Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования «Казанский (Приволжский) федеральный университет»
Институт вычислительной математики и информационных технологий

Кафедра системного анализа и информационных технологий

Направление подготовки: 10.03.01 — Информационная безопасность
Профиль: Безопасность компьютерных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РЕАЛИЗАЦИЯ КОМБИНИРОВАННОГО МЕТОДА ПОИСКА ПРОСТЫХ ЧИСЕЛ

Обучающийся 4 курса
группы 09-941 (Сарыймова Л.Н.)

Руководитель
доцент, канд. техн. наук (Мубараков Б.Г.)

Заведующий кафедрой системного анализа
и информационных технологий
д-р техн. наук, профессор (Латыпов Р.Х.)

Казань – 2023

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Простые числа играют очень важную роль не только в математике, но и в разработке приложений. Многие криптографические системы для защиты информации используют асимметричные алгоритмы. К ним относятся RSA (схема для защиты программного обеспечения и реализации цифровой подписи), алгоритм Эль – Гамаля (схема для шифрования и разработки цифровой подписи), алгоритм Диффи – Хеллмана (алгоритм передачи ключей шифрования по незащищенному каналу связи) и другие. В их основе находятся простые числа большой разрядности для того, чтобы увеличить криптостойкость данных алгоритмов. Но методы атаки постоянно совершенствуются и увеличивается вероятность взлома. Поэтому очень важно создавать новые методы проверки чисел, которые позволят быстро и качественно находить простые числа среди составных.

Стоит так же отметить, что арифметические операции над числами такой большой разрядности занимает много времени из-за больших и трудоёмких операций над ними. Следовательно, нужно стремиться к тому, чтобы создать не только надёжный, но и быстрый алгоритм поиска простых чисел.

В работе реализован комбинированный алгоритм поиска простых чисел Лукаса – Миллера – Рабина, тест на простоту числа Миллера – Рабина, тест Соловея – Штрассена и тест Леманна. Алгоритмы находят простые числа в предложенном интервале.

После реализации всех алгоритмов было проведено исследование эффективности алгоритма Лукаса – Миллера – Рабина. На разных числовых диапазонах сравнивалось время работы каждого из выбранных алгоритмов и эффективность, т.е. количество правильно найденных чисел, чисел, которые ошибочно были определены в категорию простых, и чисел, которые были упущены из вида алгоритма. Эти исследования помогут понять, какой из выбранных алгоритмов наиболее подходящий для использования его в своих криптографических системах.

Актуальность разработки и анализа методов поиска простых чисел заключается в том, что они позволяют не только защитить информацию в сети Internet, но и решить многие другие задачи в математике и науке. Кроме того, новые методы поиска простых чисел могут привести к открытию новых закономерностей в математике и расширению наших знаний о числах.

Целью выпускной квалификационной работы (далее – ВКР) является исследование комбинированного метода поиска простых чисел Лукаса – Миллера – Рабина. Для достижения этой цели были поставлены следующие задачи:

1) анализ предметной области, изучение вероятностных алгоритмов поиска простых чисел;

2) выбор средств и инструментов, необходимых для реализации методов;

3) разработка архитектуры приложения;

4) программная реализация приложения;

5) тестирование алгоритмов на различных интервалах;

6) сравнение алгоритма Лукаса – Миллера – Рабина с другими вероятностными алгоритмами.

1. Области применения простых чисел

Задача поиска больших простых чисел сложна, над ней трудятся много ученых, и все еще нет алгоритма, который за небольшое количество времени нашел бы простое число большого количества бит. Рассмотрим, для каких целей осуществлён этот поиск.

