняться величиной степени:
  - Для Алгоритма 4 степень $\frac{q-5}{8}$
  - Для Алгоритма 5 степень $\frac{q-9}{16}$

- Для Алгоритма 6 степень $\frac{q-17}{32}$

Получается, что в Алгоритме 6 производится возведение в самую маленькую степень, а в Алгоритме 4 в самую большую из представленных. Отсюда можно сделать вывод о том, что важно не только количество возведений в степень, но и величина этой степени.

5.2 Сравнение с powmod()

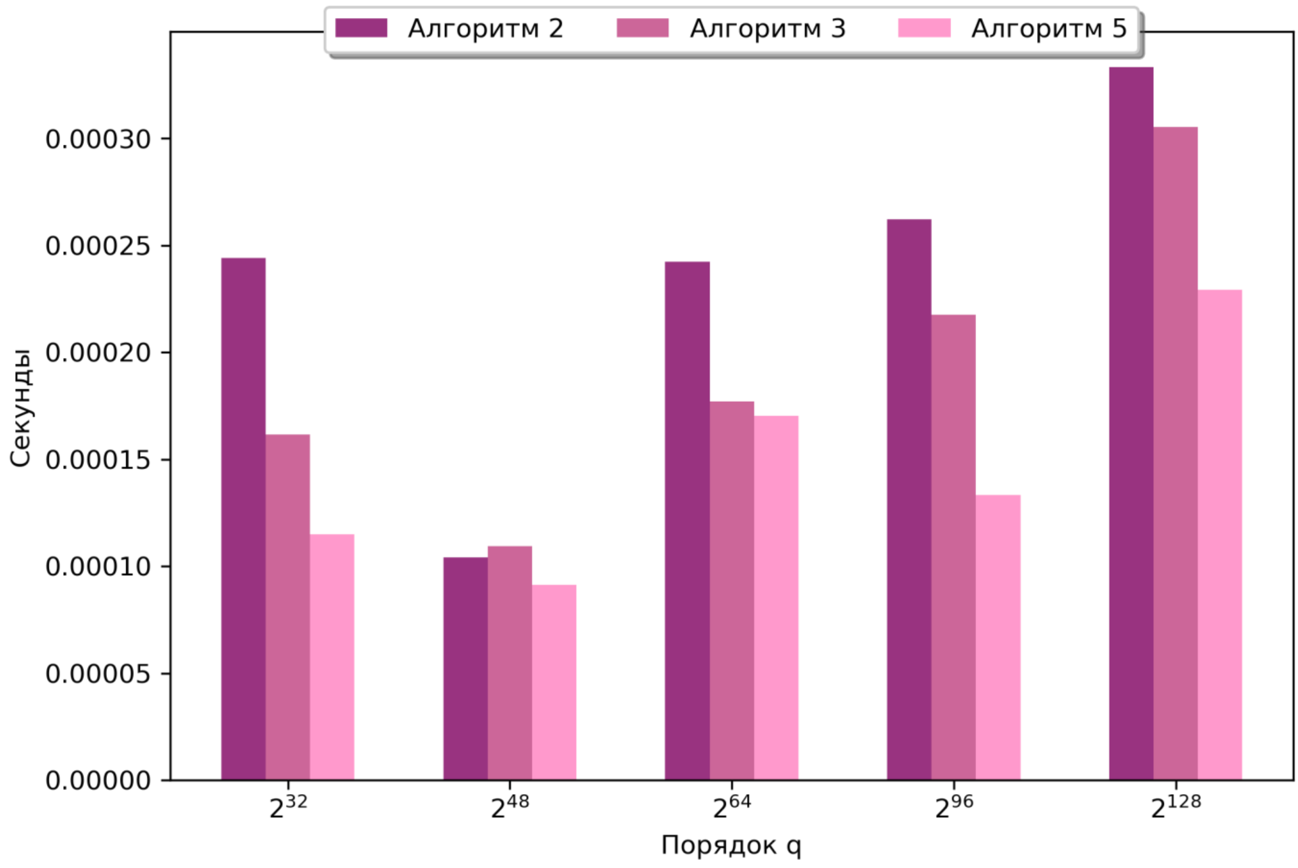
Функция powmod() значительно ускоряет возведение в степень по модулю за счет разбиения вычислений. Поэтому в этом случае можно будет рассматривать большие значения q, а также исследовать более объемные выборки значений.

