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

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Большие простые числа

Выполнил:
Студент группы БПМИ226 _____ Либина Яна Максимовна
Подпись

26.04.2024
Дата

Принял:
Руководитель проекта Елена Юрьевна Колесниченко
Имя, Отчество, Фамилия
доцент, к.ф.-м.н.
Должность, ученое звание
ДБДИП ФКН НИУ ВШЭ
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки _____ 2024 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2024

Содержание

Аннотация

В данной работе имплементируются методы факторизации чисел и их тестирования на простоту. Большое внимание уделяется вероятностным тестам на простоту больших чисел, не влезающих в диапазон встроенных целочисленных типов C++.

Ключевые слова: теория чисел, большие числа, простые числа, делители, факторизация, тест на простоту, псевдопростые числа, сильно псевдопростые числа, вероятностный тест, детерминированный тест, криптосистема

1 Введение

1.1 Описание предметной области

Большие простые числа представляют особый интерес в области криптографии. Так, например, стойкость криптосистемы RSA основывается как раз на том, что легко перемножить два больших простых числа, но факторизовать (разложить на множители) полученное число оказывается нетривиальной, или даже невыполнимой за обозримое время задачей. Это характеризует так называемые *односторонние* функции, и односторонние они не в том смысле, что вычисление обратного значения теоретически невозможно, но в том, что оно практически **очень** затруднено.

Задача факторизации числа является двойственной к задаче проверке числа на простоту. Такие проверки чаще называются *тестами*, и их эффективность и надежность также важна для работы криптосистем, ведь большие простые числа нужно уметь генерировать.

Существует несколько характеристик тестов на простоту, различающие их: универсальность, безусловность, полиномиальность и детерминированность.

- *Универсальность* означает, что тест подходит для чисел любого вида.
- *Безусловность* означает, что тест не основывается на недоказанных гипотезах (чаще всего обобщенная гипотеза Римана).
- *Полиномиальность* означает, что вычислительная сложность (время работы) ограничена многочленом от числа цифр у проверяемого числа.
- *Детерминированность* (истинность) означает, что если число было объявлено простым, то оно точно простое.

На данный момент существует лишь один алгоритм, обладающий всеми четырьмя свойствами - тест Агравала-Каяла-Саксены (AKS). Однако, он требует слишком много памяти и носит теоретический характер.

В общем случае детерминированные тесты на простоту для чисел большого размера требуют либо слишком больших вычислительных мощностей, либо слишком много времени, либо являются условными. Поэтому чаще используются или детерминированные тесты для определенного вида чисел (тест Люка-Лемера для чисел Мерсенна), или вероятностные тесты, т.е. такие, которые отличают простое число с определённой вероятностью (тест Миллера-Рабина).

1.2 Постановка задачи и результаты

Эта работа ставит перед собой задачу изучения методов и алгоритмов теории чисел, направленные на поиск делителей числа и определение простоты, применительно к большим числам (порядка 300 бит, примерно 100 десятичных знаков), а также их реализация и тестирование. Среди тестов на простоту приоритет отдаётся вероятностным тестам на простоту. В конце реализованы работа криптосистемы RSA и попытки её "взломать" с использованием имплементированных ранее методов.

В итоге выполнено:

- вероятностные алгоритмы на простоту;
- алгоритмы, эффективно находящие делители рядом с корнем числа (факторизация Ферма) и делители порядка $10^5 - 10^{10}$ (алгоритмы Полларда);
- генерация большого простого числа;

- работа криптосистемы RSA;
- тестирование вышеперечисленных методов;
- сравнительный анализ полученных результатов.

Результатом работы является программное решение поставленной задачи, предоставляющее все перечисленные выше методы.

1.3 Структура работы

Прежде всего, в секции 4 будут изложены необходимые теоретические знания для понимания контекста работы и формулировок алгоритмов.

Далее, в разделе 5.1 будут представлены алгоритмы, использующиеся для вспомогательных вычислений, в разделе 5.2 алгоритмы факторизации и в разделе 5.3- алгоритмы тестов на простоту.

В секции 6 я рассмотрю применения имплементированных алгоритмов, такие как генерация простого числа и работа криптосистемы RSA.

В секции 7 будут приведены некоторые особенности предложенной программной реализации.

В секции 8 представлены проведённые тестирования и их результаты.