1.1. Асимметричные алгоритмы шифрования

Криптографические алгоритмы разделяются на симметричные и асимметричные. Различие между двумя этими видами заключается в виде ключа: для симметричных алгоритмов он закрытый, а для асимметричных – открытый. Это основное преимущество, благодаря которым асимметричные алгоритмы получили большую популярность, ведь у симметричных алгоритмов было два довольно больших недостатка. Во-первых, передача такого ключа трудно осуществима, и порой нет полной уверенности в том, что ключ не был получен третьими лицами. Это большой риск, поэтому ключ должен был передаваться по хорошо защищенному каналу связи. Во-вторых, с помощью него нет возможности убедится в подлинности собеседника. Например, при передаче каких-либо документов лицо, получившее этот документ, не способно убедиться в его целостности, и в то же время субъект, разработавший документ, не способен доказать своё авторство.

Асимметричные алгоритмы решают все эти проблемы, ведь у каждого субъекта есть свои собственные открытый и закрытый ключи. Закрытый ключ хранится в секрете и никуда не публикуется, а открытый – становится общедоступным. Один из них помогает зашифровать данные, а другой – расшифровать, поэтому эти алгоритмы и названы асимметричными.

Асимметричные алгоритмы помогают обеспечить информационную безопасность в локальных сетях, в глобальной сети Internet, в банковских и платежных системах, так как могут помочь со следующими задачами:

– обеспечение целостности и конфиденциальности данных, которые передаются по каналам и хранятся в базах данных;

– формирование электронной цифровой подписи;

– передачи ключей, если в будущем передача сообщений будет осуществляться с помощь симметричного шифрования.

Наиболее популярные алгоритмы асимметричного шифрования: протокол Диффи – Хеллмана, RSA, алгоритм Эль – Гамаля, – во всех них за основу берутся простые числа. И для большей надёжности необходимо использовать числа высокой разрядности. Ведь параллельно ведётся работа над взломом этих алгоритмов. Эти методы с каждым разом усовершенствуются, поэтому нужно всегда быть на шаг впереди.

1.2. Развитие математики

Математиков всего мира давно привлекали простые числа. Ещё в 500-300 годах до нашей эры математики школы Пифагора интересовались совершенными и дружественными числами. Совершенным они называли число, которое равно сумме его делителей. А дружественные числа – это числа, у которых сумма делителей одного числа равно второму числу. А делители, как мы знаем, это простые числа. Далее Евклид в 300-м году до нашей эры доказывает несколько теорем. Например, основную теорему арифметики, которая гласит, что любое целое число можно представить в виде произведения простых чисел единственным образом. А также теорему о том, что простых чисел бесконечное множество.

С тех времен появилось много теорем относительно простых чисел. Например, Теорема Дирихле о простых числах в арифметической прогрессии. В ней говорится о том, что любая бесконечная арифметическая прогрессия, где первый член и разность прогрессии – взаимно простые числа, содержит бесконечное количество простых чисел.

Сейчас тоже есть гипотезы о больших числах, которые не доказаны. Так математический институт Клэя назвал 7 теорий и гипотез, которые являются наиболее сложными проблемами математики и за доказательство которых институт выплачивает награду в размере одного миллиона долларов. Эти

задачи так же называют семью задачами тысячелетия. И одна из них – гипотеза Римана. Он привёл формулу, по которой можно рассчитать количество простых чисел на заданном интервале. Однако эта формула включает в себя дзета-функцию: $\zeta(s) = \frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \cdots$, где $s = \sigma + it$. То есть s – комплексная переменная, а $\zeta(s)$ – определяемая с помощью ряда Дирихле функция. Эта формула будет считаться справедливой, если будет приведено доказательство о том, что дзета-функция принимает нулевое значение только в чётных отрицательных числах и комплексных числах с вещественной частью, равной $\frac{1}{2}$.

Существует множество нерешённых вопросов касательно простых чисел, например, гипотеза о числах близнецах, которая говорит о наличие бесконечного числа пар простых чисел, отличающихся на два, или гипотеза Гольдбаха, в которой говорится, что любое чётное число можно представить в виде суммы простых чисел. Именно поэтому, находя всё большие простые числа мы можем опровергнуть или убедиться в корректности некоторых гипотез, а какие-то даже могут помочь в их доказательстве.