Используем аналогичный предыдущему метод расчета времени работы алгоритмов и их сравнения. Будем вычислять время работы алгоритмов для простых чисел q, которые ищем случайным образом на 5 промежутках $[0, 2^{32})$, $[2^{48} - 2^{32}, 2^{48})$, $[2^{64} - 2^{32}, 2^{64})$, $[2^{96} - 2^{32}, 2^{96})$, $[2^{128} - 2^{32}, 2^{128})$. Выбираем q, удовлетворяющие условию $q \equiv 2^s + 1 \pmod{2^{s+1}}$. Также будем брать по 999 чисел на каждый $s = 2, 3, 4$.

5.2.1 Сравнение 2, 3 и 5 алгоритмов с powmod()

Проведем сравнение Алгоритма 5 с алгоритмами Мюллера и Конга. Так как они осуществляются для одного s, то возьмем для них одинаковые наборы q и s.

Получаем следующие результаты замеров:



На графике видна разница в эффективности алгоритмов:

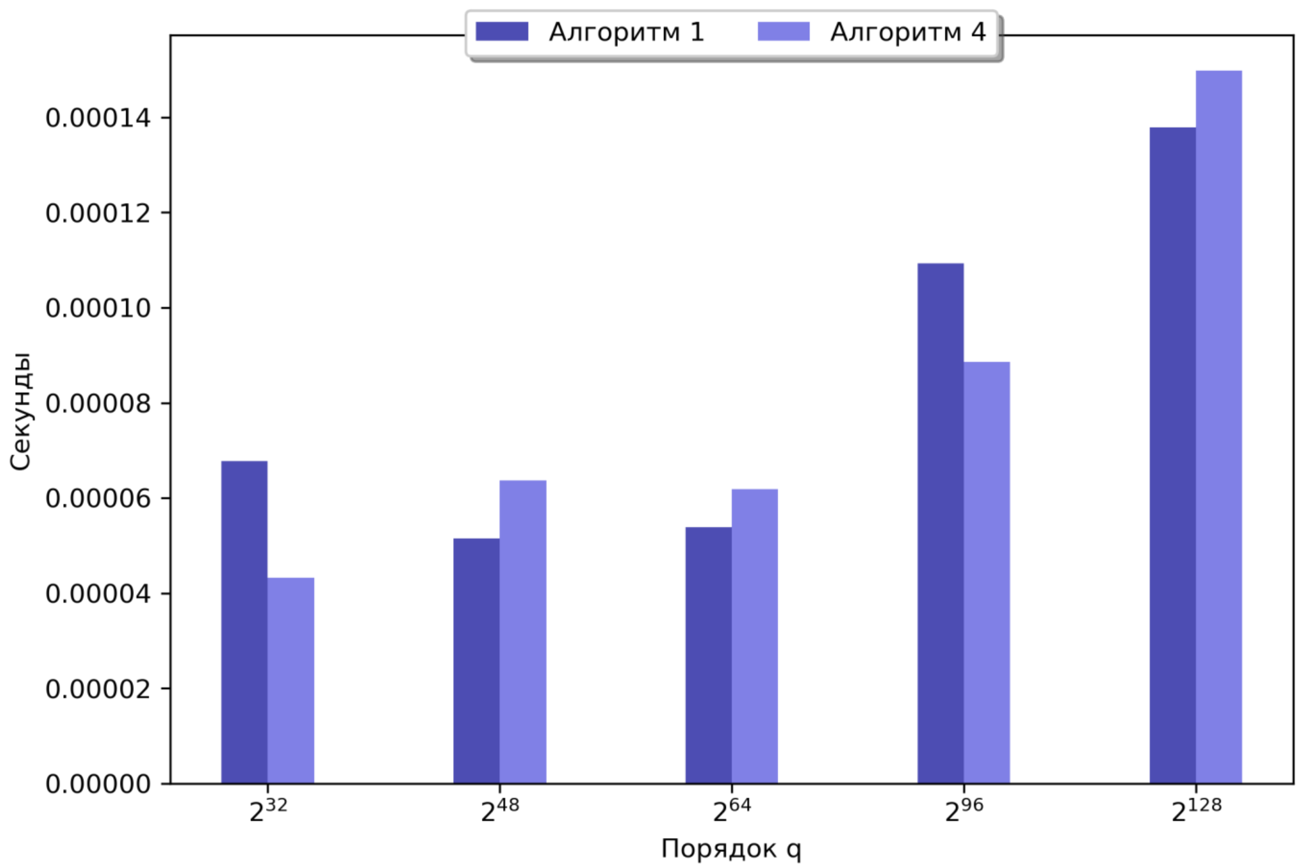
- Алгоритм 3 (Конг) эффективнее Алгоритма 2 (Мюллер) за счет привнесенного усовершенствования. В Алгоритме 2 производится 2 возведения в степень, а в Алгоритме 3 в среднем 1,5 возведения в степень
- Алгоритм 5 эффективнее Алгоритма 3 (Конг) за счет заранее вычисленного значения ξ. В Алгоритме 5 производится только 1 возведение в степень

