

ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

УДК: 192.168.1.1

Отчет об исследовательском проекте

на тему: Большие простые числа

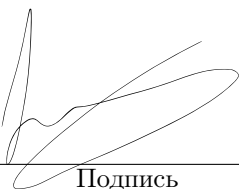
(промежуточный, этап 1)

Выполнил:

Студент группы БПМИ215

30.04.2024

Дата



Подпись

Т.К. Андриян

И.О.Фамилия

Принял:

Руководитель проекта

Дмитрий Витальевич Трушин

Имя, Отчество, Фамилия

доцент, к.ф.-м.н.

Должность, ученое звание

ФКН НИУ ВШЭ

Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 2024

Оценка (по 10-ти бальной шкале)

Подпись

Москва 2024

Содержание

1	Введение	3
1.1	Описание предметной области	3
1.2	Задачи и цели	3
2	Описание функциональных и нефункциональных требований к программному проекту	3
2.1	Функциональные требования	3
2.2	Язык программирования, сборка и тестирование	3
3	Основные математические выкладки	3
3.1	Базовые утверждения и теоремы	3
4	Алгоритмы факторизации	5
4.1	Базовый алгоритм факторизации	5
4.1.1	Введение	5
4.1.2	Замеры времени работы	5
4.2	Pollard Rho	5
4.2.1	Введение	5
4.3	Оптимизации	6
4.3.1	Подбор гиперпараметров и замеры времени работы	6
4.3.2	Преимущества и недостатки	6
4.4	Quadratic Sieve	6
4.4.1	Введение	6
4.4.2	Подбор последовательности простых чисел	7
4.4.3	Подбор последовательности целевых переменных	7
4.4.4	Нахождение нужной подпоследовательности	7
4.4.5	Оптимизации скорости работы	7
4.4.6	Подбор гиперпараметров и замеры времени работы	7
4.4.7	Оптимизации расходов памяти	8
4.4.8	Преимущества и недостатки	9
4.5	Contuned Fractions	9
4.5.1	Введение	9
4.5.2	Нахождение нужной подпоследовательности	9
4.5.3	Подбор гиперпараметров и замеры времени работы	9
4.5.4	Преимущества и недостатки	10
4.6	Elliptic Curve	10
4.6.1	Введение	10
4.6.2	Факторизация	10
4.6.3	Замеры времени работы	10
4.6.4	Преимущества и недостатки	10
5	Алгоритмы проверки на простоту	11
5.1	Pseudo Prime Test	11
5.1.1	Введение	11
5.1.2	Устойчивость	11
5.2	Strong Pseudo Prime Test	11
5.2.1	Введение	11
5.2.2	Устойчивость	11
6	Библиотека LPN	11
6.1	Введение	11
6.2	Сборка	12
6.3	Тестирование	12

Аннотация

В курсовой работе рассматриваются и описываются алгоритмы, предоставляющие возможность вычисления больших простых чисел. Выявлены ключевые подходы, а также проведена оценка скорости различных алгоритмов и их композиций.

1 Введение

1.1 Описание предметной области

В теории чисел отдельное внимание уделено простым числам и их свойствам. Простые числа играют ключевую роль в области криптографии, особенно в асимметричных криптографических системах с открытым ключом, таких как [5, RSA]. Одним из основополагающих принципов таких систем является вычислительная сложность факторизации больших составных чисел, которые в основном представляют собой произведение двух длинных простых чисел.

1.2 Задачи и цели

Целью выполнения курсовой работы является изучение основных алгоритмов, а также их реализация с последующим тестированием и сравнительным анализом эффективности. Кроме того, в рамках исследования предусматривается разработка эвристик для ускорения алгоритмов.

2 Описание функциональных и нефункциональных требований к программному проекту

2.1 Функциональные требования

Библиотека LPN¹ должна содержать в себе несколько базовых подходов для факторизации чисел, таких как: Basic Factorization [4, Chapter 2], Pollard Rho и Pollard $p - 1$ [4, Chapter 5], а также продвинутые подходы: Quadratic Sieve [4, Chapter 8], CFRAC [4, Chapter 11], и Elliptic Curve Method [4, Chapter 13]. Также в библиотеке должны быть реализованы вероятностные алгоритмы, позволяющие достаточно быстро проверять число на простоту, к примеру, Pseudo Prime Test [6].