2. Исследуемые алгоритмы поиска простых чисел

Алгоритмы поиска простых чисел делятся на три категории:

– детерминированные, то есть дающие результат со 100%-ной точностью;

– вероятностные, которые могут назвать число простым или составным с какой-то вероятностью;

– основанные на предположительно истинных, но недоказанных гипотезах.

К детерминированным алгоритмам относятся метод перебора делителей, тест Миллера, критерий Поклингтона, тесты, основанные на эллиптических кривых. Среди вероятностных можно выделить алгоритм Миллера – Рабина, тест Лукаса – Миллера – Рабина, алгоритм Соловея – Штрассена, тест Леманна и т.д. К методам, которые базируются на предположительной справедливости недоказанных гипотез (например, расширенная гипотеза Римана), можно отнести методы Адлемана – Румели, Ленстры – Коэна и другие.

Детерминированные методы поиска простых чисел обладают следующими недостатками: для некоторых из них требуется разложение на простые сомножители, а это является трудоёмкой задачей, они сложны в реализации и занимают большое количество времени. Если использовать методы, базирующиеся на гипотезах, нет уверенности, что числа действительно простые, ведь нет строгого доказательства этих гипотез, они часто требуют длительных вычислений. А вероятностные методы, хоть и не дают полную уверенность, что число простое, зато справляются за небольшое время и несложные в реализации. Именно поэтому при реализации асимметричных методов чаще выбирают вероятностные тесты.

Основной алгоритм ВКР – алгоритм Лукаса – Миллера – Рабина. Он является комбинированным алгоритмом, так как сочетает в себе усиленную версию теста простоты Лукаса, основанной на ряде Фибоначчи [1]. Его усилили дополнительной проверкой по аналогии с тестом Миллера – Рабина.

Тест Леманна – один из первых вероятностных тестов на простоту. Он является разновидностью теста Ферма. Для проверки числа на простоту с помощью теста Леманна выбираются случайные целые числа a и проверяется выполнение условия $a^{n-1} = 1 \, mod \, n$, при условии, что a и n – взаимно простые. Вероятность ошибки составляет не более 50% [2].

Алгоритм проверки числа на простоту Миллера – Рабина – это вероятностный тест, который тоже основан на малой теореме Ферма. Алгоритм Миллера – Рабина позволяет выполнять проверку за малое время и давать при этом достаточно малую вероятность того, что число на самом деле является составным. Он более точен, чем метод Леманна [3].

Тест Соловея – Штрассена был открыт в 1957 году. Он менее эффективен и менее точен, чем метод Миллера – Рабина, так как использует больше итераций для достижения той же точности [4]. Алгоритм Соловея – Штрассена является хорошим выбором для проверки простоты чисел в условиях ограниченных вычислительных ресурсов.

Рассмотрим каждый из них подробно.

2.1. Алгоритм Леманна

Алгоритм Леманна базируется на малой теореме Ферма, которая утверждает, что если число n – простое, число a – целое и неделящееся на n, то выполняется условие: $a^{n-1} \equiv 1 \, (mod \, n)$ [5]. Преобразовав это условие следующим образом:

$$a^{n-1} - 1 \equiv 0 \, (mod \, n)$$

$$(a^{\frac{n-1}{2}} - 1)(a^{\frac{n-1}{2}} + 1) \equiv 0 \, (mod \, n)$$

Обозначим $\beta = \frac{n-1}{2}$ и получим следующее утверждение: если $a^{\beta} = \pm 1$, то число n – простое. На проверке этого условия и основан алгоритм Леманна:

1) выбирается любое целое число a из промежутка (1, n), взаимно простое с n, число a называют базой;

2) проверяется условие: $a^{\beta} = \pm 1$. Если оно выполняется, то число вероятно простое.

Шаги 1-2 повторяются k раз, благодаря чему можно сделать вывод, что каждое число, которое алгоритм посчитал простым, является им с вероятностью 2^{-k}.

