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

Факультет компьютерных наук

Образовательная программа «Компьютерные науки и анализ данных»

Отчет о программном проекте

на тему:

Большие простые числа

Выполнил:

Студент группы БКНАД212 _____ И.И. Блинов

Подпись И.О.Фамилия

26.05.2024

Дата

Принял:

Руководитель проекта Дмитрий Витальевич Трушин

Имя, Отчество, Фамилия

штатный преподаватель, кандидат физико-математических наук

Должность, ученое звание

департамент больших данных и информационного поиска

Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 29.05.2024

_____

Подпись

Москва 2024

Оглавление

1 Введение **5**

1.1 Описание предметной области . . . 5

1.2 Структура работы над проектом . . . 5

1.3 Задачи проекта . . . 5

1.4 Функциональные требования . . . 6

1.5 Нефункциональные требования . . . 6

1.6 Результаты работы . . . 7

2 Обзор литературы **8**

3 Введение в теорию реализованных алгоритмов **9**

3.1 Вероятностные тесты . . . 9

3.1.1 Тест Ферма . . . 9

3.1.2 Тест Эйлера-Якоби . . . 9

3.1.3 Тест Миллера-Рабина . . . 10

3.1.4 Тест Бейли - Вагстаффа - Люка . . . 10

3.1.5 Тест с помощью V-последовательности Люка . . . 11

3.1.6 Тест Бейли - Померанца - Селфриджа - Вагстаффа . . . 11

3.1.7 Усиленный тест Бейли - Померанца - Селфриджа - Вагстаффа . . . 12

3.2 Детерминированные тесты . . . 13

3.2.1 Проверка пробным делением . . . 13

3.2.2 Тест Люка-Лемера . . . 13

3.3 Методы факторизации . . . 14

3.3.1 Метод квадратичного решета . . . 14

4 Детали вычисления последовательностей Люка **17**

5 Архитектура библиотеки **18**

5.1 Общие методы . . . 18

5.1.1 GMPRandomGenerator . . . 18

5.1.2 Интерфейсы для тестов простоты . . . 18

5.1.3 Диаграмма взаимодействия . . . 19

5.2 FileReader . . . 19

5.3 Составляющие библиотеки . . . 19

5.3.1 Вероятностные тесты простоты . . . 20

5.3.2 Детерминированные проверки на простоту чисел . . . 21

5.3.3 Метод квадратичного решета 21

6 Полученные результаты **23**

7 Список литературы **30**

8 Приложения

Аннотация

Существует множество алгоритмов, связанных с простыми числами. В этой работе рассмотрены некоторые из них, относящиеся к проблеме простоты и факторизации чисел. Целью проекта является их изучение и эффективная имплементация на основе изученных материалов, а также сравнение и анализ получившихся методов. Все представленные реализации способны работать с целым числами произвольного размера, используя для этого длинную арифметику.

Ключевые слова

Простые числа, проблема простоты, вероятностные тесты, последовательности Люка, детерминированные тесты, факторизация чисел, C++, GMP

1 Введение

1.1 Описание предметной области

Простые числа активно используются в криптографических алгоритмах. В этой области возникает потребность в быстрой генерации достаточно больших простых чисел, при этом встает вопрос эффективных способов определения простоты числа. В основе же ключевой идеи алгоритма RSA лежит высокая вычислительная сложность факторизации больших чисел, которыми на данный момент считаются числа размера не менее 512 бит [3], при использовании существующих алгоритмов и их имплементаций.

1.2 Структура работы над проектом

Подготовительная часть проекта заключается в изучении общей теории, объясняющей принципы работы рассматриваемых в дальнейшем алгоритмов и обосновывающей их корректность. И, исходя из этого, рассмотрения того, какие алгоритмы существуют в областях определения простоты чисел и их факторизации, то есть разложения в произведение простых множителей. В качестве основного источника литературы для этой части использовалась книга «Factorization and Primality Testing» [4].

Основная часть проекта заключается в имплементации выбранных алгоритмов на языке C++ с использованием библиотеки GMP для предоставления длинной арифметики, необходимой для работы с числами произвольного размера. Для взаимодействия с упомянутой библиотекой используется версия GMPXX для оберток с классами над стандартными методами согласно принципам C++. При написании необходимых методов изучался дополнительный материал в виде статей, посвященных их эффективной реализации. Также основной функционал покрыт автоматическими тестами на основе библиотеки GoogleTest а для сравнения производительности использовалась библиотека Google Benchmark.

Код проекта находится в репозитории Github:
https://github.com/Fraudik/LargePrimeNumbers/tree/dev

1.3 Задачи проекта

- Изучение общей теории и выбор алгоритмов для дальнейшей реализации
- Изучение теории по эффективной имплементации алгоритмов
- Имплементация выбранных вероятностных тестов для проверки на простоту и необходимых условий простоты
- Имплементация выбранных детерминированных тестов проверки на простоту

- Имплементация выбранных алгоритмов факторизации
- Составление автоматических тестов корректности для имплементаций алгоритмов
- Реализация функционала для замера времени работы имплементаций алгоритмов
- Замеры времени работы реализованных алгоритмов и анализ полученных результатов
- Следование указаниям по архитектуре проекта от научного руководителя

1.4 Функциональные требования

- Составление библиотеки, содержащей реализации выбранных алгоритмов по определению простоты чисел и их факторизации
- Составление тестов корректности работы имплементаций алгоритмов
- Сравнение производительности реализованных алгоритмов

В сравнительном анализе предполагается сравнить оценки временных сложностей алгоритмов и замеры времени работы их имлпементаций.

1.5 Нефункциональные требования

- Язык программирования C++ версии стандарта не ниже C++17
- Сторонняя библиотека GMP [1]
- Утилита для автоматической генерации файлов сборки CMake версии не ниже 3.22 и описание сценария сборки в файлах CMAkeLists.txt
- Утилита для автоматизированной сборки make
- Система поддержка версий Git с веб-сервисом Github
- Инструмент для автоматического форматирования кода ClangFormat и руководство по стилю кода в файлах .clang-format и .clang-tidy
- Обработка всех исключений с выводом описания полученной ошибки
- Отсутствие непредвиденных исключений при запуске программы

Под непредвиденными исключениями имеются в виду все исключения, полученные не путем вызова макроса assert из стандартной библиотеки cassert. Эта библиотека позволяет писать проверки, которые будут опущены при сборке проекта с использованием флагов оптимизаций. Режим конфигурации без их использования будет носить название «Debug», а с ними – «Release». Первый вариант используется для тестирования кода, а второй уже для его активного использования и в том числе замерах производительности.

1.6 Результаты работы

Был реализован метод квадратичного решета в классическом изложении из основного источника без дополнительных оптимизаций. Также было рассмотрено и имплементировано широкое разнообразие тестов на простоту, с основным акцентом на тестах с использованием особых последовательностей и оптимизации их вычисления. Были реализованы алгоритм проверки на простоту пробным делением, тесты Люка - Лемера, Ферма, Эйлера - Якоби, Миллера - Рабина, несколько разновидностей тестов Бейли - Вагстаффа - Люка и Бейли - Померанца - Селфриджа - Вагстаффа, а также дополнительный тест, основанный на проверке значений последовательности Люка.

Все методы покрыты тестами корректности, для них подготовлены интерфейсы для унифицированного взаимодействия и класс-обертка для генерации псевдослучайных чисел неограниченного размера. После каждого код-ревью научного руководителя вносились соответствующие правки.

Также было проведено тестирования производительности со следующими результатами: самыми быстрыми оказались вероятностные тесты, не связанные с вычислением последовательности Люка, то есть тесты Ферма, Эйлера - Якоби и Миллера - Рабина, При этом среди них разрыва в скорости выполнения практически нет.

Что касается остальных вероятностных тестов без использования особого подбора параметров — ожидаемо самыми быстрым оказались тесты Бейли - Вагстаффа - Люка с оптимизацией вычислений (описано в разделе 4) с разницей времени выполнения на 30%, как и заявлялось автором оптимизации. Одним из важных и неожиданных выводов является, что усиленные версии тестов на основе последовательностей Люка обладают лучшей производительностью при использовании параметров Селфриджа — при тестировании на числе длиной в 216 тысяч битов (65 тысяч цифр в десятичной системе) разница во времени работы составила 45%! Более того, они оказались на 21% быстрее, чем ускоренные версии обычных тестов. К сожалению, для остальных тестов с использованием последовательностей Люка использование этих параметров не дало такого существенного прироста в скорости. Автор проекта связывает это с особенностями процессов построения последовательностей Люка для теста Бейли - Вагстаффа - Люка и его усиленной формой. В остальном у этих тестов так же практически нет разницы во времени проверки чисел.