2.2 Язык программирования, сборка и тестирование

Выбор основного языка программирования в виде C++ обусловлен его высокой производительностью, что критически важно для эффективной реализации задачи проверки чисел на простоту. Использование утилиты CMake для сборки программы является хорошей практикой, обеспечивая удобное и автоматизированное управление процессом компиляции и сборки проекта. Тестирование реализовано на основе GTest, так как его легко использовать вместе с утилитой CMake, более того при помощи CI-контура будет производиться проверка кода на корректность работы при отправке в удаленный GIT репозиторий [3].

3 Основные математические выкладки

Для понимания вышеупомянутых алгоритмов необходимо привести несколько теорем, формулировок и основных математических концепций.

3.1 Базовые утверждения и теоремы

Определение 1. $a \equiv b \pmod{m}$ означает, что $a - b$ делится на m .

Определение 2. $\gcd(a, b)$ - наибольший общий делитель.

Алгоритм 3 (Алгоритм Евклида). *Предположим, что $a > b > 0$, тогда $\gcd(a, b) = \gcd(a - b, b)$.*

Теорема 4. *Если p - простое нечетное число, то $2^{p-1} \equiv 1 \pmod{p}$.*

¹LPN - разрабатываемая библиотека, анаграмма от Large Prime Numbers

Определение 5. Если n - нечетное составное число и $2^{n-1} \equiv 1 \pmod n$, то n называется псевдопростым.

Теорема 6. Если p - простое число не делящее b , то $b^{p-1} \equiv 1 \pmod p$.

Определение 7. Если n - нечетное составное число, $\gcd(n, b) = 1$ и $b^{n-1} \equiv 1 \pmod b$, тогда n является псевдопростым по модулю b .

Определение 8. Составное число a является числом Кармайкла, если для любого простого числа, не делящего a , выполняется теорема [6]. 561 - первое число Кармайкла.

Определение 9. Число n называется квадратичным остатком по модулю p , если $n \equiv r^2 \pmod p$.

Определение 10. Пусть p нечетное простое число и n натуральное, тогда Legendre symbol определяется:

$$\left(\frac{n}{p}\right) = \begin{cases} 0, & \text{если } p \text{ делит } n \\ 1, & \text{если } n \text{ является квадратичным остатком по модулю } p \\ -1, & \text{иначе} \end{cases}$$

Алгоритм 11 (Bhaskara-Brouncker). На вход подается число n , алгоритм выдает пару чисел, отношение которых является \sqrt{n} с определенной точностью.

```

1  std::tuple<int, int> Bhaskara-Brouncker(const long_int & n) {
2      long_int sqrt = std::sqrt(n);
3      auto a = sqrt;
4      long_int b0 = 0, b1 = sqrt;
5      long_int c0 = 1, c1 = n - sqrt * sqrt;
6      long_int p0 = 1, p1 = sqrt;
7      long_int q0 = 0, q1 = 1;
8      size_t i = 1;
9
10     while (c1 != 1) {
11         a = (sqrt_n + b1) / c1;
12         b0 = std::exchange<long_int, long_int>(b1, a * c1 - b1);
13         c0 = std::exchange<long_int, long_int>(c1, c0 + a * (b0 - b1));
14         p0 = std::exchange<long_int, long_int>(p1, (p0 + a * p1) % n);
15         q0 = std::exchange<long_int, long_int>(q1, (q0 + a * q1) % n);
16     }
17     return {p1, q1};
18 }

```

Определение 12. Эллиптическая кривая над полем \mathbb{F}_p задается уравнением $y^2 = x^3 + a \times x + b \pmod p$, где a и b параметры, а p - простое число.

Лемма 13. Пусть даны 2 точки на эллиптической кривой с координатами (x_1, y_1) и (x_2, y_2) соответственно. Пусть также уравнение кривой задается как $y^2 = x^3 + a \times x + b \pmod p$ и $4a^3 + 27b^2 \neq 0$. Тогда

$$\text{если } x_1 \neq x_2, \text{ то } \lambda = \frac{y_1 - y_2}{x_1 - x_2} \quad (1)$$

$$\text{иначе } \lambda = \frac{3x_1^2 + a}{2y_1} \quad (2)$$

И в итоге:

$$x_3 = \lambda^2 - x_1 - x_2 \quad (3)$$

$$y_3 = \lambda \times (x_3 - x_1) + y_1 \quad (4)$$

Определение 14. Пусть дана эллиптическая кривая, а также две рациональные точки на кривой (x_1, y_1) и (x_2, y_2) . Введем бинарную операцию:

$$(x_1, y_1) \partial (x_2, y_2) = (x_3, -y_3) \quad (5)$$

Определение 15. Определим ∞ для бинарной операции ∂ :

$$(x, y) \partial (x, -y) = (x, -y) \partial (x, y) = \infty \quad (6)$$

Определение 16. Пусть дана эллиптическая кривая $y^2 = x^3 + ax + b$, $4a^2 + 27b^2 \neq 0$. Определим $E(a, b)$ группу со всеми рациональными точками на кривой и ∞ , а в качестве бинарной операции возьмем ∂ .