2.2. Алгоритм Лукаса – Миллера – Рабина

Этот алгоритм использует тест простоты Лукаса, который использует ряд Фибоначчи, задающийся определением:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_{n+2} = F_{n+1} + F_n. \end{cases}$$

Этот тест выглядит следующим образом. Если число n > 5 удовлетворяет условию: $F_{n-e} \equiv 0 \ (mod\ n)$, где $e = L(n, 5)$ – символ Лежандра, то n – простое.

Тест Лукаса был усилен по аналогии с тестом Миллера – Рабина, поэтому и был назван алгоритмом Лукаса – Миллера – Рабина. Он состоит из следующих шагов:

1) находится число $e = L(n, 5)$ – символ Лежандра;

2) следом число $n - e$ представляется в виде произведения чётной и нечётной части. То есть: $n - e = t \cdot 2^s$;

3) выполняется проверка условия: $F_t \equiv 0 \ (mod\ n)$. Если оно выполняется, то число вероятно простое. Иначе – переход к следующему шагу;

4) ищется такое $m = t \cdot 2^i$, где $0 < i < s$, для которого справедливо $F_{m-1} + F_{m+1} \equiv 0 \ (mod\ n)$. Если m найдено, то число считается вероятно простым. Если условие не выполняется, то число вероятно составное.

Для того, чтобы найти функцию Фибоначчи от числа, применяется следующее соотношение:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n.$$

То есть, перемножив n раз матрицы вида $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$ и взяв верхний справа элемент этой матрицы, можно получить функцию Фибоначчи от числа n.

Этот алгоритм был придуман преподавателями кафедры системного анализа и информационных технологий Казанского Федерального Университета. Он относительно новый, если сравнивать с другими алгоритмами, так как статья по нему вышла в 2022 [6]. В ней авторы рассказали об усилении алгоритма Лукаса, привели теоремы, благодаря которым доказана его эффективность, приведены примеры LMR-псевдопростых чисел, которые алгоритм Лукаса – Миллера – Рабина может определить как составные.

2.3. Алгоритм Миллера – Рабина

Алгоритм Миллера – Рабина является модификацией теста Миллера, который основан на недоказанной расширенной гипотезе Римана. Майкл Рабин модифицировал этот метод так, что он не зависит от гипотезы Римана и является вероятностным [7].

В этом алгоритме используются следующие утверждения.

Пусть число n – простое, а число n-1 представляется в виде произведения четной и нечетной части, то есть $n - 1 = 2^s \cdot d$. Тогда для любого числа a, взаимно простого с n, выполняется хотя бы одно из условий:

1) $a^d \equiv 1\ (mod\ n)$;

2) существует целое число $r < s$ такое, что $a^{2^r d} \equiv -1\ (mod\ n)$.

Число a называют базой или свидетелем простоты.

На рисунке 1 представлен псевдокод, который описывает этот тест простоты.

Ввод: n > 2, нечетное натуральное число

Вывод: "составное" или "вероятно простое"

Представить n-1 **в виде** $2^s t$

Цикл A: **повторить** k **раз**:

Выбрать случайное целое число a в отрезке [2, n-2]

Вычислить $x = a^t \bmod n$

Если x = 1 **или** x = n-1, **то перейти на следующую итерацию цикла A**

Цикл B: **повторить** s-1 **раз**

$x = x^2 \bmod n$

Если x = 1, **то вернуть** "составное"

Если x = n-1, **то перейти на следующую итерацию цикла A**

Вернуть "составное"

Вернуть "вероятно простое"

Рисунок 1 – Алгоритм Миллера – Рабина, записанный на псевдокоде

2.4. Алгоритм Соловея – Штрассена

Тест Соловея – Штрассена также опирается на малую теорему Ферма. Однако существуют составные числа, которые называются числами Кармайкла и для которых утверждение из малой теоремы Ферма выполняется для любой базы [8]. Тест Соловея – Штрассена, как и тест Миллера – Рабина, способен отсеять числа Кармайкла при поиске простых чисел.