Детерминированный тест проверки пробным делением до корня из числа значительно отстает по производительности от вероятностных, как и предполагалось по асимптотикам.

Тесты Бейли - Вагстаффа - Люка и Бейли - Померанца - Селфриджа - Вагстаффа не имеют существенной разницы в производительности, поэтому в качестве одного из выводов проекта предполагается при необходимости практичного и реже всех остальных ошибающегося вероятностного теста использовать именно второй. Если же будут использоваться усиленные тесты Бейли - Вагстаффа - Люка, то для них рекомендуется использование параметров, предложенных Селфриджем, как для уменьшения шанса на оишбку при тестировании числа, так и для улучшенной производи-

тельности.

2 Обзор литературы

Как было ранее упомянуто, общее описание основной части алгоритмов и необходимая теория для них даны в книге «Factorization and Primality Testing». В качестве дополнительных источников на данный момент использовались статьи с определением вероятностных алгоритмов Люка [14] и их эффективной имплементации на псевдокоде [8], [10]. Для реализации вероятностного теста Бейли - Померанца - Селфриджа - Вагстаффа использовалась статья [7], а для его усиленной версии — [13]. Также последняя статья вместе с [16] служили источниками при написании вероятностного теста с использованием V-последовательностей Люка. При написании теста Люка-Лемера использовались дополнительные статьи с его подробным описанием [6] и [5]. Также, для реализации метода квадратичного решета кроме основного источника использовалась статья [18] для улучшения понимания алгоритма.

Относительно возможных аналогов важными особенностями являются объединение алгоритмов в единой библиотеке и построение архитектуры на основе современных стандартов и возможностей выбранного языка программирования.

3 Введение в теорию реализованных алгоритмов

Во всех алгоритмах предполагается, что тестируются натуральные числа больше 1 (которое не является ни простым, ни составным).

Здесь и далее обозначение $\pmod q$ будет означать "по модулю числа q". Выражение $a \equiv r \pmod q$ будет означать равенство классов вычетов a и r по модулю q, то есть "r — это остаток от деления a на x": $a = b \cdot q + r$, где $b \neq 0$ и $0 \leqslant r < |b|$.

В первых трех тестах будет фигурировать число n и параметр a, задающийся вручную. Если условие теста выполняется, то n называется возможным простым относительно базы a.

3.1 Вероятностные тесты

Сначала рассмотрим вероятностные проверки на простоту. Это алгоритмы, которые проверяют необходимые, но не достаточные условия простоты чисел. Их используют перед основными проверками на простоту, так как обычно они обладают меньшей вычислительной сложностью — подробней этот момент будет разобран в части, относящейся к сравнению производительностей имплементаций. Составные числа, которые успешно проходят эти тесты, называются псевдопростыми.

Все тесты, представленные в этом подразделе, имеют вычислительную сложность $\tilde{O}(\log^2(n))$ при использовании перемножения чисел на основе быстрого преобразования Фурье, где $\tilde{O}(g(n))$ в данной работе определим как $O(g(n)\log^k g(n))$ для некоторого k. Это сделано для того, чтобы убрать длинные цепочки выражений из логарифмов и вложенных логарифмов, слабо влияющих на фактический рост времени работы алгоритмов. Используемая библиотека GMP такое умножение поддерживает.

3.1.1 Тест Ферма

Первый такой тест это проверка малой теоремы Ферма:

$$a^{n-1} - 1 \equiv 0 \pmod n$$

где n — проверяемое на простоту числу, $a \in [2; n-2]$ — параметр теста, называемой базой. Также существует бесконечно много чисел Кармайкла, которые удовлетворяют этому условию для всех a [19].

3.1.2 Тест Эйлера-Якоби

Следующим рассмотренным тестом является тест Эйлера-Якоби [15]:

$$a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod n$$

где $\left(\frac{a}{n}\right)$ — символ Якоби. n — проверяемое на простоту числу, $a \in [1; n)$ — параметр теста, называемой базой.

Сивмвол Якоби это произведение символов Лежандра следующего вида:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k}$$

где $p_1, \ldots, p_k$ — различные простые делители числа a. Символ же Лежандра определен следующим образом:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{если } a \text{ это квадратичный вычет по модулю } p \text{ и } a \not\equiv 0 \pmod{p}, \\ -1 & \text{если } a \text{ это квадратный невычет по модулю } p, \\ 0 & \text{если } a \equiv 0 \pmod{p}. \end{cases}$$

Целое число a называется квадратичным вычетом по модулю p, если существует такое $x \in (0; p)$, что $x^2 \equiv a \pmod{m}$. При этом, если такого x не существует, то число a называется квадратичным невычетом по модулю p.

3.1.3 Тест Миллера-Рабина

Ещё одним таким тестом является проверка на простоту Миллера-Рабина [11] Он включает два условия, и для его прохождения число должно удовлетворять одному из них:

$$\begin{cases} a^d \equiv 1 \pmod{n} \\ a^{2^r d} \equiv -1 \pmod{n} \text{ для некоторого } r \in [0; s) \end{cases}$$

где $n > 2$ и нечетно, а $2^s d = n - 1$ при $s, d > 0$ и $d \equiv 1 \pmod{2}$, $a \in [2; n)$ и n взаимно просты.

3.1.4 Тест Бейли - Вагстаффа - Люка

Затем был рассмотрен вероятностный тест Бейли - Вагстаффа - Люка [14]. Пусть есть целые числа $P > 0$, Q и $D = P^2 - 4Q$, а $U_k(P, Q)$ и $V_k(P, Q)$ — последовательности Люка. Тогда тест считается пройденным, если

$$U_{\delta(n)} \equiv 0 \pmod{n}.$$

и n взаимно просто с Q. Здесь n всё так же проверяемое на простоту число, $\delta(n) = n - \left(\frac{D}{n}\right)$.

У этого теста есть и более сильная форма. Если разложить $\sigma(n)$ как $2^s \cdot d$ где d нечетное аналогично тесту Миллера-Рабина и n с D взаимно просты, то эта более сильная форма теста

заключается в проверке того, выполняется ли одно из двух следующих условий:

$$\begin{cases} U_d \equiv 0 \pmod{n} \\ V_{2^r \cdot d} \equiv 0 \pmod{n} \text{ для некоторого } r \in [0; s) \end{cases}$$

Последовательности Люка определены следующим образом:

$$U_0(P,Q) = 0,$$
$$U_1(P,Q) = 1,$$
$$U_n(P,Q) = P \cdot U_{n-1}(P,Q) - Q \cdot U_{n-2}(P,Q) \text{ при } n > 1,$$

и

$$V_0(P,Q) = 2,$$
$$V_1(P,Q) = P,$$
$$V_n(P,Q) = P \cdot V_{n-1}(P,Q) - Q \cdot V_{n-2}(P,Q) \text{ при } n > 1.$$

3.1.5 Тест с помощью V-последовательности Люка

Эту последовательность использует и другой тест. Числа, которые проходят его, носят название псевдопростых чисел Люка-V [13] — от проверямой последовательности, члены которой обозначаются обозначаются как $\{v_i\}$. У них есть и другое название — псевдопростые Диксона второго порядка [16].

Тест заключаются в следующем. Пусть снова есть целые числа $P > 0$, Q и $D = P^2 - 4Q$, а $V_k(P,Q)$ — последовательность Люка, определенная как выше. Проверяемое число n взаимно просто с Q и D, $\delta(n) = n - \left(\frac{D}{n}\right)$, где (D/n) это символ Якоби. Тогда тест считается пройденным, если

$$V_{\delta(n)} \equiv 2Q^{\left(1-\left(\frac{D}{n}\right)\right)/2} \pmod{n}$$

3.1.6 Тест Бейли - Померанца - Селфриджа - Вагстаффа

Был реализован и тест Бейли - Померанца - Селфриджа - Вагстаффа [7], сочетающие несколько других тестов и являющийся доказано детерминированным на числах до 2^{64} [17]. Он сочетает несколько других вероятностных тестов на простоту и состоит из следующих этапов:

1. Проверка, проходит ли число n тест Миллера-Рабина с базой 2. Если не проходит, то алгоритм

на этом заканчивается.