Также можно заметить, что при всех рассматриваемых порядках q время работы алгоритмов меняется незначительно: изменения в пределах 0,002 сек. В то же время, при сравнении без powmod(), для разных порядков q время работы алгоритмов отличалось в разы.

5.2.2 Сравнение 1 и 4 алгоритмов с powmod()

Проведем сравнение Алгоритма 4 с алгоритмом Аткина. Так как они осуществляются для одного s, то возьмем для них одинаковые наборы q и s.

Получаем следующие результаты замеров:



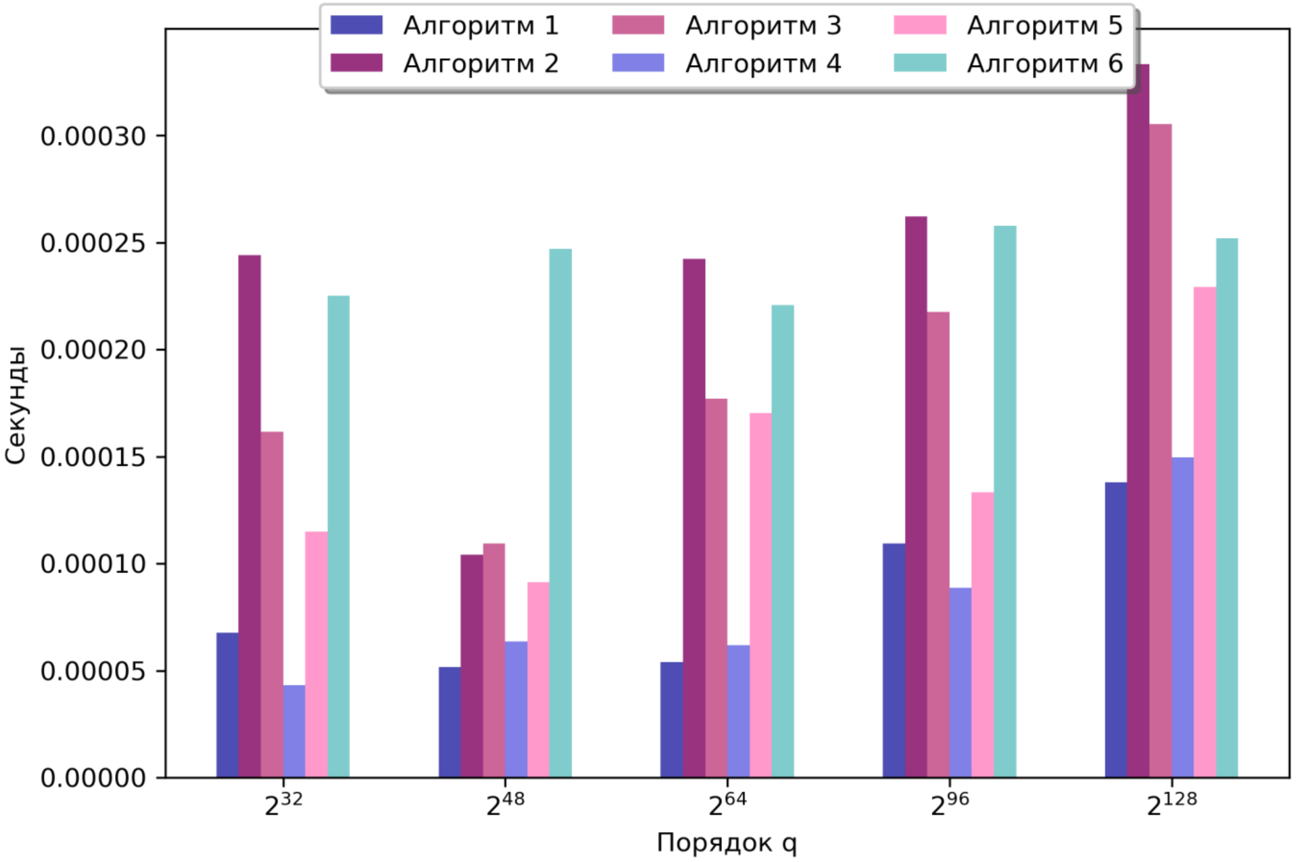
По полученным результатам можно сделать следующие выводы:

- Алгоритм 1 и Алгоритм 4 работают примерно за одинаковое время, так как они оба используют 1 возведение в степень $\frac{q-5}{8}$

5.2.3 Сравнение всех алгоритмов с powmod()

Проведем сравнение всех алгоритмов между собой. Покажем на одном графике все результаты подсчета времени работы алгоритмов.

Получаем следующие результаты замеров:



Итоговые выводы при сравнении алгоритмов:

- Алгоритм 1 и Алгоритм 4 работают примерно за одинаковое время, так как они оба используют 1 возведение в степень $\frac{q-5}{8}$
- Алгоритм 3 (Конг) работает быстрее Алгоритма 2 (Мюллер), так как в алгоритме Конга используется в среднем 1,5 возведения в степень (в отличие от 2 возведений у Мюллера)
- Алгоритм 5 работает быстрее Алгоритма 3 (Конг), так как в Алгоритме 5 используется только 1 возведение в степень (в отличие от в среднем 1,5 возведения у Конга)
- Алгоритм 6 работает медленнее Алгоритма 5, а Алгоритм 5 работает медленнее Алгоритма 4, хотя в каждом из них используется 1 возведение в степень. Так как во всех алгоритмах использовалась функция powmod() для ускорения возведения в степень по модулю, то разница во времени работы алгоритмов может быть связана с произведениями:
    - Для Алгоритма 4 в среднем 3,5 произведений
    - Для Алгоритма 5 в среднем 5 произведений
    - Для Алгоритма 6 в среднем 6,25 произведений

Получается, что в Алгоритме 6 производится больше всех произведений, а в Алгоритме 4 меньше всех. Отсюда можно сделать вывод о том, что важно не только количество возведений в степень, но и количество произведений (в случае, когда возведение в степень оптимизировано).

5.3 Выводы о влиянии powmod()

Одинаковые результаты с использованием и без powmod():

- Алгоритм 2 медленнее Алгоритма 3, а Алгоритм 3 медленнее Алгоритма 5
- Алгоритм 1 и Алгоритм 4 работают примерно за одинаковое время

Различия вызванные powmod():

- Без использования powmod() алгоритмы работают существенно медленнее: без powmod() время работы порядка 1 сек уже при $q \sim 10^8$, а с powmod() для любых $q < 2^{128}$ и $s \leq 4$ время работы $< 0{,}004$ сек
- Без powmod() Алгоритм 6 быстрее Алгоритма 5, а Алгоритм 5 быстрее Алгоритма 4, в то время как с powmod() все наоборот

Так как powmod() позволяет ускорить вычисления, то предпочтительно его использовать при реализации алгоритмов. Поэтому в данной работе во всех реализациях алгоритмов используется powmod(). Однако стоит учитывать, что уменьшение значения степени при возрастании s в этом случае не уменьшает время работы, а наоборот увеличивает его (за счет увеличения количества произведений).