Этот алгоритм базируется на утверждении о том, что число n – составное тогда, когда количество целых чисел a, меньших n и взаимно простых с n, которые удовлетворяют условию: $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) (mod\ n)$, не превосходит $\frac{n}{2}$. В этом условии $\left(\frac{a}{n}\right)$ является символом Якоби [9].

На рисунке 2 можно увидеть, как описан алгоритм на псевдокоде.

Ввод: $n > 2$, нечетное натуральное число
k - параметр точности теста
Вывод: "составное" или "вероятно простое"

for $i = 1, 2, ..., k$:
  Выбрать случайное целое число a в отрезке $[2, n-1]$
  Если НОД$(a, n) > 1$, **то:**
    Вернуть "составное"
  Если $a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}$, **тогда:**
    Вернуть "составное"
Вернуть "вероятно простое"

Рисунок 2 – Алгоритм Соловея – Штрассена на псевдокоде

3. Программная реализация приложения

3.1. Архитектура приложения

Перед тем, как разработать архитектуру приложения, нужно понять, для каких целей оно должно быть разработано. ВКР посвящена исследованию алгоритмов поиска простых чисел. Значит, в ней должны быть реализованы все исследуемые методы проверок: алгоритм Лукаса – Миллера – Рабина, алгоритм Миллера – Рабина, тест Соловея – Штрассена и тест Леманна. Кроме того, эти алгоритмы должны искать простые числа в определенном интервале. Значит, должны быть поля, где можно было бы ввести числа, между которыми будет осуществлён поиск. Также под каждой кнопкой с определенной функцией должно быть поле, в котором будет отображаться результат. И для того, чтобы сравнивать не только корректность выполнения разных функций, но и скорость работы каждого алгоритма, необходимо создать поля для отображения времени действия.

Таким образом, можно сделать вывод, что будет удобно сделать оконное приложение. Его прототип должен состоять из следующих элементов:

– поля для ввода количества бит двух чисел, между которыми осуществляется поиск (эти поля должны быть предназначены только для чисел и иметь ограничения);

– кнопки для каждой функции;

– текстовые поля, куда будут выводится найденные простые числа выбранного диапазона (важно, чтобы у пользователя не было возможности изменять значения в этом поле);

– поля для отображения времени действия каждой функции (эти поля также не должны быть изменяемыми пользователем).

Схематично прототип приложения должен был выглядеть следующим образом (рисунок 3).

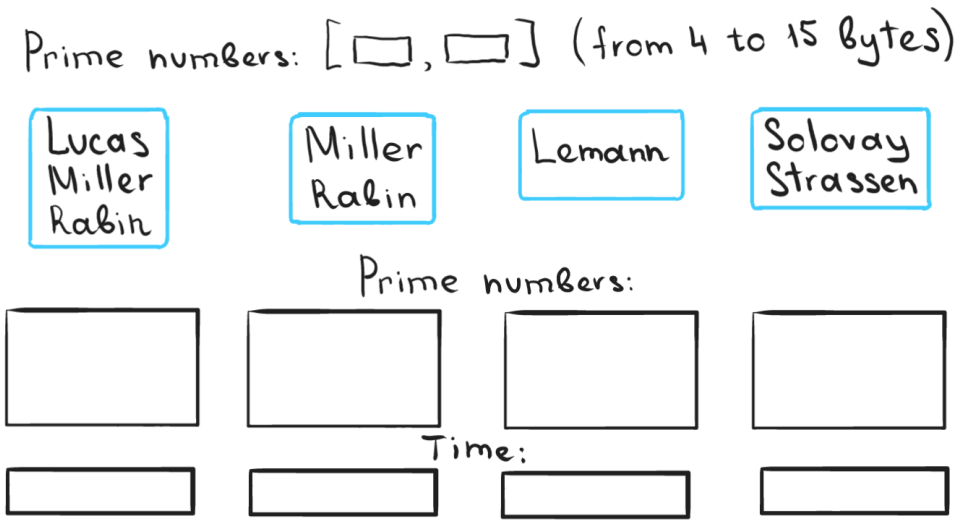

Рисунок 3 – Схема приложения

После разработки данной архитектуры началась работа над непосредственно самим приложением.