2. Проверка, проходит ли число n тест Бэйли - Вагстаффа - Люка со специально подобранными параметрами D, P, Q.

Для выбора параметров авторами предлагается два варианта. При этом перед этим необходимо проверить, что число n не является квадратом другого числа.

Первый: пусть D это первый элемент в последовательности $\{5, -7, 9, -11, 13, \ldots\}$ для которого $\left(\frac{D}{n}\right) = -1$, а $P = 1$ и $Q = (1 - D)/4$. Полученные параметры в рамках этого проекта называются параметрами Селфриджа, так как именно он предложил этот вариант.

Второй: пусть D это последний элемент в последовательности $\{5, 9, 13, \ldots\}$ для которого $\left(\frac{D}{n}\right) = -1$, а P — наименьшее нечетное число, большее $\sqrt{D}$ и $Q = (P^2 - D)/4$.

На практике этот алгоритм несколько расширяют, заменяя обычный тест Бейли - Вагстаффа - Люка его усиленной формой [17].

3.1.7 Усиленный тест Бейли - Померанца - Селфриджа - Вагстаффа

Последний рассмотренный вероятностный тест это усиленная версия предыдущего [13]. Его авторы предлагают немного изменить выбор D, P, Q, а также добавить несколько дополнительных проверок и тест на значение элемента V-последовательности Люка, который проходит сравнительно мало составных чисел — всего 5 для чисел со значениями меньше 10^15.

Итоговый тест следующий:

1. Проверка, проходит ли число n тест Миллера-Рабина с базой 2. Если не проходит, то алгоритм на этом заканчивается.

2. Проверка, проходит ли число n усиленный тест Бэйли - Вагстаффа - Люка со специально подобранными параметрами D, P, Q. Отбор параметров происходит как в первом варианте, при этом, если получили $Q = -1$, то устанавливаем параметры Q и P равными 5. Значение параметра $D = P^2 - 4Q$ остается тем же. Утверждается, что с такими параметрами существует меньше составных чисел, проходящий этот тест.

3. Проверка, проходит ли число n тест с помощью V-последовательности Люка с теми же параметрами P, Q, D.

4. Проверка критерия Эйлера: выполняется ли равенство $Q^{(n+1)/2} = Q \cdot (Q/n) \pmod{n}$, где (Q/n) — символ Лежандра.

3.2 Детерминированные тесты

Теперь перейдем к детерминированным проверкам на простоту — такие алгоритмы гарантированно определяют, является ли проверяемое число простым или составным.

3.2.1 Проверка пробным делением

Первым таким тестом является проверка делимости проверяемого числа на возможные делители кроме единицы вплоть до заданной верхней границы. При значении этой границы, равном квадратному корню проверяемого числа, округленному до целого в меньшую сторону, этот тест гарантированно определяет простоту числа. Это следует из того соображения, что, если a делитель проверяемого числа p, то либо $b = p/a$, либо a меньше или равен $\sqrt{p}$. В противном случае $a \cdot b > p$. Вычислительная асимптотика такого алгоритма составляет $O(\sqrt{n})$ соответственно.

Также для этого теста существует оптимизация, при которой сначала число отдельно проверяется на четность, а затем перебираются только нечетные значения - в этом случае проверяется не $\lfloor\sqrt{n}\rfloor - 1$ значений, а $\lfloor(\lfloor\sqrt{n}\rfloor - 1)/2\rfloor$ значений. Здесь итоговое значение округляется вниз, так как если оно нечетное, то предыдущее было четным и четных чисел до этого значения на одно больше, чем нечетных, без учета единицы.

3.2.2 Тест Люка-Лемера

Другим имплементированным алгоритмом является тест Люка-Лемера [5], [6]. Он работает только с числами вида $M_N = 2^N - 1$, где $N \in \mathbb{N}$, которые также называются числами Мерсенна.

Определим последовательность $\{s_i\}$ при $i \geq 0$ следующим образом:

$$s_i = \begin{cases} 4 & \text{при } i = 0; \\ s_{i-1}^2 - 2 & \text{при } i > 0 \end{cases}$$

Число Мерсенна M_N является простым тогда и только тогда, когда выполняется равенство

$$s_{N-2} \equiv 0 \pmod{M_N}$$

Кроме того, можно перед самим тестом осуществить дополнительную проверку, позволяющую часть чисел Мерсенна сразу отнести к составным. Для этого нужно проверить простоту N — утверждается, что если оно составное, то и M_N тоже. У этого утверждения есть простое доказательство: если N составное, то $N = a \cdot b; a, b > 1$. Тогда:

$$M_N = 2^N - 1 = 2^{ab} - 1 = (2^a)^b - 1 = (2^a - 1)((2^a)^{b-1} + (2^a)^{b-2} + \cdots + 2^a + 1)$$

То есть M_N раскладывается в произведение двух множителей и является составным числом.

Вычислительная сложность этого теста составляет $\tilde{O}(\log^2(n))$. Это самый быстрый известный на данный момент детерминированный тест на проверку простоты, других детерминированных тестов с такой же асимптотикой также не существует. По этой причине именно он лежит в основе проекта по поиску новых простых чисел Мерсенна: «Great Internet Mersenne Prime Search» [2].

3.3 Методы факторизации

Последний раздел относится к задаче о нахождении делителей числа или определении его как простого. Из таких методов был реализован только один: метод квадратичного решета.

3.3.1 Метод квадратичного решета

Алгоритм факторизации числа n этим методом состоит из трех общих шагов [4]:

1. Находим некоторую базу факторов: список предпосчитанных простых чисел в заранее заданном количестве N. Для каждого числа p из нее решаем уравнения $x^2 \equiv n \pmod{p}$

2. Выполняем само просеивание решетом, чтобы в дальнейшем получить кандидатов для проверки на третьем шаге с помощью метода Диксона.

3. Дальше с помощью отсеянных кандидатов ищем такую пару x, y, что $x^2 \equiv y^2 \pmod{n}$ при $x \not\equiv y \pmod{n}$ и $x \not\equiv -y \pmod{n}$. Утверждается, что тогда НОД(n, $x - y$) и НОД (n, $x + y$) будут делителями n (не равными 1 и самому n). При нахождении таких делителей останавливаем алгоритм.

Рассмотрим следующие шаги более детально. Описанный алгоритм приводится в том же источнике [4] и является версией метода квадратичного решета с оптимизацией Роберта Сильвермана.

1. Находим N простых числа по порядку, начиная с 2 включительно, таких, что их с проверяемым числом символ Лежандра равен 1. Назовем упорядоченное множество этих чисел FB (factor base).

2. Инициализируем массив для этапа просеивания (решето) нулями от $-M$ до M, где M — некоторая заранее заданная константа.

3. Решаем уравнения $x^2 \equiv p \pmod{n}\ \forall p \in FB$, где n — факторизуемое число. Обозначим решения этого уравнения для p за R_p.

4. Для всех корней в R_p прибавляем $\log(p)$ к элементам решетам с индексами $r_i, r_i + p, \ldots : r_i \in R_p$ в пределах размера массива.

5. Запись с индексом i в решете отождествляем с числом $\lfloor\sqrt{n}\rfloor+i$. Вычисляем некоторое значение порога для отсеивания чисел как $\log(n)/2 + \log(M) - T \cdot \log(p_{max})$, где T — некоторая заранее заданная константа, p_{max} — максимальное простое число из FB. Отбираем те числа, которые в итоге оказались больше этого значения порога.

6. Теперь начинается завершается шаг с алгоритмом факторизацией Диксона. Для этих чисел формируем разложения на простые делители с помощью перебора всех простых из FB. Составляем матрицу из векторов, где каждый вектор соотносится с одним из этих чисел. В нем на i-м месте будет стоять 1, если в разложении этого числа на простые множители p_i находится в нечетной степени, и 0 иначе.

7. Выполняем преобразование Гаусса над этой матрицей. Теперь те векторы, для которых первые $2 \times M$ элементов равны 0, а остальные ненулевые, отображают полные квадраты (числа, корень из которых равен какому-то целому числу). Для такого вектора составляем уравнение $x^2 \equiv y^2 \pmod{n}$, где с одной стороны находится произведением $(\lfloor\sqrt{n}\rfloor+p_i)^2$ для всех i, для которых элементы в векторе с этими индексами равным 1, а с другой — произведение p_i^2 по тем же правилам.