Ссылка на репозиторий: [1]

2 Обзор литературы

Большая часть базовой теории, лежащей в основе алгоритмов, содержана в серии лекций [2] по теории чисел А.В.Устинова. Оттуда же были взяты основы реализации криптосистемы RSA. Основным источником идей для алгоритмов и последовательности их реализации является книга Дэвида М. Брессуда "Factorization and Primality Testing"[3]. Оттуда были получены базовые представления о различных нетривиальных алгоритмах факторизации, таких как алгоритм Ферма и алгоритмы Полларда, и о тестах на простоту Люка-Лемера, Ферма и Миллера-Рабина. Более подробно о тесте Миллера-Рабина я узнала из оригинальной статьи Рабина [4], там же описывается процесс генерации большого простого числа, используя тест Миллера-Рабина. В той же книге [3] Брессуда описываются последовательности Люка, а алгоритм их эффективного вычисления был взят из статьи [5] Алексея Коваля "On Lucas Sequences Computations". В статье [6] Бэйли и Уогстаффа было предложено определение псевдопростых и сильно псевдопростых чисел Люка, и это определение используется для теста Люка. В той же статье предложен BPSW тест, являющийся комбинацией теста на сильную псевдопростоту по основанию 2 и сильную псевдопростоту Люка.

3 Описание функциональных и нефункциональных требований

3.1 Функциональные требования

Программа предоставляет следующие **вспомогательные методы**, использующиеся в алгоритмах факторизации и тестирования на простоту: быстрое возведение в степень, алгоритм Евклида, расширенный алгоритм Евклида, предложенный Кнутом [3, стр. 10], вычисление наименьшего целого числа, большего или равного $\sqrt{n}$, вычисление символа Якоби.

Методы факторизации: перебор делителей до максимума, алгоритм Ферма, ро-алгоритм Полларда, $p-1$-алгоритм Полларда.

Тесты простоты: перебор делителей, тест Ферма, тест Миллера-Рабина, тест Люка на псевдопростоту, тест Бейли—Померанца—Селфриджа—Уогстаффа.

Также предоставляется метод для генерации простого числа и отдельно представлены методы кодирования и декодирования из криптосистемы RSA.

3.2 Нефункциональные требования

Программа выполняется на языке C++17, программа запускалась с компилятором gcc. Используется библиотека boost::multiprecision 1.84.0 для работы с большими числами. Для тестирования используется библиотека Google C++ Testing Framework. Для вывода результата теста на простоту в консоль используется библиотека "Magic Enum". Используется система сборки CMake 3.22, система поддержки версий git вместе с Github.

Все неободимые зависимости подключаются на этапе сборки через FetchContent в CMake. Пререквизиты для алгоритмов проверяются при помощи assert. Стиль кода описан в файлах .clang-format и .clang-tidy.

4 Теоретический минимум

Определение 1. Пусть $a = b * q + r, 0 \leq r < b$. Тогда r называется *остатком* от числа a по модулю b. Также говорят, что a сравнимо с r по модулю b, что записывается как $a \equiv r (\mathrm{mod}\ b)$.

Определение 2. Простым числом Мерсенна называется простое число вида

$$M(n) = 2^n - 1,$$

где n - нечётное простое число.

Замечание. Если n - составное, то $M(n)$ - составное.

Утверждение 3 (Критерий простоты чисел Мерсенна)**.** *Пусть n - нечётное простое. Число $M(n)$ простое тогда, и только тогда, когда $M(n)$ делится на S_{n-1}, где S_k - последовательность, задаваемая рекуррентно:*

$$S_k = \begin{cases} 4 & k = 1 \\ S_{k-1}^2 - 2, & k > 1 \end{cases}$$

Утверждение 4 (Малая теорма Ферма)**.** *Если p - простое число и $(a, p) = 1$, то*

$$a^{p-1} \equiv 1 (mod\ p).$$

Определение 5. Составное число n называется *псевдопростым Ферма по основанию* a, если $(a, n) = 1$ и

$$a^{n-1} \equiv 1 (\mathrm{mod}\ n).$$

NB! Если мы не знаем точно, что n - составное, и n удовлетворяет сравнению, то называем n *возможно простым* (TestStatus::ProbablyPrime).

Определение 6. Числа, являющиеся псевдопростыми по всем взаимно простыми с ними основаниями, называются *числами Кармайкла.*

Определение 7. Составаное число $n = d * 2^s + 1$ (d - нечётное) называется *сильно псевдопростым* числом по основанию a, если

$$a^d \equiv 1 (\mathrm{mod}\ n)$$

или

$$a^{d*2^r} \equiv -1 (\mathrm{mod}\ n) \text{ для некоторого } 0 <= r < \mathrm{s}$$

Если выполняется хотя бы одно из условий, то число a называется *свидетелем простоты* числа n.
Если оба утверждения неверны, то число a называется свидетелем того, что число n - составное.

NB! Аналогично, если мы не знаем, что n - составное, то будем называть n *сильно возможным простым* (TestStatus::StrongProbablyPrime).

Замечание. Не существует чисел, сильно псевдопростых по всем взаимно простым с ними основаниям.

Определение 8. Если сравнение $x^2 \equiv a (\mathrm{mod}\ m)$ имеет решение (относительно x), то число a называется *квадратичным вычетом* по модулю m. Иначе, a называется *квадратичным невычетом* по модулю m.