Определение 17. Если $x_1 \equiv x_2 \pmod n$ и $y_1 \equiv -y_2 \pmod n$, тогда определим ∞ для бинарной операции ∂ по модулю n :

$$(x_1, y_1)\partial(x_2, y_2) = \infty \quad (7)$$

Определение 18. Определим бинарную операцию по модулю:

$$\begin{aligned} &\text{если } x_1 \neq x_2 \pmod n, \gcd(x_1 - x_2, n) = 1 \text{ и } s = (x_1 - x_2)^{-1} \pmod n, \text{ то} \\ &\quad \lambda = (y_1 - y_2) \times s \pmod n \\ &\text{если } x_1 = x_2 \pmod n, \gcd(y_1 + y_2, n) = 1 \Rightarrow y_1 \equiv y_2 \pmod n \text{ и } s = (2y_1)^{-1} \pmod n, \text{ то} \\ &\quad \lambda = (3 \times x_1^2 + a) \times s \pmod n \end{aligned}$$

Тогда бинарная операция ∂ определяется как:

$$(x_1, y_1)\partial(x_2, y_2) \equiv (x_3, -y_3) \pmod n \quad (8)$$

Определение 19. Введем степень пары по модулю нечетного простого числа в группе $E(a, b)/p$ как:

$$(x_i, y_i) \equiv (x_1, y_1)\#i \pmod p \quad (9)$$

4 Алгоритмы факторизации

4.1 Базовый алгоритм факторизации

4.1.1 Введение

Известно, что для нахождения всех делителей числа N достаточно просмотреть все делители от 2 до \sqrt{N} . Данный алгоритм очень прост в написании, но занимает слишком большое количество времени для чисел больше чем 20 знаков.

4.1.2 Замеры времени работы

Number of Digits	Run Time
10	0.5 ms
15	100 ms
20	40000 ms

4.2 Pollard Rho

4.2.1 Введение

Предположим, что n — составное число, а d - нетривиальный делитель. Возьмем некоторую неразложимую в вещественных числах функцию $f(x)$, на практике обычно используется $f(x) = x^2 + 1$. Возьмем некоторую стартовую точку x_0 и начнем считать $x_i = f(x_{i-1})$, $y_i = x_i \pmod d$. Тогда если $y_i \equiv y_j \pmod d \iff x_i \equiv x_j \pmod d$. Из этого следует, что $x_i - x_j \vdots d$. Тогда велик шанс, что $\gcd(x_i - x_j, n) \neq 1$. Проблема заключается в том, что не известен изначальный делитель d . Поэтому предлагается иной подход:

$$x_1 - x_3$$

$$x_3 - x_6$$

$$x_3 - x_7$$

$$x_7 - x_{12}$$

...

$$x_7 - x_{15}$$

Общая формула: $x_{2^n-1} - x_j$, где $2^{n+1} - 2^{n-1} \leq j \leq 2^{n+1} + 1$. Главная идея такого подсчета — на каждой итерации разница координат увеличивается ровно на 1, что увеличивает шансы на нахождение цикла ($x_i \equiv x_j \pmod n$).

Итоговая идея: считаем каждый раз $\gcd(x_i - x_j, n)$.

4.3 Оптимизации

Поскольку операция $\gcd(x_i - x_j, n)$ является одной из наиболее ресурсоёмких в цикле, предлагается ввести гиперпараметр *frequency*, определяющий, с какой периодичностью следует выполнять вычисление.

4.3.1 Подбор гиперпараметров и замеры времени работы

Тестирование проводилось на числах вида $p_1 \times p_2$, где p_1 и p_2 - простые числа. Number of Attempts - параметр, который указывает сколько различных x_0 пришлось перебрать, перед тем как удалось разложить число на простые. Ниже приведены оптимальные значения параметров, а также среднее время выполнения программы:

Number of Digits	frequency	Number of Attempts	Run Time
15	10	1	2 ms
20	10	1	85 ms
20	20	1	80 ms
20	100	1	80 ms
20	500	2	80 ms
25	10	1	365 ms
25	20	1	360 ms
25	100	2	340 ms
25	500	4	340 ms

4.3.2 Преимущества и недостатки

Pollard Rho справляется с задачей лучше базовой факторизации, также алгоритм прост в написании. Однако его скорость работы и эффективность недостаточно высоки для практического применения.