3.2. Технические особенности создания приложения

Так как на этапе создания архитектуры наиболее удобным вариантом стало создание оконного приложения, то выбор пал на среду разработки Microsoft Visual Studio, так как в ней удобно писать программный код на языке C#, а также реализовывать оконные приложения с помощью Windows Forms, где разработчикам предоставлены инструменты для создания окон, кнопок, текстовых полей, списков и других элементов интерфейса, а также возможности для их расположения и стилизации. Windows Forms использует язык программирования C# или Visual Basic .NET и работает на платформе .NET Framework.

На форму приложения были добавлены все перечисленные выше окна и текстовые поля. Затем была начата работа над программной реализацией. После нажатия на одну из кнопок первоначально осуществлялась проверка на то, заполнены ли все окна для ввода количества бит чисел и правильно ли они заполнены. Если на этом этапе все прошло успешно, то начинается генерация всех чисел определенного количества бит. Для генерации всех чисел заданного интервала была создана функция GenerateAllNumbers(). Следом из этих чисел

отбираются все нечетные и отправляются на один из выбранных алгоритмов. С этого момента начинает отсчет времени для работы алгоритма. Если алгоритм считает какое-то из чисел простым, то оно добавляется в общую строку со всеми найденными простыми числами. Как только все числа из заданного диапазона были переданы в алгоритм и получен результат, поиск считается завершенным, отсчёт времени останавливается. Найденные числа анализируются и сравниваются со списком эталонных простых чисел. В окошке снизу выводятся ошибки алгоритма и найденные простые числа, время выполнения помещается в окошко ниже. В общем виде программа выглядит следующим образом (рисунок 4).

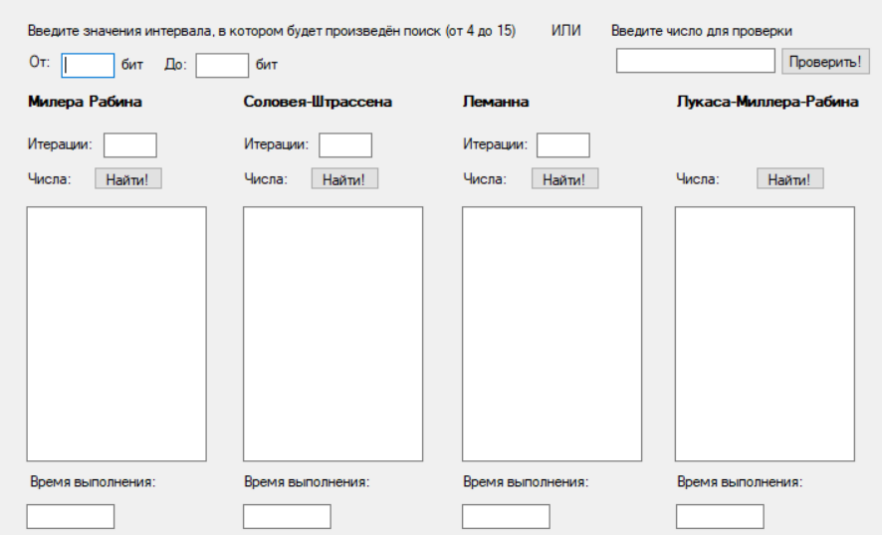

Рисунок 4 – Итоговый вид приложения

Были проработаны все исключения, которые могут возникнуть. На рисунках 5-10 можно увидеть все ситуации, при которых всплывает окно с ошибкой.