Определение 9. *Символом Лежандра* $(\frac{a}{p})$ для целого a и простого p называется величина, определяемая как:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & a \text{ делится на } p \\ 1, & a \text{ - квадратичный вычет по модулю } p \\ -1, & a \text{ - квадратичный невычет по модулю } p \end{cases}$$

Свойства символа Лежандра:

1. $a \equiv b (\mathrm{mod}\ p) \Rightarrow \left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$

2. $\left(\frac{a*b}{p}\right) = \left(\frac{a}{p}\right) * \left(\frac{b}{p}\right)$
3. $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} (\text{mod } p)$ (Критерий Эйлера)
4. $\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$
5. $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}$
6. $\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}} \left(\frac{p}{q}\right)$ (Квадратичный закон взаимности)

Определение 10. *Символом Якоби* $(\frac{a}{N})$ для целого a и $N = p_1 p_2 .. p_s$ называется величина, определяемая как:

$$\left(\frac{a}{N}\right) = \left(\frac{a}{p_1}\right) * \left(\frac{a}{p_2}\right) * ... * \left(\frac{a}{p_s}\right),$$

где $\left(\frac{a}{p_i}\right)$ - символ Лежандра.

Символ Якоби обладает теми же свойствами, что и символ Лежандра + мультипликативность знаменателя:

$$\left(\frac{a}{m*n}\right) = \left(\frac{a}{m}\right) * \left(\frac{a}{n}\right)$$

Определение 11 (*Последовательности Люка*)**.** Пусть P, Q - такие целые числа, что $D = P^2 - 4Q \neq 0$, $P > 0$. Тогда *последовательностями Люка* $U_n(P, Q)$ и $V_n(P, Q)$ называются последовательности:

$$U_k(P, Q) = \frac{\alpha^k - \beta^k}{\alpha - \beta},$$

$$V_k(P, Q) = \alpha^k + \beta^k,$$

где α, β - корни уравнения $x^2 - Px + Q = 0$.

Утверждение 12. *Последовательности Люка можно задать следующими рекуррентными соотношениями:*

$$U_0 = 0, U_1 = 1, U_k(P, Q) = PU_{k-1} - QU_{k-2}$$
$$V_0 = 2, V_1 = P, V_k(P, Q) = PV_{k-1} - QV_{k-2}$$

Утверждение 13 (Свойства последовательностей Люка)**.**

$$DU_k = 2V_{k+1} - PV_k \Leftrightarrow U_k = \frac{2V_{k+1} - PV_k}{P^2 - 4Q} \tag{1}$$

$$V_{n+m} = V_n V_m - Q^m V_{n-m} \tag{2}$$

Именно эти свойства будут использоваться для вычисления последовательностей.

Определение 14. В условиях предыдущего определения:

Нечётное составное число n называется *псевдопростым числом Люка* с параметрами (P, Q), если

$$n \nmid Q, \left(\frac{D}{n}\right) = -1 \text{ и } n \mid U_{n+1}.$$

Определение 15. Обозначим $\delta(n) = n - \left(\frac{D}{n}\right)$, пусть $\delta(n) = d * 2^s$, где d нечётное.

Нечётное составное число n называется *сильным псевдопростым числом Люка*, если

$\left(\frac{n}{D}\right) = 1$ и выполняется одно из условий:

- $U_d \equiv 0 \mod n$, или
- $V_{d*2^r} \equiv 0 \mod$ n для какого то $0 < r <= s$.

Определение 16. *Сверхсильным псевдопростым числом Люка* называется сильное псевдопростое число люка с параметрами (P, 1) с модифицированным условием:

- $U_d \equiv 0 \mod n$ **и** $V_d \equiv \pm 2 \mod n$, или
- $V_{d*2^r} \equiv 0 \mod$ n для какого то $0 < r <= s$.

5 Алгоритмы

5.1 Вспомогательные алгоритмы

Алгоритм 1 (*Алгоритм Евклида*)**.**
Вход: положительные числа a, b.
Выход: НОД(a, b).

Пусть есть последовательность чисел $a > b > r_1 > ... > r_n$, таких что:
$a = b * q_0 + r_1$
$b = r_1 * q_1 + r_2$
$r_1 = r_2 * q_2 + r_3$
...
$r_{n-1} = r_n * q_n$
Тогда $r_n =$ НОД(a, b).
Асимптотика алгоритма: $O(log(min(a, b)))$

Утверждение. *Если a и b - целые числа и хотя бы одно из них не 0, то существуют такие целые числа m и n, что $(a, b) = m * a + n * b$.*

Алгоритм 2 (*Расширенный алгоритм Евклида, Knuth GCD*)**.** Алгоритм находит коэффициенты m и n из предыдущего утверждения.