4.4 Quadratic Sieve

4.4.1 Введение

Предположим, что мы умеем быстро находить два числа x и y , удовлетворяющие условию $x^2 \equiv y^2 \pmod n$. Тогда существует относительно большая вероятность того, что $\gcd(x - y, n)$ не тривиальный делитель n .

Введем функцию $g(r) = r \times r \pmod n$, где r является случайной величиной из равномерного распределения. Попробуем разложить $g(r)$ на простые множители, но перед этим предлагается зафиксировать набор простых чисел p_1, \dots, p_k , так как разложение числа на простые в общем случае является сложной задачей.

$$g(r) = p_1^{a_1} \times p_2^{a_2} \times \dots \times p_k^{a_k} \quad (10)$$

Тогда если все степени делителей четные, то число r подходит, иначе ищем другое. Однако вероятность такого события слишком мала, придется перебрать огромное количество чисел. Поэтому ведем вектор:

$$v(r) = (b_1, \dots, b_k), \text{ где } b_i = a_i \pmod 2 \quad (11)$$

Предположим набрался набор r_1, \dots, r_m такой, что:

$$g(r_1) \times \dots \times g(r_m) \equiv r_1^2 \times \dots \times r_m^2 \pmod n \quad (12)$$

Обе части уравнения являются квадратами, то есть нашлись два таких числа, что $x^2 \equiv y^2 \pmod n$. Также заметим, что $\forall i \in [1, k] \sum_{t=1}^m b_{it} \equiv 0 \pmod 2$ эквивалентна [12]. Осталось научиться эффективно подбирать набор r_1, \dots, r_m , далее называем такой набор целевым.

4.4.2 Подбор последовательности простых чисел

Рассмотрим функцию:

$$f(r) = r \times r - n \quad (13)$$

Заметим, что $f(r) = g(r)$ при $r \in [\sqrt{n}, \sqrt{2n}]$. Тогда если p делит $f(r)$, то $n \equiv r^2 \pmod{p}$. Из этого следует, что достаточно рассматривать простые числа, для которых n является квадратичным по модулю. Назовем такой набор простых чисел *primes*, а его размер `factor_base`.

4.4.3 Подбор последовательности целых переменных

Предположим, что мы умеем решать уравнения вида $x^2 \equiv n \pmod{p}$, где $p \in \text{primes}$. Введем новую переменную $r = \lfloor \sqrt{n} \rfloor$. Вместо того чтобы брать r_i как случайную величину, предлагается рассмотреть отрезок размера $2m$, где m - гиперпараметр. Тогда $r_i = r + i$, где $i \in [-m, m]$. Если выбранное m будет относительно небольшое, то и значения $f(r_i)$ также будут относительно близкими к нулю, что повышает вероятность полного разложения $f(r_i)$ через простые из *primes*.

4.4.4 Нахождение нужной подпоследовательности

В конечном итоге, нужно найти подмножество из r_i для которых выполняется равенство[12]. Для успешных разложений $f(r_i)$ построим вектор[11]. Запишем все такие векторы в матрицу $G \in \mathbb{Z}_2^{m \times n}$, где m - количество r_i , которых удалось разложить на простые через *primes*, а n - размер вектора *primes*. При помощи Гауссовский преобразований приведем матрицу к ступенчатому виду. Тогда нулевые строки будут являться решением равенства[12].

4.4.5 Оптимизации скорости работы

Решето

При создании матрицы G из предыдущего шага необходимо разложить $2m$ чисел на простые множители из множества *primes*. Это приведет к выполнению $\mathcal{O}(m \times |\text{primes}|)$ операций **деления**, что потребует значительного времени. Вместо этого заметим:

$$f(r_i) \equiv 0 \pmod{p} \Leftrightarrow f(r_i + p) \equiv 0 \pmod{p} \quad (14)$$