8. Вычисляем НОД n и $x - y$ (или $x + y$). Если он не равен 1 или n, то мы нашли искомый делитель n. Иначе переходим к следующему вектору, для которого выполнены те же условия, и так до тех пор, пока не найдем делитель или не рассмотрим все такие векторы.

Отдельно стоит отметить упомянутые параметры N, M, T. Они задаются вручную до запуска алгоритма и регулируются под размер факторизуемых чисел.

В 3 шаге для решения квадратичных вычетов имплементирован алгоритм Лехмера, описанном в том же источнике (стр. 108, глава 8.3). Алгоритм работает следующим образом:

1. Генерируем случайным образом число h и вычисляем $h^2 - 4n \pmod{p}$. Если символ Лежандра получившегося выражения и p равен -1, то переходим к следующему шагу. Иначе повторяем до тех пор, пока это не выполнится.

2. Определим последовательность $\{v_i\}$ через рекурсию:

$$v_i = \begin{cases} h & n = 1 \\ h^2 - 2n & n = 2 \\ h \times v_{i-1} - n \times v_{i-2} & n > 1 \end{cases}$$

3. Тогда $v_{2i} = v_i^2 - 2n^i$ и $v_{2i+1} = v_i \times v_{i+1}$ и решение искомого уравнения:

$$x = v_{(p+1)/2} \times \frac{p+1}{2} \pmod{p}$$

Согласно анализу Карла Померанца [12] предполагаемая вычислительная сложность такого алгоритма квадратичного решета составляет $O(e^{\sqrt{\ln(n)\ln(\ln(n))}})$.

4 Детали вычисления последовательностей Люка

Сами последовательности Люка и их свойства описаны в [4], [14]. На основе этого можно написать их рекурсивное вычисление или более быстрое. Так, авторы [10] в рамках эффективного построения криптографических подписей предложили оптимизированный алгоритм вычисления этих последовательностей. Этот алгоритм используется в реализованной библиотеке при вычислении последовательной Люка для теста Бейли - Померанца - Селфриджа - Вагстаффа, так как в нем требуются дополнительные промежуточные проверки на значения разных членов последовательности V. Также алгоритм для этого теста реализовывался с учетом рекомендаций его авторов. Эти рекомендации включают оптимизации за счет переиспользования ранее полученных результатов вычислений и добавление дополнительных проверок с использованием этих же результатов.

Для обычной формы теста реализована более эффективная версия этого алгоритма из посвященной оптимизации вычисления последовательности статьи [8]. Она же используется в реализации теста с проверкой значения одного элемента из V-последовательности Люка. Для обоих тестов оставлены и реализации с предыдущей версией алгоритма вычисления последовательностей Люка для сравнительного анализа производительности.

5 Архитектура библиотеки

В рамках проекта реализована библиотека с реализацией описанных выше алгоритмов, а также тесты и утилиты для сопоставительного анализа имплементаций по времени работы. Содержательная часть библиотеки размещена в трех директориях, которые носят названия «deterministic_tests», «probable_prime_tests», «quadratic_sieve_factorization», и в них находятся, соответственно, методы для детерминированных тестов на простоту, вероятностных тестов на простоту и реализация Метода квадратичного решета. Кроме них выделены и общие методы, вынесенные на верхний уровень.

5.1 Общие методы

Из общих методов стоит выделить генератор больших псевдослучайных чисел «GMPRandomGenerator», интерфейсы для унифицированного обращения к методам определения простоты чисел и класс для чтения из файлов «FileReader».

5.1.1 GMPRandomGenerator

Генератор псевдослучайных чисел с использованием сторонней библиотеки GMP. Выдает числа в заданном диапазоне, обычно от нуля до значения проверяемого числа минус один. Также он позволяет задать число в качестве зерна, обеспечивая возможность воспроизводимости результатов его работы при одинаковых числах. Генератор основан на алгоритме Вихря Мерсенна и имеет равномерное распределение [9].

В самой библиотеке используется в двух местах: при генерации случайных баз для вероятностных тестов на простоту и в алгоритме факторизации с помощью метода квадратичного решета, в ходе вычисления квадратичных вычетов.

5.1.2 Интерфейсы для тестов простоты

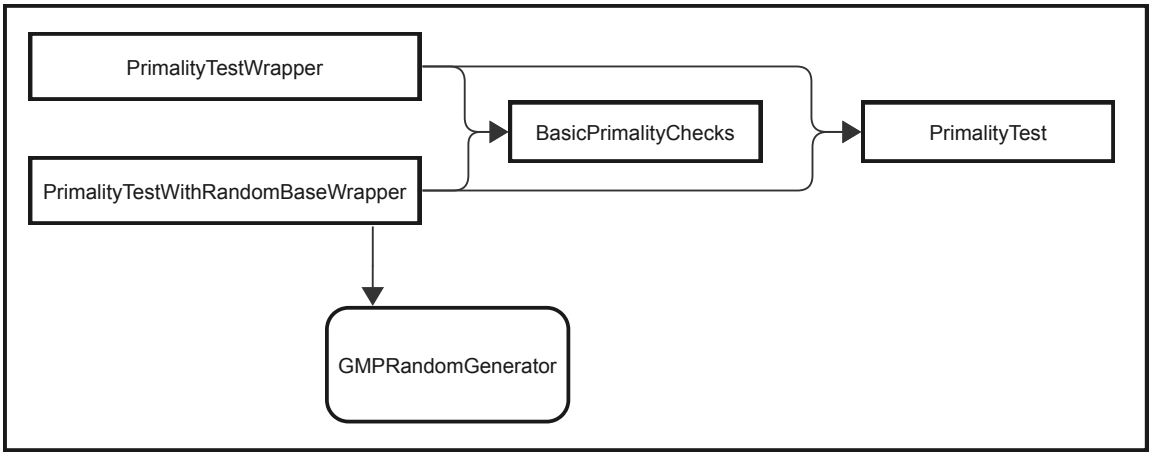
Всего написано два таких интерфейса с использованием шаблонов. Первый для тестов с произвольными дополнительными аргументами, к примеру базой, или вовсе без них. Он носит название «PrimalityTestWrapper» Второй для тестов с использованием случайно сгенерированной базой в диапазоне от нуля до проверяемого числа минус один – он также предоставляет возможность передать зерно для генератора псевдослучайных чисел и носит название «PrimalityTestWithRandomBaseWrapper».

Кроме того, в обоих интерфейсах перед запусками самих переданных тестов осуществляются минимальные проверки на простоту: ручная обработка случаев числа, равного 0, 1 или 2; проверка на четность числа и проверка на то, является ли число полным квадратом. Это сделано потому, что для некоторых тестов на простоту требуется, чтобы проверяемые числа проходили все или часть из этих тестов. Также эти проверки рекомендуются перед осуществлением всех остальных,

более вычислительно затратных тестов, позволяя сразу определить часть составных чисел. Метод, осуществляющий эти проверки, называется «BasicPrimalityChecks».

5.1.3 Диаграмма взаимодействия

Структура взаимодействий тестов, интерфейсов и генератора выглядит следующим образом, с самим тестом на простоту «PrimalityTest».



5.2 FileReader

Также было написан класс для чтения из произвольного текстового файла в контейнер типа вектор. Класс отвечает за открытие и закрытие потоков на чтение из файла по предоставленному пути, десериализацию данных из файла в контейнер и предназначен для скрытия деталей чтения из файла при реализации требующих этого методов в целях соблюдения разграничения уровней абстракции. Для корректного считывания файл должен иметь следующий формат: на первой строке содержать последующее число элементов в нем, а во всех остальных сами элементы, каждый на новой строке.