Вход: положительные числа a, b.

Выход: $m, n, (a, b)$, такие что $(a, b) = m * a + n * b$.

Шаги алгоритма:
$r_1 = a - bq_0$
$r_2 = b - r_1q_1 = b - (a - bq_0)q_1 = b(1 + q_0q_1) - a * q_1$
$r_3 = r_1 - r_2q_2 = a(1 + q_1q_2) - b(q_0 + q_2 + q_0q_1q_2)$
...
$r_n = ax + by$

Во многих задачах, связанных с простыми числами, требуется возводить числа в степень по модулю. Для этого будем пользоваться алгоритмом бинарного возведения в степень.

Алгоритм 3 (*Бинарное возведение в степень*)**.**
Вход: целые числа $x, a >= 0, n > 0$
Выход: $x^a \bmod n$.

Представим число a в бинарном виде:

$$a = a_k 2^k + a_{k-1} 2^{k-1} + ... + a_1 2 + a_0, \text{ где } a_i \in 0, 1.$$

Изначально результат равен 1, положим $z = x$. Далее для $i = \overline{0, .., k-1}$, если $a_i = 1$, то результат умножается на z по модулю n. Затем для любого a_i число z возводится в квадрат.
Асимптотика: $O(\log_2(n))$

Алгоритм 4 (*Поиск $\lceil\sqrt{n}\rceil$*)**.**
Вход: целое число $n > 0$
Выход: $\lceil\sqrt{n}\rceil$.

Воспользуемся бинарным поиском. Положим изначально $l = 1, r = n$. Теперь, пока $l < r$ смотрим, является ли середина отрезка $m = \frac{l+r}{2}$ меньше $\sqrt{n}$ ($\Leftrightarrow m * m < n$). Если да, то сдвигаем левую границу на $m + 1$, в противном случае $r = m$. Если в какой-то момент $m * m = n$, то n - полный квадрат и возвращается m. После всего цикла возвращается l.
Асимптотика: $O(\log_2(n))$

Алгоритм 5 (*Вычисление последовательностей Люка*)**.**
Вход: целые числа $P > 0, Q$, такие что $P^2 - 4Q \neq 0$, $k > 0$
Выход: члены последовательности Люка $U_k(P, Q), V_k(P, Q)$.

Представим число k в бинарном виде:

$$k = \sum_{i=0}^{n-1} k_i 2^i, \ n = \lceil \log_2 k \rceil, k_i \in 0, 1, k_{n-1} = 1$$

Обозначим

$$K_j = \sum_{i=j}^{n-1} k_i 2^{i-j}$$

Тогда $k = K_0$ и

$$K_{j-1} = k_{j-1} + \sum_{i=j}^{n-1} k_i 2^{i-j+1} = k_{j-1} + 2K_j = (K_j + k_{j-1}) + K_j$$

$$K_{j-1} + 1 = (K_j + k_{j-1}) + K_j + 1$$

По свойству 2:

$$V_{K_{j-1}} = V_{K_j + k_{j-1}} V_{K_j} - Q^{K_j} V_{k_{j-1}} \tag{3}$$

На j-й итерации положим $l_j = K_j$, $h_j = K_j + 1$, при этом итерации нумеруются от $n-1$ до 0

$$Q^{l_j} = Q^{K_j} = Q^{2K_{j+1}+k_j} = Q^{l_{j+1}} Q^{h_{j+1}} Q^{k_j}$$
$$Q^{h_j} = Q^{K_j+1} = Q^{2K_{j+1}+k_j+1} = Q^{l_{j+1}} Q^{h_{j+1}} Q Q^{k_j}$$

Тогда при $k_{j-1} = 1$ имеем

$$V_{Kj-1} = V_{K_j+1} V_{K_j} - Q^{K_j} V_1 \Leftrightarrow V_{l_{j-1}} = V_{h_j} V_{l_j} - P Q^{l_j} \text{ и}$$

$$V_{K_{j-1}+1} = V_{K_j+1} V_{K_j+1} - Q^{K_j+1} V_0 \Leftrightarrow V_{h_{j-1}} = V_{h_j}^2 - 2Q^{h_j}$$

А при $k_{j-1} = 0$ имеем

$$V_{l_{j-1}} = V_{l_j}^2 - 2Q^{h_j}$$
$$V_{h_{j-1}} = V_{l_j} V_{h_j} - P Q^{l_j}$$

Замечу, что при нашей нумерации итераций шаг $j-1$ следует за j, поэтому формулы выше - формулы перехода на новую итерацию.

Изначально положим $V_{l_{n-1}} = 2, V_{h_{n-1}} = P$ (начальные значения последовательности V_k. В конце цикла $V_l = V_{K_0} = V_k$), $Q_l = Q_h = 1$.