Введём массив *Sieve* размером $2m$, заполненный нулями. Если для индекса i выполняется условие $f(r_i) \equiv 0 \pmod{p}$, то в соответствующей ячейке с индексом $i + m$ добавляется $\log p$. Для каждого $p_k \in \text{primes}$ находим наименьший индекс $i_{k_0} \in [-m, m]$, удовлетворяющий условию $f(r_{i_{k_0}}) \equiv 0 \pmod{p_k}$. Добавляем $\log(p)$ в каждую ячейку с индексом $i_{k_j} = i_{k_0} + p_k \times j$. В итоге нам понадобится $\mathcal{O}(m \times \log |\text{primes}|)$ операций **сложения**. Осталось по данным из вектора *Sieve* найти числа, которые могут быть факторизованы через *primes*. Для этого введем:

$$\text{threshold} = \frac{\log n}{2} + \log m - t \times \log \max(\text{primes}), \text{ где} \quad (15)$$

t - гиперпараметр

Тогда если значение в ячейке *Sieve* превышает *threshold*, то осуществляется попытка разложения $f(r_i)$ через *primes*. В случае успеха строится вектор[11] и добавляется в матрицу G .

Многопоточное решето

Основной задачей решета является добавление $\log p_i$ в ячейки *Sieve*. Предлагается разделить массив *Sieve* на непересекающиеся отрезки, после чего в отдельных потоках исполнить алгоритм *Sieve* для указанной части массива.

4.4.6 Подбор гиперпараметров и замеры времени работы

После внедрения оптимизаций, связанных с *Sieve*, появились гиперпараметры, которые оказывают влияние на время работы программы, а также на её успешность выполнения. Тестирование проводилось на числах вида $p_1 \times p_2$, где p_1 и p_2 - простые числа. Ниже приведены оптимальные значения параметров, а также среднее время выполнения программы ($M=1000000$):

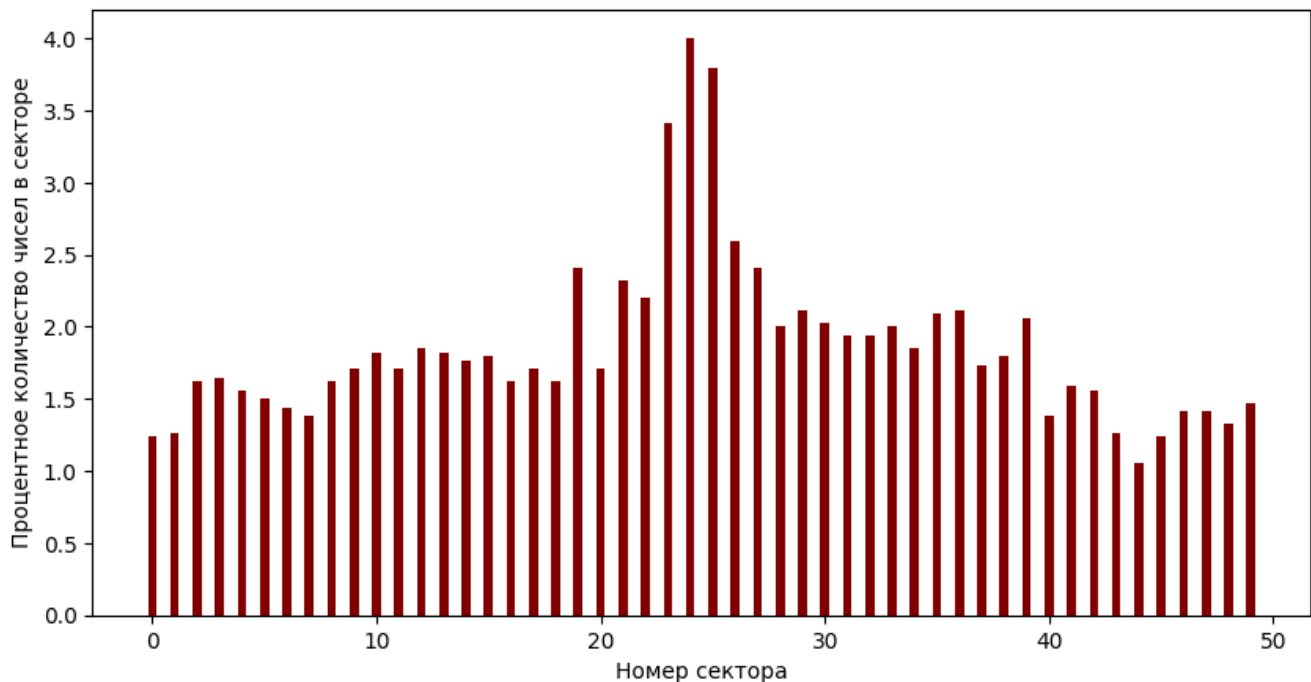
Number of Digits	Factor Base	m	t	Run Time(1 Thread)	Run Time(8 Thread)	Memory Usage
24	300	0.5M	0.4	5 ms	7 ms	7.5 MB
30	900	2M	0.7	45 ms	38 ms	30 MB
36	1900	13.5M	0.8	250 ms	160 ms	210 MB
40	3300	33M	0.8	700 ms	500 ms	500 MB
46	6000	173M	0.9	3500 ms	2100 ms	2.6 GB
50	8300	480M	0.8	12000 ms	9000 ms	7.15 GB