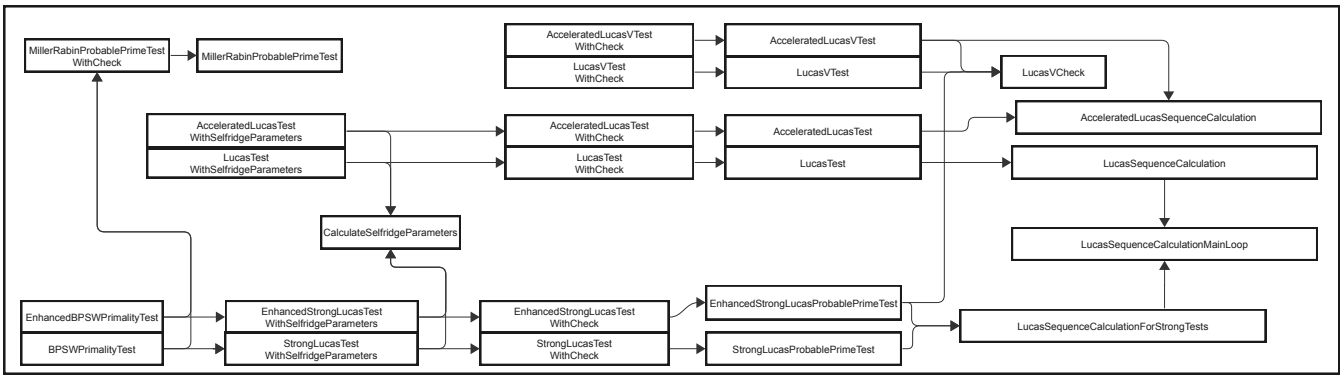
5.3 Составляющие библиотеки

Сами тесты и имплементации алгоритмов находятся во вложенных директориях, описанных выше. Существует некоторые общие принципы их объявления и наименования. У тестов, которые предЪявляют некоторые требования к подаваемым числам и аргументам выделены отдельные методы с окончанием «WithCheck» и именно они предоставляются внешнему пользователю библиотеки. В них используется проверки из стандартной библиотеки C++ <cassert», позволяющие не осуществлять эти проверки в сборках с использованием оптимизаций производительности.

5.3.1 Вероятностные тесты простоты

Тесты Ферма, Эйлера-Якоби, Миллера-Рабина реализованы в точности так, как описаны, и каждый состоят из двух методов: имплементации самого теста в одном методе и вызове этого метода с assert на пререквезиты теста, например, значения базы, в другом. Второй метод в конце названия имеет приставку WithCheck и предполагается, что пользователь библиотеки будет вызывать именно его.

В остальных тестах необходимо вычисление последовательностей Люка, в рамках которого реализовано несколько методов. Структура их взаимодействий выглядит следующим образом:



Увеличенная диаграмма представлена в конце отчета.

При наименованиях в библиотеке тест Бейли - Померанца - Селфриджа - Вагстафф сокращается до БПСВ, а тесты Бейли - Вагстаффа - Люка — до тестов Люка, своих общепринятых сокращений.

Методы с окончанием WithSelfridgeParameters используют параметры P, Q, полученные с помощью алгоритма Селфирджа, описанного в шаге 2 раздела 3.1.6 отчета, с оптимизацией из шага 2 раздела 3.1.7. Сам этот алгоритм реализован в методе <CalculateSelfridgeParameters».

Методы с приставкой Enhanced реализовывают тест, описанный в разделе 3.1.7 отчета. Основное отличие от обычной версии находится в методе «EnhancedStrongLucasProbablePrimeTest», где кроме самого усиленного теста Люка реализован еще и 4 шаг упомянутого теста (проверка Критерия Эйлера), а также 3 шаг как вызов метода «LucasVCheck» с уже вычисленным элементом V-последовательности Люка.

Методы с приставкой Accelerated реализовывают оптимизированное вычисление последовательной Люка, описанное в 4 разделе отчета. В рамках вычисления последовательностей Люка для неоптимизированной версии и усиленных тестов выделен отдельный метод «LucasSequenceCalculationMain
содержащий основной цикл вычисления.

5.3.2 Детерминированные проверки на простоту чисел

Реализованы оба из описанных в теоретической части теста: проверка пробным делением и тест Люка-Лемера. В реализации первого реализована оптимизация по отдельной проверке на четность и дальнейшем переборе только нечетных числе, а также написано два метода. В первом верхней границей является квадратный корень из проверяемого числа, а во втором ее можно задать отдельно, чтобы можно было быстро проверить несколько небольших возможных делителей перед запуском вычислительно сложных тестов. Для второго реализован только один метод с проверкой на то, что его запускают именно с числом Мерсенна.

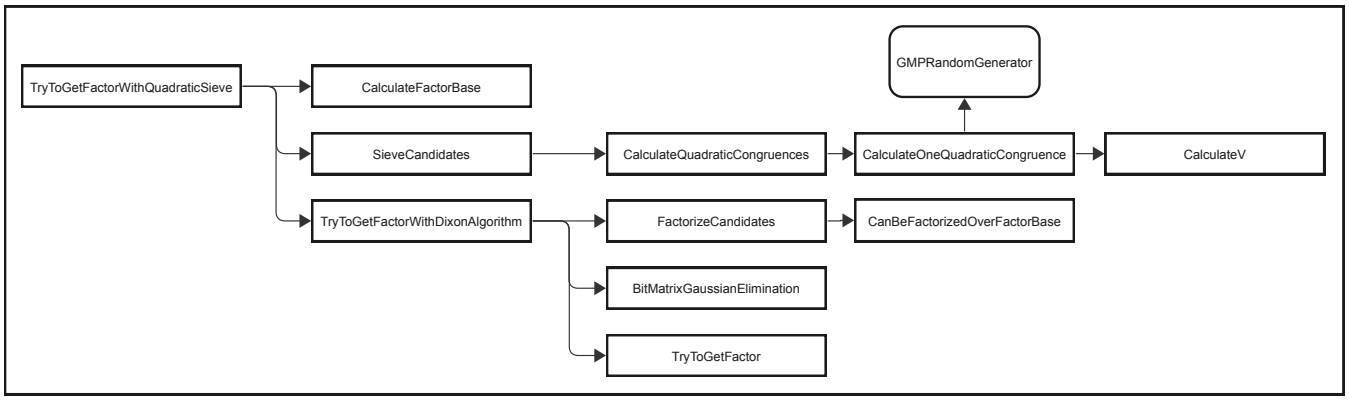
5.3.3 Метод квадратичного решета

Алгоритм реализован в соответствии с его описанием в [4], дополнительные оптимизации не рассматривались.

Для вычисления фактор базы в начале алгоритма используется алгоритм решета Эратосфена. При этом заранее известно k-ое простое число и при формировании решета Эратосфена проверяется, хватит ли размера решета равном этому простому числу для получения требуемых N простых чисел. Если же нет, то размер решета оценивается как $n \cdot \log(n) \cdot P$, где P — некоторый коэффициент. Это следует из теоремы о распределении простых чисел. Также согласно основному источнику проекта простое число примерно с 50% вероятностью будет удовлетворять условию на значение символа Лежандра.

Из этих соображений на момент написания отчета коэффициент P был взят равным 100. Известное k-ое равно 99991, это 9590-ое простое число. В основном источнике указываются подобранные оптимальные значения параметров для факторизации чисел разного размера. Максимальное число рассмотренных цифр в числе равно 66, в этом случае использовался размер фактор базы в 4500 чисел. Поэтому предпосчитанного k-го простого числа должно хватить для всех стандартных случаев использования библиотеки.

Структура взаимодействий в имплементации алгоритма устроена следующим образом:



Она также приложена в конце отчета.

Здесь «SieveCandidates» включает в себя шаги с 2 по 5 из теоретического описания, «TryToGetFactor
— с 6 по 8, где метод «TryToGetFactor» является реализацией последнего шага алгоритма.

Метод «CalculateQuadraticCongruences» с последующими содержат реализацию алгоритм Лемера, где метод «CalculateOneQuadraticCongruence» включает в себя 1 и 3 шаги, а метод «CalculateV» —- получение требуемого для 3 шага значения элемента последовательности через ее рекурсивное вычисление, то есть 2 шаг этого алгоритма.