Рассматриваем j от $n-1$ до 0:

Воспользуемся формулой 3, чтобы рекурсивно вычислить V_k. Пусть мы находимся на j-й итерации.

1. $Q^l = Q^l * Q^h$
2. Если $k_j = 1$, то
   - $Q_h = Q_l * Q$
   - $V_l = V_h * V_l - P * Q_l$
   - $V_h = V_h^2 - 2Q_h$
3. Иначе, если $k_j = 0$, то
   - $Q_h = Q_l$
   - $V_l = V_l^2 - 2Q_j$
   - $V_h = V_l * V_h - P * Q_l$

Когда выходим из цикла, $V_l = V_k$. По формуле 1 считаем

$$U_k = \frac{2V_{k+1} - PV_k}{P^2 - 4Q}.$$

Возвращаем U_k, V_k.

5.2 Алгоритмы факторизации

Алгоритм 6 (*Перебор делителей*)**.**
Вход: целое число $n > 0$.
Выход: разложение вида

$$n = p_1^{e_1} p_2^{e_2} ... p_r^{e^r} * f,$$

где $p_1, ..., p_r$ - различные простые, а $f > \sqrt{n}$ - неразложенная часть.

f возникает из-за того, что для больших чисел полный перебор делителей займёт слишком много времени, поэтому в большинстве случаев нас будет интересовать разложение на простые хотя бы до 5000.

Положим $f = n$. Для начала выделим наибольшие степени 2 и 3 из f. Теперь, когда мы хотим рассматривать числа, не кратные 2 и 3, мы можем рассматривать числа вида $6k + 1$ и $6k + 5$. Положим $d = 5$ и переменную $add = 2$. Пока d не превзошло заданного максимума (например, 5000) и $d^2 <= f$, делим f на d «до упора», т.е. выделяем максимальную степень d из f, и затем прибавляем $d = d + add$ и $add = 6 - add$. Таким образом, переменная add будет меняться между 2 и 4, а число d между $6k + 1$ и $6k + 5$.

Если в какой-то момент $d^2 > f$, то f - простое, и мы полностью разложили число n на простые. Иначе, в итоге f может остаться как простым, так и составным.

Алгоритм 7 (*Алгоритм факторизации Ферма*)**.**
Вход: нечётное число $n > 0$
Выход: делитель числа n.

Пусть дано нечётное число n. Наблюдение Ферма заключается в том, что любое нечётное число можно представить в виде разности двух полных квадратов:

$$n = x^2 - y^2 = (x - y)(x + y).$$

Таким образом, $(x - y)$ и $(x + y)$ - делители числа n. Если какое-то из них составное, можем продолжить раскладывать.

Положим

$$r = x^2 - y^2 - n$$

и будем перебирать по очереди y и x. Если $r = 0$, то разложение найдено.

Будем искать делители от ближайших к $\sqrt{n}$. Начнём с $x = \lceil \sqrt{n} \rceil, y = 1$. Пока $r > 0$, увеличиваем y. Как только r стало меньше 0, увеличиваем x на 1 и переходим к следующей итерации. И т.д., пока r не станет 0.

Стоит отметить, что необязательно следить за x и y по отдельности, потому что нас интересует только изменение $r = x^2 - y^2 - n$. Заметим, что $(t + 1)^2 - t^2 = 2t + 1$. Обозначим переменными $u = 2x + 1, v = 2y + 1$. Тогда,

$$x \mathrel{+}= 1 \Rightarrow u \mathrel{+}= 2;\ y \mathrel{+}= 1 \Rightarrow v \mathrel{+}= 2.$$

В итоге, полученные делители - $\frac{u+v-2}{2}$ и $\frac{u-v}{2}$.

NB! Этот алгоритм хорошо ищет делители, близкие к корню и очень неэффективен, если делители далеко и тем более, если число простое.

Асимптотика: Если $n = pq$, то время работы: $O(|p - q|)$

Алгоритм 8 (ρ-*алгоритм Полларда*)**.**
Вход: составное число $n > 0$, положительные число max_iter (макс. кол-во итераций), x_0, c
Выход: делитель числа n или -1, если делитель не был найден. Пусть есть полином $f(x) = x^2 + c$, а также d - неизвестный нетривиальный делитель n.

Рассмотрим рекурсивную последовательность $x_i = f(x_{i-1}) \bmod n$ и последовательность $y_i = x_i \bmod d$. Так как вычетов по модулю d конечное число, в какой-то момент произойдет зацикливание $y_i = y_j$, а значит $x_i \equiv x_j \bmod d \Rightarrow d|(x_i - x_j)$. Отсюда можно сделать вывод, что $d|gcd(n, x_i - x_j)$ и если $g = gcd(n, x_i - x_j) > 1$ и $g! = n$, то g -нетривиальный делитель n.