4.4.7 Оптимизации расходов памяти

Одним из ключевых ресурсов, используемый вычислительной программой, является оперативная память (ОЗУ). Из представленных данных следует, что на большинстве стандартных компьютеров недостаточно ОЗУ для факторизации числа, содержащего 55 цифр. Внедрение решета позволило значительно повысить эффективность алгоритма, однако сопровождается значительным расходом памяти. Создание и хранение массива Sieve требует практически 95% процентов всей выделяемой программой памяти.

Распределение чисел

Заметим, что для нахождения набора $r_1 \dots r_m$ из равенства 12, достаточно набрать $factor_base + 1$ чисел, которые успешно раскладываются на простые множители с использованием набора простых чисел $primes$. Предлагается рассмотреть на распределение данных чисел:



Как видно из графика, в центре наблюдается увеличение количества, что следовало ожидать, так как значения функции $f(r)$ [13] приближены к нулю в центре, где $r \approx \sqrt{n}$. Тем не менее уменьшение размера множества $primes$ или сокращение диапазона приводит к недостаточному числу чисел, доступных для факторизации. Таким образом, возникает вопрос о возможности выбора альтернативной функции $f(r)$, которая позволила бы сократить размер используемого интервала.

Multiple Polynomials

Введем новую функцию:

$$F(r) = ar^2 + 2br + c \tag{16}$$

Заметим, что при $n = b^2 - ac$:

$$a \times F(r) = a^2 r^2 + 2abr + ac = a^2 r^2 + 2abr + b^2 - n = (ar + b)^2 - n \quad (17)$$

Если $a \times F(r)$ делится на p , то n является квадратичным остатком по модулю p , из чего следует, что при правильно подобранных параметрах не нужно будет изменять набор *primes*.

Подберем оптимальные параметры для функции $F(r)$. Одновременно минимизируем модуль минимального и максимального значения функции, которые могут получиться при прохождении отрезка $[-m, m]$.

Минимальное значение функции достигает при $ar = -b$, то есть $-F(-b/a) = \frac{n}{a}$.

Максимальное значение функции достигает при в точке $r = -m - \frac{b}{a}$.

$$F(-m - \frac{b}{a}) = F(m - \frac{b}{a}) = a \times m^2 - \frac{n}{a} \quad (18)$$

Получаем, что в качестве a стоит взять:

$$\frac{\sqrt{2n}}{m} \quad (19)$$

Также потребуем, чтобы a было простым, а n было квадратичным остатком по модулю a , тогда:

$$b^2 \equiv n \pmod{a} \quad (20)$$

$$c = \frac{b^2 - n}{a} \quad (21)$$

4.4.8 Преимущества и недостатки

Одним из основных преимуществ данного метода является высокая скорость работы по сравнению с остальными предложенными методами разложения. Основным недостатком является высокая потребность в ресурсах. При попытке разложения числа длиной в 50 знаков требуется 7.5 ГБ ОЗУ для поддержания вектора *Sieve*. Анализируя таблицу, можно предположить, что для факторизации 55-значного числа потребуется в 4-5 раз больше памяти, то есть около 32 ГБ, что превышает доступный объем памяти в большинстве компьютеров. Чтобы продолжить вычисления, придется перейти с ОЗУ на жесткий диск, что существенно замедлит исполнение программы.

4.5 Continued Fractions

4.5.1 Введение

Алгоритм CFRAC основывается на той же идее, что и Quadratic Sieve. Основное различие между ними заключается в методе нахождения пар x и y , удовлетворяющих условию $x^2 \equiv y^2 \pmod{p}$.