6 Полученные результаты

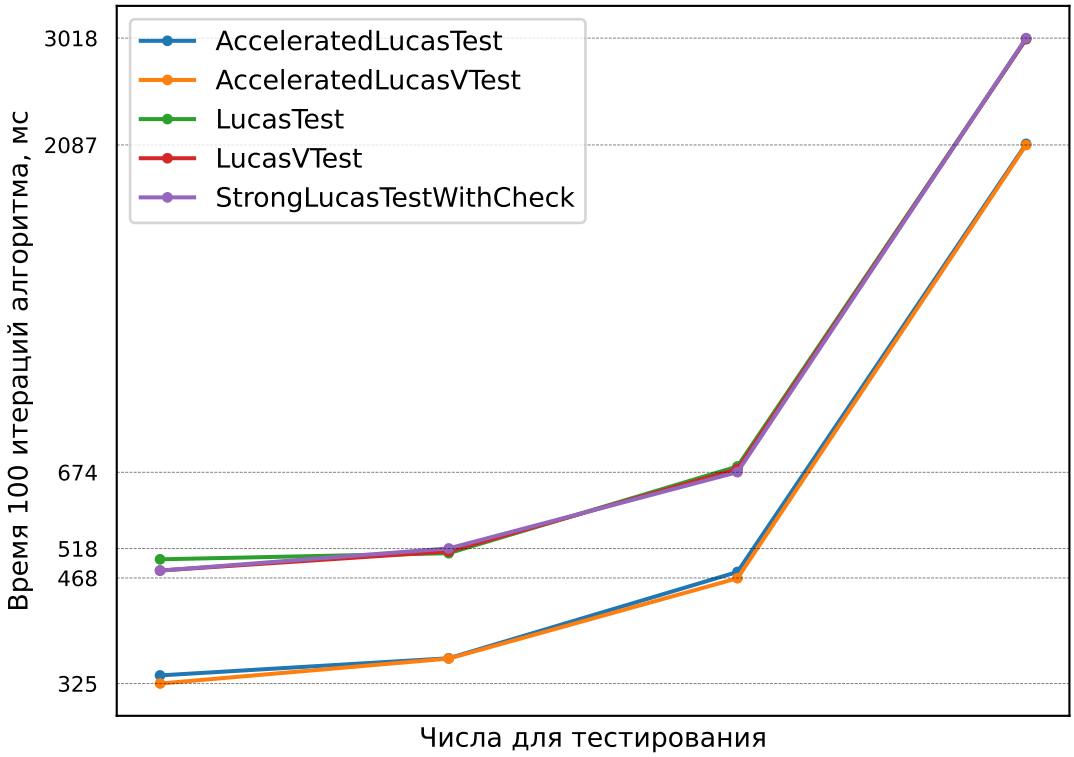
Для тестирования использовалась библиотека Google Benchmark. Тестирование проводилось на компьютере с процессором i7-12800HX и размером оперативной памяти в 32ГБ с DDR5. Использовался компилятор GNU C++ в WSL (Windows Subsystem for Linux).

В качестве чисел для тестирования использовались числа из списка известных самых больших чисел. В силу того, что самый эффективного проект по их поиску это GIMPS благодаря тесту Лукса-Лемера, все рассмотренные числа являются числами Мерсенна. А именно были рассмотрены числа M_{607}, M_{1279}, M_{2203}, M_{2281}, M_{3217}, M_{4423}, M_{9689} , M_{9941}, M_{11213}, M_{19937}. Здесь M_p обозначает p число Мерсенна, то есть $2^p - 1$. Длина этих чисел варьируется от 183 до 6‘002 цифр в десятичной системе. Для построения самих графиков по результатам замеров использовался скрипт на языке Python с использованием библиотеки matplotlib.

Немного об особенностях представления графиков. Во-первых, большинство сравнительных замеров производительности проводились на 4 самых больших числах, из-за отсутствия ощутимой разницы на первых 6. Во-вторых, было принято решение убрать подписи с оси абсцисс в виду их неинформативности —- над числами проводилось кодирование и перевод в их порядковые номера по возрастанию из-за слишком больших отличий в размере. В-третьих, все графики имеют логарифмическую шкалу по оси ординат. В-четвертых, все алгоритмы запускались на каждом числе по 100 раз, и в качестве итогового числа времени работы бралось среднее результатов этих запусков.

Также далее будем использовать систему сокращений, используемой в самой библиотеке, для удобства чтения и консистентности с наименованиями в самой библиотеке.

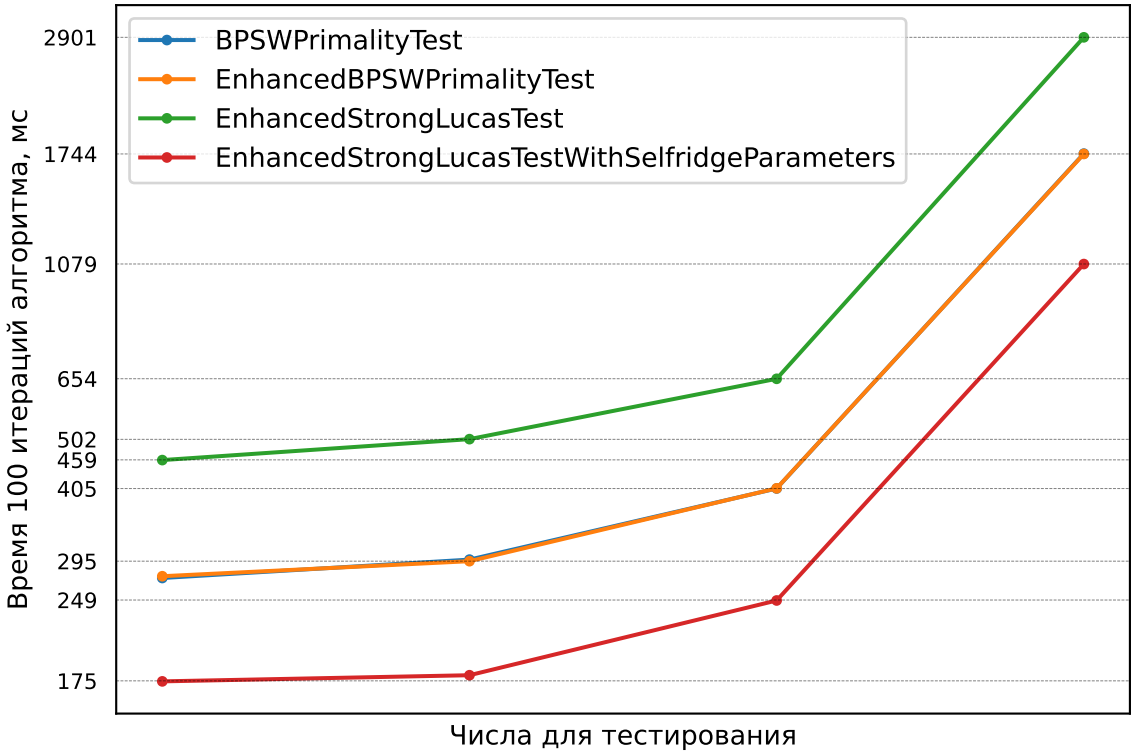
Теперь к самим результатам. Сначала проводились тесты, основанные на последовательностях Люка, кроме тестов БПСВ и усиленного теста Люка. Реализации с использованием ускоренного вычисления последовательности Люка действительно показали лучшее время исполнения. Ожидаемо тесты с использованием последовательности V и обычного теста Люка не отличаются временем исполнения, но кроме того усиленная версия последнего имеет ту же скорость исполнения, как и обычная. Тестирование проводилось на 4 самых больших чисел в выборке из-за отсутствия разницы на предыдущих 6. При запуске для всех алгоритмов P и Q был равны 4 и 2 соответственно. График данного теста производительности представлен ниже:



Также дополнительно было проведено сравнение базовой реализации обычной формы теста Люка и ее ускоренной версии на числе M_{216091} длиной в 65‘050 цифр. Всего было они были запущены порядка 10 раз и результаты следующие: для базовой реализации время проверки на простоту составило в среднем 776 секунд, для ускоренной — 542. Можно заметить, что второй алгоритм отработал быстрее на 30%, как и заявлялось в работе автора, предложившим используемую в ускоренной версии оптимизацию [8].