Так как d заранее неизвестно, нужно уметь искать подходящие пары (i,j). R.P. Brent [3, стр. 64] предлагает смотреть на разности вида $x_{2^i-1} - x_j, 2^{i+1} - 2^{i-1} <= j <= 2^{i+1} - 1$. Последовательность разностей начинает так:

$$x_1 - x_3$$

$$x_3 - x_6$$

$$x_3 - x_7$$

$$x_7 - x_{12}$$

$$x_7 - x_{13}$$

$$x_7 - x_{14}$$

$$x_7 - x_{15}$$

...

Разница между координатами увеличивается на один, и когда 2^i станет больше, чем длина цикла, то мы найдём нужную разность.

Чтобы много раз не вычислять GCD, будем накапливать произведение этих разностей и каждые 10 итераций проверять $g = gcd(\prod(x_i - x_j))$. Если в какой-то момент он не 0, то возвращаем g. Если он стал n или за максимальное число итераций не нашлось g, то возвращаем -1 и стоит пробовать ещё раз с другими параметрами x_0 и c.

Алгоритм 9 ($p-1$-*алгоритм Полларда*)**.**
Вход: составное число $n > 0$, положительное число max_iter (макс. кол-во итераций), целое число $c > 0$
Выход: делитель числа n или -1, если делитель не был найден.

Пусть p - простой делитель числа n и $(p-1)|10000!$

Пусть $m = c^{10000!} \bmod n,\ (c, n) = 1$

По МТФ, $c^{p-1} \equiv 1 \bmod p \Rightarrow m \equiv 1 \bmod p \Rightarrow p|(m-1)$.

Велик шанс того, что $n \not| (m-1)$, откуда $gcd(n, m-1)$ является делителем n.

В цикле будем накапливать степени m. Каждые 10 итераций будем проверять $gcd(n, m-1)$, и если он >1, то он является нетривиальным делителем.

Если делитель не был найден за указанное число итераций, можно попробовать еще раз с другим значением c.

5.3 Тесты на простоту

Во всех алгоритмах этого раздела на вход подаётся положительное число n, на вызоде возвращается значение из перечисления TestStatus, то есть либо простое, либо возможно простое, либо сильно возможно простое, либо составное.

Алгоритм 10 (*Перебор делителей до заданного максимального*)**.** Если значение max не задано, то $max = \lceil\sqrt{n}\rceil$. Переберём все числа от 3 до max. Если число n делится на какое-то из них, то n - составное. Если нет, то число не имеет делителей $<= max$. В частности, если $max = \lceil\sqrt{n}\rceil$, то n - простое.
Асимптотика: $O(max)$

Алгоритм 11 (*Тест Люка-Лемера для простых чисел Мерсенна*)**.** Дано нечётное число $n >= 3$. Хотим определить, является ли число $M(n) = 2^n - 1$ простым.

Воспользуемся критерием простоты для чисел Мерсенна. Зададим число $S_1 = 4$. Посчитаем последовательность $S_k = S_{k-1}^2 - 1 \bmod M(n)$ вплоть до S_{n-1} члена. Если $S_{n-1} = 0$, то $M(n)$ - простое. Иначе - составное.

Алгоритм 12 (*Тест Ферма* на псевдопростоту)**.** Если для числа n и числа a: $(a, n) = 1$ выполняется малая теорема Ферма (т.е. $a^{n-1} \equiv 1 \bmod n$), то n - возможно простое.

Асимптотика: $O(log(2n) * log(log(n)) * log(log(log(n))))$ (с использованием быстрого возведения в степень по модулю n)

Алгоритм 13 (*Тест Миллера-Рабина* на сильную псевдопростоту)**.** Хотим проверить, является ли число n сильно возможным простым.

Зададим число итераций k. Идея алгоритма заключается в том, чтобы для k случайно выбранных чисел a проверить, являются ли они свидетелями простоты числа n. Если хотя бы одно из них оказалось свидетелем того, что n - составное, то n действительно составное число. Иначе n - сильно возможно простое, и оно простое с вероятностью $(\frac{1}{4})^k$.
Для увеличения производительности в начале алгоритма переберем маленькие делители до 5000. Отдельно рассмотрим оптимизацию с дополнительным проведением теста Ферма после перебора делителей. Если после проверок n не было объявлено составным, то переходим к основному тесту.
Асимптотика: $O(klog^3(n))$

Алгоритм 14 (*Тест Люка на сильную псевдопростоту*)**.**

1. Для поиска P и Q воспользуемся алгоритмом Сэлфриджа, описанного в статье [6, стр.1401]:

   Пусть D - первое число в последовательности 5, -7, 9, -11, 13, ..., такое что $\left(\frac{D}{n}\right) = -1$. Здесь стоит отметить, что, во-первых, если в какой-то момент мы встретили $\left(\frac{D}{n}\right) = 0$, то мы нашли делитель числа n, значит, оно составное. Во-вторых, если n является полным квадратом, то $\left(\frac{D}{n}\right) > -1$ для любого D, поэтому стоит исключить такую возможность.