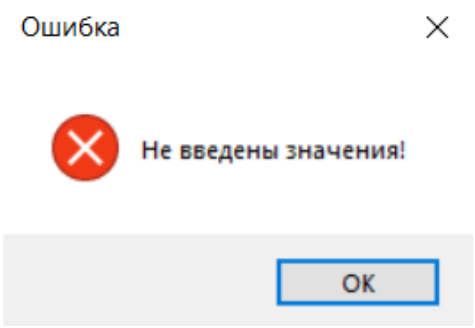

Рисунок 5 – Ошибка, которая возникает, когда не введено одно из значений интервала

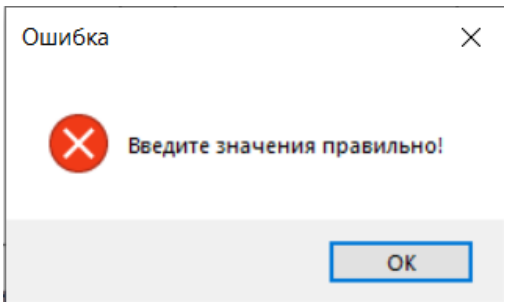

Рисунок 6 – Ошибка, которая возникает, когда введены какие-либо другие символы, кроме чисел

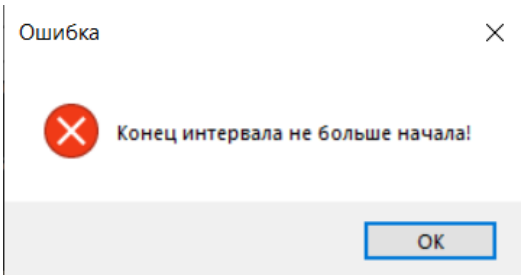

Рисунок 7 – Ошибка, которая возникает, когда конечное значение интервала больше начального