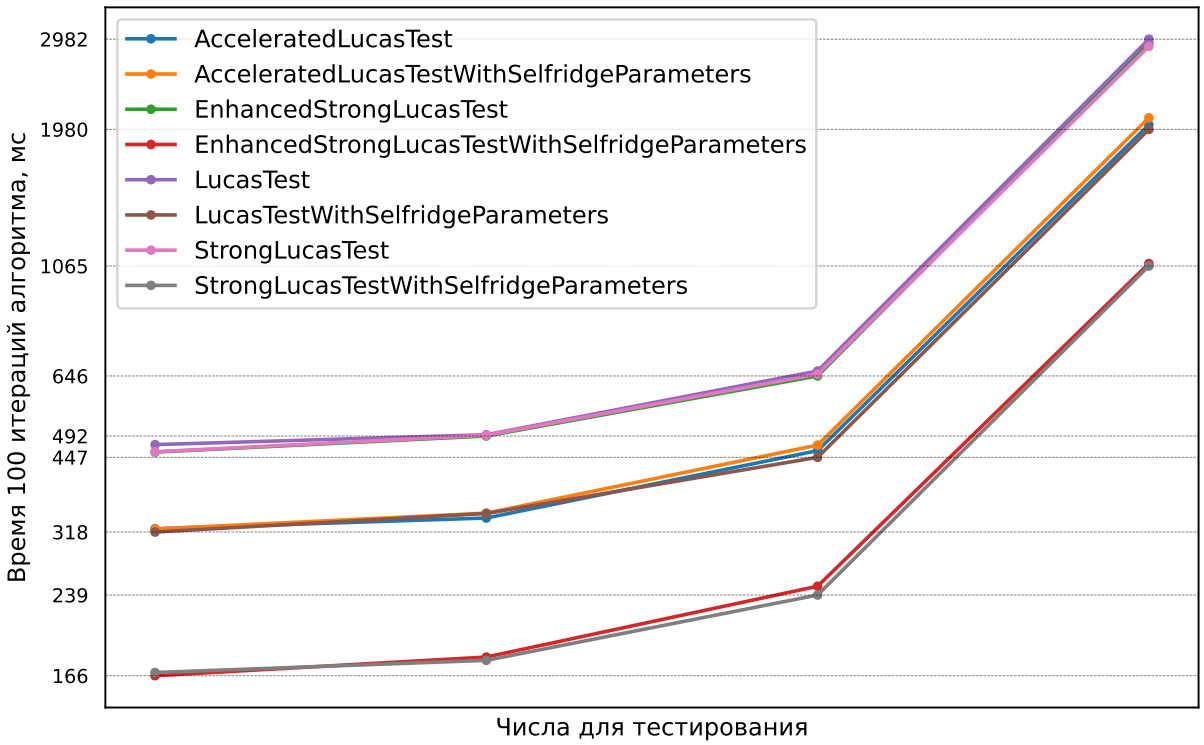
Затем были проведены сравнительные тесты БПСВ и его усиленной формы. Выяснилось, что их имплементации обладают практически одинаковой скоростью исполнения. Для дополнительного сравнения с ними тестировалась версия усиленного теста Люка из статьи об усиленной форме БПСВ [13]. Ожидалось, что она будет иметь меньшее время исполнения на тех же числах — однако был получен ровно противоположный результат, несмотря на осуществления дополнительного теста Миллера - Рабина в тестах БПСВ. Автор данного проекта связывает это с тем, что в тестах БПСВ используется особый подбор параметров PP и Q по методу Селфриджа, и это влияет не только на количество ложноположительных результатов при тестировании чисел на простоту, но и на время исполнения. Для проверки этого данный сравнительный тест был перегенерирован с включением еще одного алгоритма: все той же описанной версии усиленного теста Люка, но с использованием этих же параметров. В этот раз результаты совпали с ожиданиями, и этот добавленный алгоритм

показал себя быстрее остальных в данном сравнении:



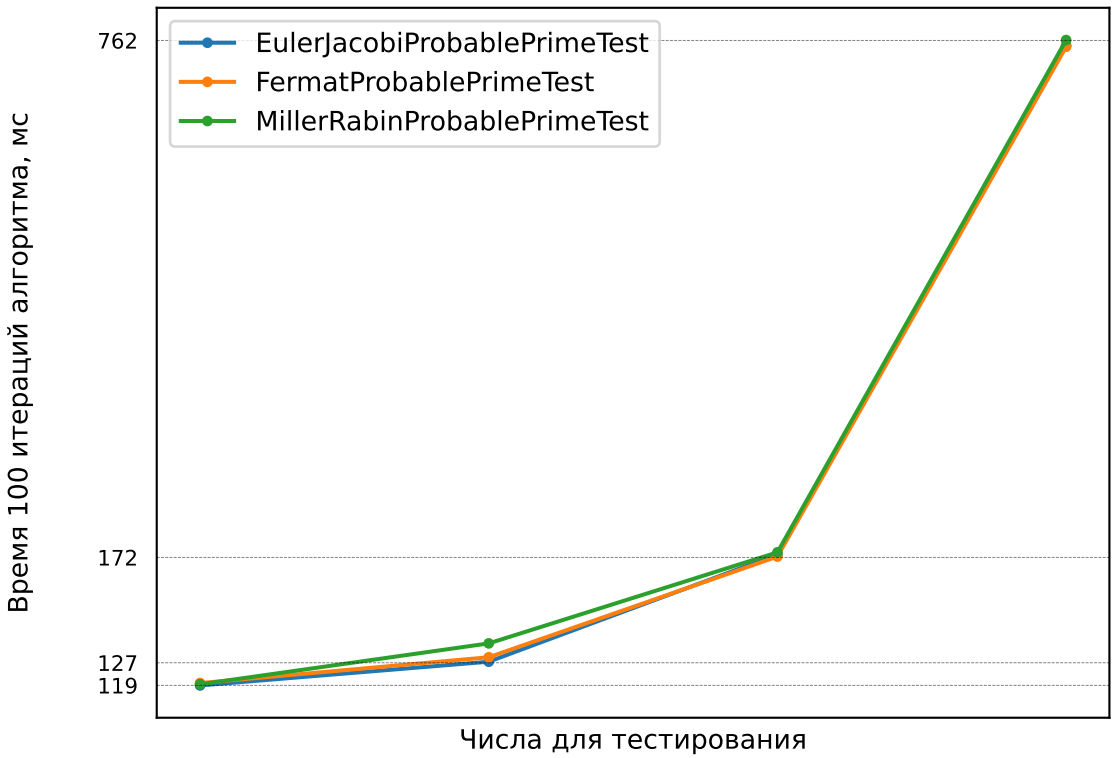
10 итераций этих алгоритмов на числе M_{216091} показали следующие результаты: обычный тест БПСВ и усиленный показали среднее время исполнения в 626 и 636 секунд соответственно, обновленная версия сильного теста Люка и эта же версия с использованием параметров Селфриджа — 775 и 429 секунд соответственно — то есть разница составила целых 45% процентов!

Тогда были проведены сравнительные тесты обычных тестов и их же, но с использованием параметров Селфриджа. Оказалось, что такой прирост в скорости работы проявляется только у усиленных форм теста Люка, так как у них отличаются построения последовательностей Люка — в особенности границы итераций. Однако благодаря этому сравнению выяснилось, что усиленные тесты Люка с использованием этих параметров показывают даже лучшие результаты по производительности, чем ускоренная версия обычного теста Люка:

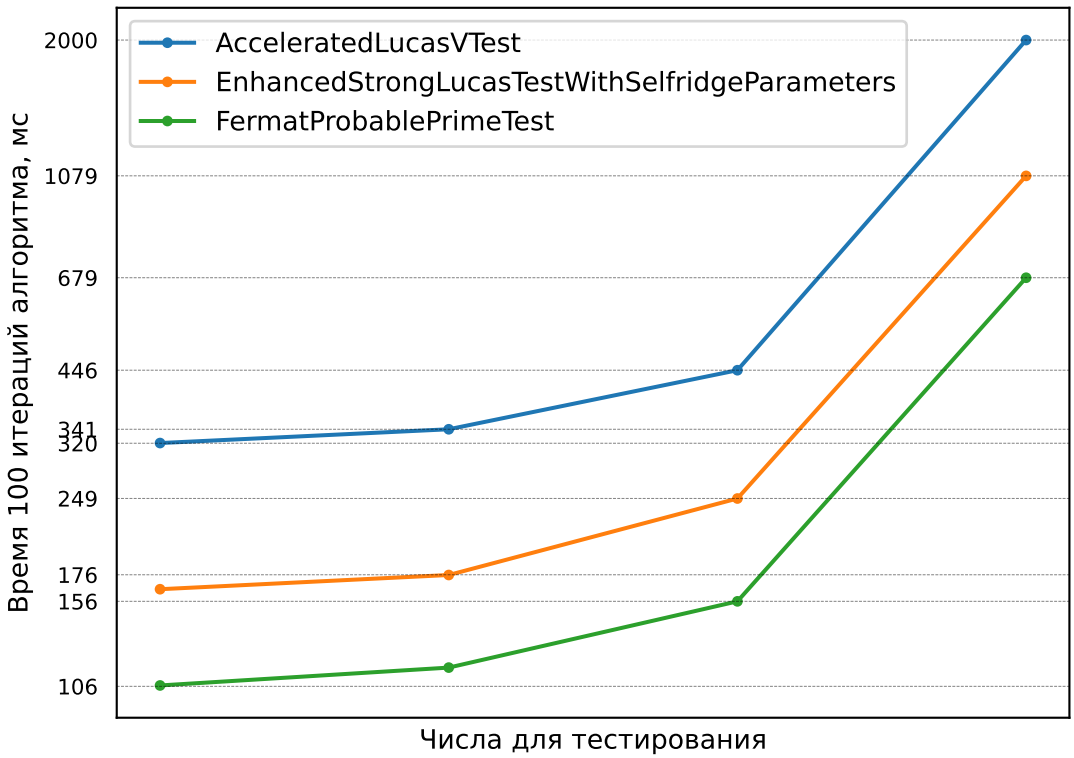


При сравнении на все том же числе M_{216091} усиленные тесты с параметрами Селфриджа показали практически один и тот же результат в среднем — 420 секунд, а ускоренная версия стандартного теста — 531 секунду, то есть на 21% хуже.

После этого были проведены тесты производительности общих вероятностных тестов без использования последовательностей Люка. Все три теста: Ферма, Эйлера - Якоби и Миллера - Рабина не имели значительных различий во времени выполнения:



При сравнении общих вероятностных тестов и тестов Люка был выбран тест Ферма и два самых быстрых алгоритма из тестов Люка. Имплементация теста Ферма оказалась более производительной в плане времени выполнения, как это видно на графике:



Было проведено и сравнение проверки простоты числа с помощью пробного деления до корня из числа и усиленная версия теста BPSW, как вероятностный тест с наименьшим числом ложноположительных результатов. В итоге детерменированный тест ожидаемо отработал сильно дольше вероятностного. Для этого сравнения были выбраны очень небольшие числа, сравнительно чисел для остальных сравнений: 6700417, 2147483647, 999999000001 и 67280421310721. Это было сделано из-за очень низкой сравнительной скорости работы алгоритма пробного деления. Из-за слишком большого разрыва во времени работы осмысленного графика не получилось: на последнем числе детерминированный тест отработал за 101 миллисекунду, а вероятностный — за $0,007$ миллисекунд.

Сравнение с факторизацией квадратичным решетом в таком формате не имеют смысла, так как время работы этого алгоритма больше зависит от задаваемых при запуске параметров, чем от величины самого числа. Кроме того, в отличие от остальных тестов для его работы требуется и дополнительная память, которой на тестирующем устройстве не хватало. Вручную было проведено сравнение с алгоритмом проверки пробным делением до корня и усиленным БПСВ тестом: ожидаемо алгоритм квадратичного решета может определять простоту быстрее алгоритма пробного деления,

а усиленный тест БПСВ всегда работает быстрее.

7 Список литературы

[1] Gnu multi-precision library. https://gmplib.org/. (Accessed on 28/05/2024).

[2] Great internet mersenne prime search: The math. https://www.mersenne.org/various/math.php. (Accessed on 28/05/2024).

[3] Size considerations for public and private keys. https://www.ibm.com/docs/en/zos/2.3.0?topic=certificates-size-considerations-public-private-keys. (Accessed on 28/05/2024).

[4] Дэвид Брессуд. Factorization and primality testing. издательство Springer New York, 1989.

[5] Джон Джарома. Note on the lucas–lehmer test. https://www.irishmathsoc.org/bull54/M5402.pdf, октябрь 2004. (Accessed on 28/05/2024).

[6] Ана Виллегас Санабрия Дэниел Кортилд. Lucas-lehmer primality test. https://www.researchgate.net/publication/358646149_Lucas-Lehmer_Primality_Test, февраль 2022. (Accessed on 28/05/2024).

[7] Сэмюэл Стэндфилд Вагстафф-младший Карл Померанс, Джон Селфридж. The pseudoprimes to $25 \cdot 10^9$. Mathematics of computation, сентябрь 1980.

[8] Алексей Коваль. On lucas sequences computation. Int. J. Communications, Network and System Sciences, ноябрь 2010.

[9] Такудзи Нисимура Макото Мацумото. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation, январь 1998.

[10] Жан-Жак Кискуатер Марк Джой. Efficient computation of full lucas sequences. Electronics Letters 32(6):537–538, март 1996.

[11] Рабин Михаэль. Probabilistic algorithm for testing primality. Journal of Number Theory, февраль 1980.

[12] Карл Померанц. Smooth numbers and the quadratic sieve. Algorithmic Number Theory, 2008.

[13] Сэмюэл Стэндфилд Вагстафф-младший Робер Бэйди, Андрей Фиори. Strengthening the baillie-psw primality test. https://arxiv.org/abs/2006.14425. (Accessed on 28/05/2024).

[14] Сэмюэл Стэндфилд Вагстафф-младший Роберт Бэйли. Lucas pseudoprimes. Mathematics of computation, октябрь 1980.

[15] Фолькер Штрассен Роберт Соловей. A fast monte carlo probabilistic algorithm for primality test. SIAM Journal on Computing, март 1977.

[16] Анджей Роткевич. Lucas pseudoprimes. Functiones et Approximatio Commentarii Mathematici, декабрь 2000.

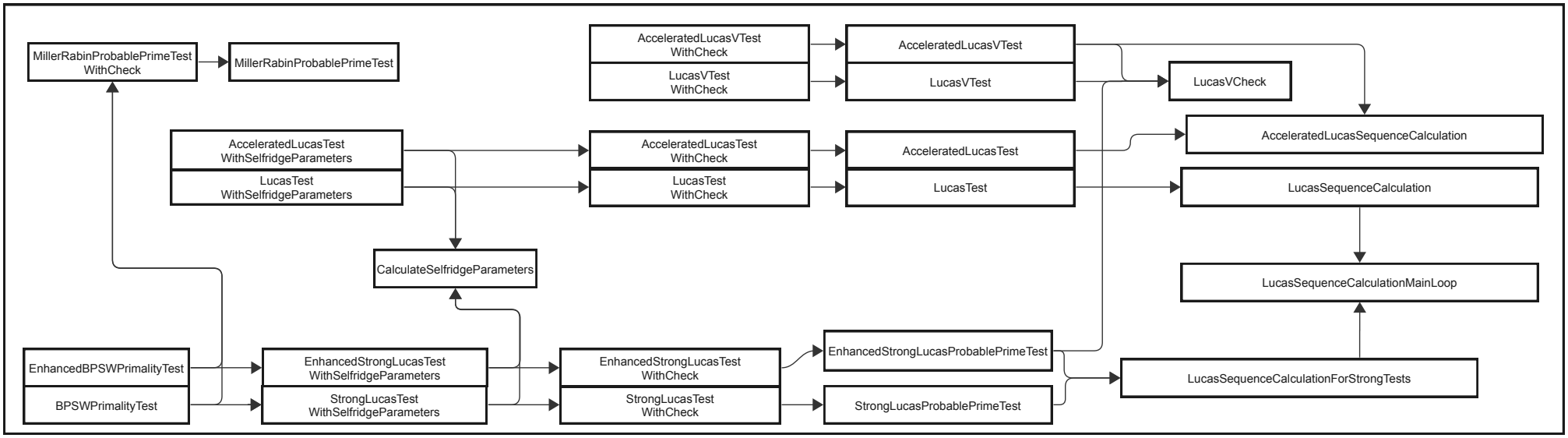
[17] Томас Рэй. The baillie-psw primality test. https://web.archive.org/web/20191121062007/http://www.trnicely.net/misc/bpsw.html. (Accessed on 28/05/2024).

[18] Роберт Сильверман. The multiple polynomial quadratic sieve. Mathematics of Computation, январь 1987.

[19] Эндрю Грэнвилл и Карл Померанс У. Р. Алфорд. There are infinitely many carmichael numbers. Annals of Mathematics, август 1992.

8 Приложения

«Структура взаимодействия методов в имплементации тестов с вычислением последовательностей Лукаса»



«Структура взаимодействия методов в имплементации метода квадратичного решета»