   Если D нашлось, то $P = 1$, $Q = \frac{1-D}{4}$.

2. Проведём тест на сильную псевдопростоту Люка согласно определению 15.

Алгоритм 15 (*BPSW тест*)**.**

1. Прежде всего, переберём делители числа n до 1000.
2. Проведём тест Миллера-Рабина единоразово по основанию 2
3. Проведём тест Люка на сильную простоту.

Если число n прошло все проверки, то **очень** вероятно, что оно простое. Все составные числа до 2^64 не проходят этот тест, и не найдено ещё ни одного составного числа больше 2^64, которое проходило бы этот тест.

6 Применение

6.1 Генерация простого числа

В программе реализован следующий метод генерации простого числа:

Алгоритм 16 (*Случайное простое число заданной длины*)**.**
Вход: длина числа в битах $m > 0$
Выход: простое число длины m

Генерируется двоичная последовательность нужной длины, например, 250 бит, где каждый бит сгенерирован случайно - $t = \alpha_1\alpha_2...\alpha_{250},\ \alpha_1 = 1$

Пусть t нечётное. Будем проверять числа t, t+2, t+4 и т.д. тестом BPSW, пока не найдем простое. Плотность простых чисел среди чисел длиной 250 бит можно грубо оценить как $\frac{1}{log250} < \frac{1}{250}$, поэтому если за 300-500 проб не нашлось простого числа, стоит заново сгенерировать число t и начать сначала.

6.2 Криптосистема RSA

Криптосистема RSA называется системой с публичным ключом, или ассиметричной системой. Эта система предполагает, что существует открытый ключ e (открытая экспонента), известный всем, и секретный ключ d, известный только участнику переписки, принимающему зашифрованное сообщение.

Функции шифрования и дешифровки в RSA имеют следующий вид:

$$Enc(m) = m^e \bmod n = c,\ Dec(c) = c^d \bmod n$$

Эти функции взаимно обратные и лишь знание e не позволяет вычислить d, так как это сводится к факторизации n.

Алгоритм 17 (*Генерация ключей для RSA*)**.**
Вход: длина ключа в битах
Выход: открытый ключ (e, n), закрытый ключ d

1. Генерируем два простых числа p, q заданной длины. Желательно, чтобы делители чисел $p-1$ и $q-1$ были большими простыми. Этого можно добиться, выбирая p и q среди простых чисел вида $2ip' + 1, 2iq' + 1$, где p', q' - сгенерированные простые нужной длины.
2. n = pq
3. $\phi(n) = (p-1)(q-1)$ ($\phi(n)$ - функция Лежандра)

4. Выберем не слишком маленькое число e: $1 < e < \phi(n)$, $(e, n) = 1$

5. Зная p-1 и q-1, найдём d такое, что $de \equiv 1 \bmod \phi(n)$ с помощью расширенного алгоритма Евклида $(ed + nc = 1)$

   Пара (e, n) - публикуется как открытый ключ.

   Число d - закрытый ключ, держится в секрете.

Алгоритм 18 (*Шифрование RSA*).
Вход: сообщение m, модуль n, открытая экспонента e
Выход: зашифрованное сообщение c

1. Преобразуем сообщение m в число, где каждому символу в верхнем регистре сопоставляется его номер ASCII

2. Вычислим $c = m^e \bmod n$

Алгоритм 19 (*Дешифрование RSA*).
Вход: защифрованное сообщение c, открытый ключ (e, n), секретный ключ d
Выход: расшифрованное сообщение m

1. Вычислим $m = c^d \bmod n$

2. Сопоставим каждому числу из двух цифр в m символ из таблицы ASCII.

7 Структура программы

1. Файлы в программе распределены по папкам:

   - utilities со вспомогательными алгоритмами и хедерами.
   - factorization с алгоритмами факторизации
   - test_primality с проверками на простоту
   - RSA с методоми для RSA
   - tests с тестированиями

2. В корне проекта находится файл main.cpp, в котором продемонстрирована работа программы.
3. Весь код обернут в namespace Proj
4. В папке utilities находится файл large_type.h, в котором объявляется псевдоним

```
1    using large_int = boost::multiprecision::cpp_int;
```

Этот файл подключается во все остальные файлы программы и под типом large_int в данной работе подразумевается именно cpp_int из библиотеки Boost. Однако, при необходимости разработчика этот псевдоним можно поменять на другой тип и подстроить особенности программы под него.