6 Тесты и приложения

6.1 Функция для вычисления ξ

Функция принимает на вход заданные q и s и вычисляет значение ξ для этих параметров:

```
def xi_value(q, s):
    d = 0
    for j_d in range(2, q):
        if kronecker(j_d, q) == -1:
            d = j_d
            break
    if d == 0:
        sys.exit("Can not find d")
    xi = powmod(d, (q - 1) // (2 ** s), q)
    return xi
```

6.2 Тесты

Проверим работу алгоритмов 1-3 с помощью следующей функции, которая принимает значения s, q, c, а также проверяемый алгоритм:

```
def test_123(s, q, c, alg):
    mod_s = 2 ** (s + 1)
    print(q, '=', q % mod_s, 'mod', mod_s, '; Символ Лежандра (c/q) =', kronecker(c,
q))
    x = alg(c, q)
    print('x =', x, '; x^2 =', x ** 2 % q, '; c =', c)
    print('Проверка равенства:', x ** 2 % q == c)
```

Эта функция выполняет проверки и выводит следующую информацию:

- Соответствует ли q условию $q \equiv 2^s + 1 \pmod{2^{s+1}}$ для заданного s
- Является ли c квадратичным вычетом в $\mathbb{F}_q$: для этого вычисляется значение символа Лежандра (если оно равно 1, то c – вычет)
- Выводит найденный с помощью алгоритма квадратный корень x
- Возводит x в квадрат и сравнивает результат с исходным c

Алгоритм 1 (Аткин)

Проверка работы алгоритма для $q \equiv 5 \pmod 8$:

- 1 тест

```
test_123(2, 10141, 1111, algoritm_1)
```

```
10141 = 5 mod 8 ; Символ Лежандра (c/q) = 1
x = 1895 ; x^2 = 1111 ; c = 1111
Проверка равенства: True
```

- 2 тест

```
test_123(2, 1001093, 7707, algoritm_1)
```

```
1001093 = 5 mod 8 ; Символ Лежандра (c/q) = 1
x = 147179 ; x^2 = 7707 ; c = 7707
Проверка равенства: True
```

- 3 тест (некорректные входные значения - невычет)

```
test_123(2, 305101, 666, algoritm_1)
```

```
305101 = 5 mod 8 ; Символ Лежандра (c/q) = -1
x = 0 ; x^2 = 0 ; c = 666
Проверка равенства: False
```

Алгоритм 2 (Мюллер)

Проверка работы алгоритма для $q \equiv 9 \pmod{16}$

- 1 тест

```
test_123(3, 11801, 23, algoritm_2)
```

```
11801 = 9 mod 16 ; Символ Лежандра (c/q) = 1
x = 2221 ; x^2 = 23 ; c = 23
Проверка равенства: True
```

- 2 тест

```
test_123(3, 1009433, 234567, algoritm_2)
```

```
1009433 = 9 mod 16 ; Символ Лежандра (c/q) = 1
x = 747634 ; x^2 = 234567 ; c = 234567
Проверка равенства: True
```

- 3 тест (некорректные входные значения - невычет)

```
test_123(3, 300953, 666, algoritm_2)
```

```
300953 = 9 mod 16 ; Символ Лежандра (c/q) = -1
An exception has occurred, use %tb to see the full traceback.
SystemExit: Can not find d
```

Алгоритм 3 (Конг)

Проверка работы алгоритма для $q \equiv 9 \pmod{16}$

- 1 тест

```
test_123(3, 11801, 23, algoritm_3)
```

```
11801 = 9 mod 16 ; Символ Лежандра (c/q) = 1

x = 2221 ; x^2 = 23 ; c = 23

Проверка равенства: True
```