4.5.2 Нахождение нужной подпоследовательности

Из алгоритма[11] следует равенство:

$$P_i^2 - n \times Q_i^2 = (-1)^i \times C_i \quad (22)$$

Если же его рассмотреть по модулю n , то получим еще одно равенство, которое позволяет нам получить нужную подпоследовательность:

$$P_i^2 = (-1)^i \times C_i \pmod{n} \quad (23)$$

Получается, что CFRAC полностью сводится к алгоритму Quadratic Sieve[4.4], только вместо подбора подпоследовательности через решето, используется алгоритм Bhascara-Brouncker[11].

4.5.3 Подбор гиперпараметров и замеры времени работы

Как и в квадратичном решете, в CFRAC используется вектор *primes*, размер которого является гиперпараметром. Тестирование проводилось на числах вида $p_1 \times p_2$, где p_1 и p_2 - простые числа. Ниже приведены оптимальные значения параметров, а также среднее время выполнения программы

Number of Digits	Factor Base	Run Time
24	300	15 ms
30	300	110 ms
36	500	900 ms
40	1000	8000 ms

4.5.4 Преимущества и недостатки

Основное преимущество CFRAC перед Quadratic Sieve заключается в количестве памяти, необходимом для выполнения алгоритма. CFRAC требует только $\mathcal{O}(\text{factor_base})$ дополнительной памяти для хранения массива простых чисел. Более того, CFRAC требует меньшего количества простых чисел, чем Quadratic Sieve.

Однако основным недостатком CFRAC является скорость работы. Quadratic Sieve работает значительно быстрее по сравнению с CFRAC, что делает его предпочтительным выбором в случаях, когда скорость выполнения имеет первостепенное значение.

4.6 Elliptic Curve

4.6.1 Введение

Так как компьютер имеет ограниченную точность, то любое чисто с плавающей точкой можно привести к рациональной дроби, поэтому мы переходим от координат (x, y) к трехмерным координатам (X, Y, Z) , где $x = \frac{X}{Z}$, $y = \frac{Y}{Z}$. Воспользуемся определением эллиптической кривой[12] и подставим в него новые координаты:

$$\begin{aligned}\frac{Y^2}{Z^2} &= \frac{X^3}{Z^3} + a \times \frac{X}{Z} + b \\ Y^2 Z &= X^3 + a \times X \times Z^2 + b \times Z^3\end{aligned}\tag{24}$$

Теорема 20. Пусть набор (X, Y, Z) является решением равенства[24], определим:

$$(X_i, Y_i, Z_i) = \left(\frac{X_i}{Z_i}, \frac{Y_i}{Z_i}\right) = \left(\frac{X}{Z}, \frac{Y}{Z}\right)\#i[19].\tag{25}$$

Тогда:

$$X_{2i} = (X_i^2 - a \times Z_i^2)^2 - 8b \times X_i \times Z_i^3\tag{26}$$

$$Z_{2i} = 4Z_i \times (X_i^3 + a \times X_i \times Z_i^2 + b \times Z_i^3)\tag{27}$$

$$X_{2i+1} = Z \times [(X_i \times X_{i+1} - a \times Z_i \times Z_{i+1})^2 - 4b \times Z_i \times Z_{i+1} \times (X_i \times Z_{i+1} + X_{i+1} \times Z_i)]\tag{28}$$

$$Z_{2i+1} = X \times (X_{i+1} \times Z_i - X_i \times Z_{i+1})^2\tag{29}$$

4.6.2 Факторизация

Возьмем в качестве случайных величин X , Y и a , а в качестве b возьмем $y^2 - x^3 - ax \pmod n$. Начальное состояние обозначим за $(X, Y, 1)$. Тогда если p является делителем n и $k!$ делится на $|E(a, b)/p|$, то

$$(X, Y, Z)\#k! = (\dots(((X, Y, Z)\#1)\#2)\#3\dots)\#k\tag{30}$$

будет являться единичным элементом в группе $E(a, b)/p$, то есть будет делить $Z_{k!}$. Тогда если k не является слишком большим, то велик шанс того, что $\text{gcd}(Z_{k!}, n)$ является не тривиальным делителем.

4.6.3 Замеры времени работы

Эллиптические кривые представляют собой метод факторизации чисел, который становится предпочтительным при обработке длинных чисел, где традиционные методы, такие как квадратичное сито, начинают показывать себя неэффективно из-за значительных временных затрат и требований к ресурсам. Тестирование проходило на числах вида $p_1 \times p_2$, где p_1 и p_2 являются простыми.