5. У тестов на простоту возвращаемым значением является значение из перечисления TestStatus, объявленного в файле utilities/test_status.h

```
1    namespace Proj {
2    enum class TestStatus { Prime, StrongProbablyPrime, ProbablyPrime, Composite };
3    }
```

6. Возвращаемым значением факторизации методом перебора делителей является значение из структуры TrialFactors, объявленной в файле factorization/trial_division.h

```
1    struct TrialFactors {
2        std::vector<large_int> p;
3        std::vector<int> exp;
4        large_int f;
5    };
```

7. Для эмулирования работы RSA используетсяч хедер RSA/rsa.h:

```
namespace Proj {

class RSA {
public:

    RSA(int bits_cnt);

    size_t e;
    large_int n;
    void decode(size_t c);

private:
    size_t m_;
    size_t d_;
    void generate_keys(int bits_cnt);
};

size_t rsa_encode(size_t e, large_int n, std::string m);

}
```

Тогда возможный вид реализации выглядит как:

```
    Proj::RSA alice(300);
    size_t e = alice.e;
    Proj::large_int n = alice.n;
    std::string m = "super secret message";
    size_t c = Proj::rsa_encode(e, n, m);
    alice.decode(c);
```

В результате в приватном поле m_ экземпляра класса RSA alice будет лежать расшифрованное сообщение.

8 Тестирование алгоритмов

8.1 Тестирование на корректность

Все методы покрыты тестами на корректность.

1. Для тестирования вспомогательных функций результат проверяется либо путём сравнения с известным результатом, либо проверкой напрямую. Например, результат поиска $\lceil \sqrt{n} \rceil = res$ верен, если $(res-1)^2 < n <= res^2$.

2. Для тестирования алгоритма Люка-Лемерса используется заранее известный список показателей tests/mersenne_exp.h, дающих число Мерсенна.

3. Для тестирования алгоритма Миллера-Рабина используется заранее известный список простых чисел (tests/small_primes.h), а также числа, которые тест Люка-Лемерса определил как простые.

4. Для тестирования остальных алгоритмов на простоту используется заранее известный список простых чисел и простые числа, сгенерированные с помощью теста Миллера-Рабина.

5. Для тестирования методов факторизации используются либо заранее известные числа, либо произведение двух или более сгенерированных чисел

6. Для тестирования корректности работы криптосистемы RSA проверяется совпадение изначального и декодированного сообщений.

8.2 Сравнительные тесты

Тесты выполняются на компьютере с ОС Windows 10, процессор i5-9300H, 2400 МГц, 4 ядра, 16ГБ RAM.

Различные алгоритмы сравниваются по времени работы и количеству ошибок для чисел разного размера. Подробные результаты тестирований будут представлены на защите.

1. Сравнение алгоритма Миллера-Рабина, включающего тест Ферма, и обычной версии.

   Оба алгоритма не совершили ни одной ошибки на входных данных.

2. Сравнение Миллера-Рабина и теста Ферма.

3. Сравнение Миллера-Рабина и BPSW.

4. Сравнение метода факторизации Ферма и методов Полларда-Ро.

5. Атака на RSA.

9 Заключение

В результате проделанной работы был сделан вывод, что тест BPSW является оптимальным из всех предложенных, т.к. у него минимальная возможность ошибки и хорошее время выполнения. Однако, тест Миллера-Рабина служит хорошей альтернативой.

Алгоритмы факторизации Ферма и Полларда не справляются с поиском больших делителей, однако быстро находят небольшие, что подчеркивает важность выбора для RSA таких p и q, чтобы делители $p-1$ и $q-1$ были как можно больше.

В целом была проведена большая теоретическая работа и я изучила множество новой информации.

Список литературы

[1] URL: https://github.com/yalibina/Large_Prime_Numbers.

[2] А. В. Устинов. “Теория чисел”. 2023. URL: https://drive.google.com/file/d/1Cj7MMGnnc8sHbaajGIuj_WaU7XJppNeG/view.

[3] David M. Bressoud. *Factorization and primality testing*. Undergraduate Texts in Mathematics. New York: Springer-Verlag, 1989.

[4] Michael O. Rabin. “Probabilistic algorithm for testing primality”. В: *Journal of Number Theory* 12.1 (1980), с. 128—138. URL: https://www.sciencedirect.com/science/article/pii/0022314X80900840?via%3Dihub.

[5] Aleksey Koval. “On Lucas Sequences Computation”. В: *Int. J. Communications, Network and System Sciences* 3 (2010), с. 943—944. URL: https://file.scirp.org/pdf/IJCNS20101200011_90376712.pdf.

[6] Robert Baillie и Jr. Samuel S. Wagstaff. “Lucas Pseudoprimes”. В: *MATHEMATICS OF COMPUTATION* 35.152 (1980), с. 1391—1417. URL: https://www.ams.org/journals/mcom/1980-35-152/S0025-5718-1980-0583518-6/S0025-5718-1980-0583518-6.pdf.