- 2 тест

```
test_123(3, 1009433, 234567, algoritm_3)
```

```
1009433 = 9 mod 16 ; Символ Лежандра (c/q) = 1

x = 747634 ; x^2 = 234567 ; c = 234567

Проверка равенства: True
```

- 3 тест (некорректные входные значения – q не простое)

```
test_123(3, 1625, 98, algoritm_3)
```

```
1625 = 9 mod 16 ; Символ Лежандра (c/q) = 1

x = 438 ; x^2 = 94 ; c = 98

Проверка равенства: False
```

Проверим работу алгоритмов 4-6 с помощью следующей функции, которая принимает значения s, q, c, а также проверяемый алгоритм. В процессе работы эта функция вычисляет значение ξ для данного q

```
def test_456(s, q, c, alg):
    mod_s = 2 ** (s + 1)
    print(q, '=', q % mod_s, 'mod', mod_s, '; Символ Лежандра (c/q) =', kronecker(c,
q))
    xi = xi_value(q, s)
    x = alg(c, q, xi)
    print('x =', x, '; x^2 =', x ** 2 % q, '; c =', c)
    print('Проверка равенства:', x ** 2 % q == c)
```

Эта функция выполняет проверки и выводит следующую информацию:

- Соответствует ли q условию $q \equiv 2^s + 1 \pmod{2^{s+1}}$ для заданного s
- Является ли c квадратичным вычетом в $\mathbb{F}_q$: для этого вычисляется значение символа Лежандра (если оно равно 1, то c – вычет)
- Выводит найденный с помощью алгоритма квадратный корень x
- Возводит x в квадрат и сравнивает результат с исходным c

Алгоритм 4

Проверка работы алгоритма для $q \equiv 5 \pmod 8$

- 1 тест

```
test_456(2, 50461, 111, algoritm_4)
```

```
50461 = 5 mod 8 ; Символ Лежандра (c/q) = 1

x = 31367 ; x^2 = 111 ; c = 111

Проверка равенства: True
```

- 2 тест (некорректные входные значения – невычет)

```
test_456(2, 517613, 500000, algoritm_4)
```

```
517613 = 5 mod 8 ; Символ Лежандра (c/q) = -1

x = 385506 ; x^2 = 419541 ; c = 500000

Проверка равенства: False
```

Алгоритм 5

Проверка работы алгоритма для $q \equiv 9 \pmod{16}$

- 1 тест

```
test_456(3, 544793, 404, algoritm_5)
```

```
544793 = 9 mod 16 ; Символ Лежандра (c/q) = 1

x = 418943 ; x^2 = 404 ; c = 404

Проверка равенства: True
```

- 2 тест (некорректные значения – q не простое)

```
test_456(3, 160025, 17, algoritm_5)
```

```
160025 = 9 mod 16 ; Символ Лежандра (c/q) = 1

x = 8778 ; x^2 = 81259 ; c = 17

Проверка равенства: False
```

Алгоритм 6

Проверка работы алгоритма для $q \equiv 17 \pmod{32}$

- 1 тест

```
test_456(4, 50126833, 111111, algoritm_6)
```

```
50126833 = 17 mod 32 ; Символ Лежандра (c/q) = 1
```

```
x = 1978118 ; x^2 = 111111 ; c = 111111

Проверка равенства: True
```

- 2 тест (некорректные значения – невычет)

```
test_456(4, 700139537, 111111, algoritm_6)
```

```
700139537 = 17 mod 32 ; Символ Лежандра (c/q) = -1

x = 0 ; x^2 = 0 ; c = 111111

Проверка равенства: False
```

Список литературы

[1] N. Koo, G. H. Cho, and S. Kwon, "Square root algorithm in $\mathbb{F}_q$ for $q \equiv 2^s + 1 \pmod{2^{s+1}}$", Electronics Letters, vol. 49, no. 7, pp. 467-469, March 28, 2013.

[2] А. Ю. Нестеренко. Теоретико-числовые методы в криптографии. М. : Московский государственный институт электроники и математики, 2012.