Number of Digits	Run Time
70	15 min
75	50 min

4.6.4 Преимущества и недостатки

Главным преимуществом эллиптических кривых является их способность факторизовать очень длинные числа, которые практически невозможно разложить другими методами. Алгоритм стоит использовать для чисел длиной не меньше 60 знаков.

5 Алгоритмы проверки на простоту

5.1 Pseudo Prime Test

5.1.1 Введение

Для проведения проверки числа a на простоту воспользуемся теоремой [6]. Выберем некоторое количество простых чисел b , таких что $\gcd(a, b) \equiv 1$. Затем осуществим вычисления, проверяя условие для каждого выбранного простого числа b . Если данное условие выполнилось для всех выбранных b , то с относительно высокой вероятностью можно утверждать, что a является простым числом. Иначе гарантируется, что a является составным.

5.1.2 Устойчивость

Данный вид тестирования не обладает стойкостью из-за чисел, называемых числами Кармайкла [7], для которых тест всегда будет давать положительный результат, независимо от количества простых чисел, используемых в нем.

5.2 Strong Pseudo Prime Test

5.2.1 Введение

Представим число n в виде:

$$n = 2^a \times t + 1, \tag{31}$$

где t - нечетное, а a хотя бы 1.

Тогда для любого $b \in \mathbb{N}$ выполняется:

$$b^{n-1} - 1 = (b^t - 1) \times (b^t + 1) \times (b^{2t} + 1) \times \dots \times (b^{2^{a-1} \times t} + 1) \tag{32}$$

Тогда если $\gcd(b, n) = 1$ и n является простым, то n делит ровно 1 из делителей.

5.2.2 Устойчивость

Данный вид тестирования обладает большей устойчивостью, чем Pseudo Prime Test [5.2]. Например, если взять $b = 2$, то первым составным числом, прошедшим тест, будет 2047.

6 Библиотека LPN

6.1 Введение

Библиотека LPN [3] предоставляет реализацию методов Pollard Rho Factorization [4.2], Quadratic Sieve Factorization [4.4], CFRAC Factorization [4.5] и Elliptic Curve Factorization [4.6]. Каждый алгоритм описан в виде класса и имеет фиксированный набор статических публичных методов:

```
1 class MethodNameFactorization {
2 public:
3     static FactorSet Factorize(const long_int & n);
4     static FactorSet Factorize(const long_int & n, const Config & config);
5 }
```

- FactorSet - структура, содержащая в себе пары (делитель, кратность делителя).
- long_int - число с неограниченным количеством знаков, заимствован из библиотеки boost [1].
- Config - объект с гиперпараметрами, к примеру, для CFRAC это factor_base.

6.2 Сборка

Сборка происходит через утилиту Cmake[2]. Пользователю доступно несколько конфигураций сборки:

- Release - сборка с максимальным количеством оптимизаций от компилятора
- Debug - сборка с максимальным количеством отладочной информации
- MinSizeRel - сборка, максимально сжимающая размеры выходных файлов
- RelWithDebInfo - сборка с отладочной информацией, но и с некоторыми оптимизациями от компилятора

Пользователь должен убедиться, что на устройстве установлена утилита Cmake[2] версии 3.22 и выше, а также должна быть установлена библиотека boost[1] последней версии.

6.3 Тестирование

Библиотека имеет обширный набор тестов, которые доступны только в режиме отладки. Эти тесты предназначены для проверки корректности функционирования алгоритмов в различных сценариях использования. Тестирование происходит каждый раз при отправке кода в удаленный репозиторий[3]. Такой подход обеспечивает высокий уровень надежности библиотеки.

Список литературы

- [1] Boost provides free peer-reviewed portable c++ source libraries. URL: <https://www.boost.org/>.
- [2] Cmake: A powerful software build system. URL: <https://cmake.org/>.
- [3] Tigran Andrian. Git repository. URL: <https://github.com/tigran-edu/Large-Prime-Numbers>.
- [4] David M.Bressoud. *Factorization and primality testing*. 1989. URL: <https://link.springer.com/book/10.1007/978-1-4612-4544-5>.
- [5] Rivest–Shamir–Adleman. Rsa (cryptosystem). 1977. URL: <https://www.rsa.com/wp-content/uploads/rsa-security-overview-idplus-whitepaper.pdf>.
- [6] Peter Selinger. The miller-rabin primality test. URL: <https://www.mathstat.dal.ca/~selinger/courses/2014F-4116/downloads/handout3.pdf>.
- [7] Wikipedia. Carmichael number. URL: https://en.wikipedia.org/wiki/Carmichael_number.