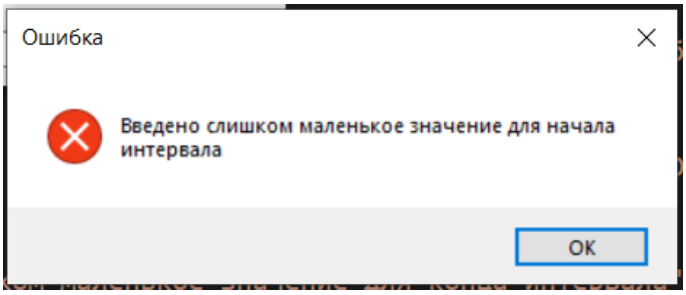

Рисунок 8 – Ошибка, которая возникает, когда значение начала интервала меньше 4

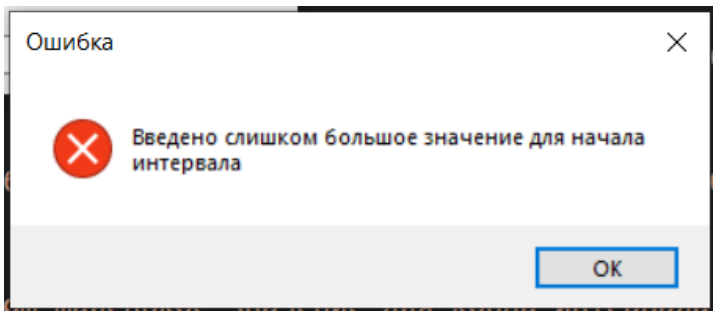

Рисунок 9 – Ошибка, которая возникает, когда значение начала интервала больше 15

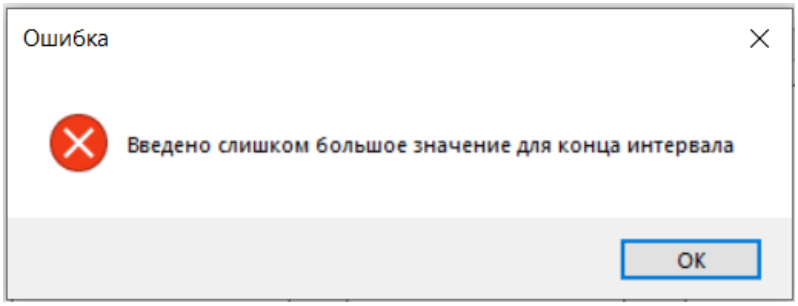

Рисунок 10 – Ошибка, которая возникает, когда значение конца интервала больше 15

Для генерации случайного числа используется класс RNGCryptoServiceProvider, генерирующий число определенного количества бит. За тест простоты Миллера – Рабина отвечает функция Miller_Rabin(). В ней используется функция powmod(). Она возводит число в степень по модулю, используя быстрый алгоритм. За тест простоты Леманна отвечает функция Lemann(). В ней используется функция поиска наибольшего общего делителя gcd(). Solov_Shtrass() – функция, с помощь которой реализован тест простоты Соловея – Штрассена. В её реализации можно увидеть функцию Jacobi(), она находит символ Якоби. Функция Lucas_Miller_Rabin() осуществляет проверку числа на простоту методом Лукаса – Миллера – Рабина. Она содержит в себе функцию Fibonacci(), где вычисляется число в ряде Фибоначчи, которая в свою очередь содержит функцию MatrixMultiply(), так как поиск числа в ряде Фибоначчи осуществляется с помощью матричного умножения.

Результат действия программы можно увидеть на рисунке 11.

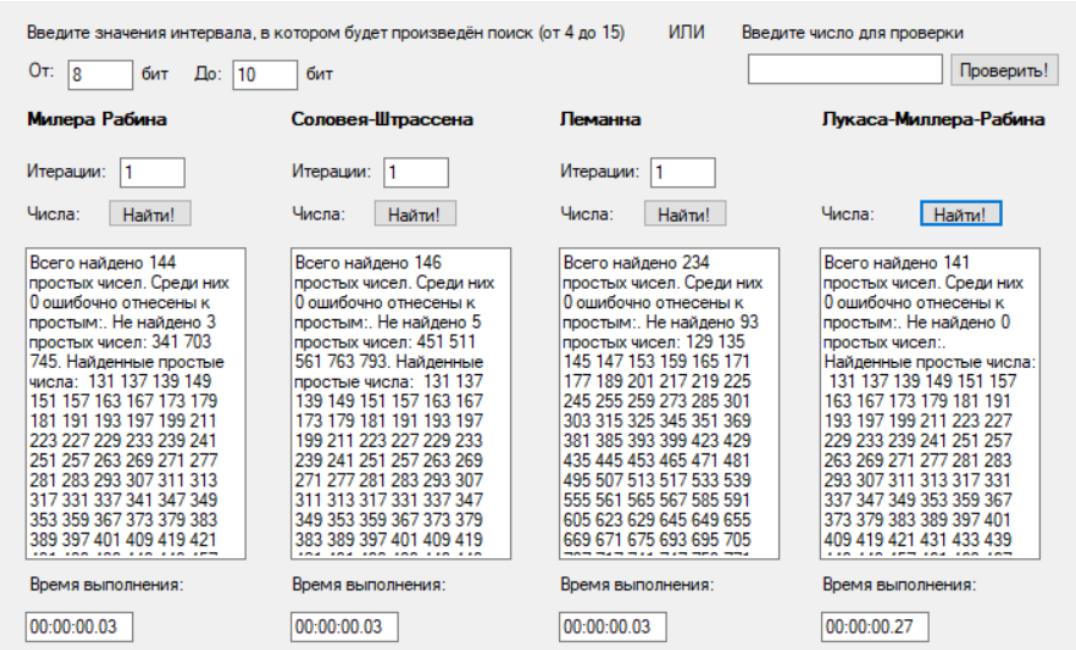

Рисунок 11 – Результат действия программы

Был добавлен ещё один способ проверки. Справа сверху было добавлено окно, в которое можно поместить одно число, и после нажатия кнопки «Проверить!», число будет проверятся каждым тестом поочередно. Результат действия можно увидеть на рисунке 12. Эта проверка было добавлена на тестирование алгоритмов с помощью различных псевдопростых чисел.

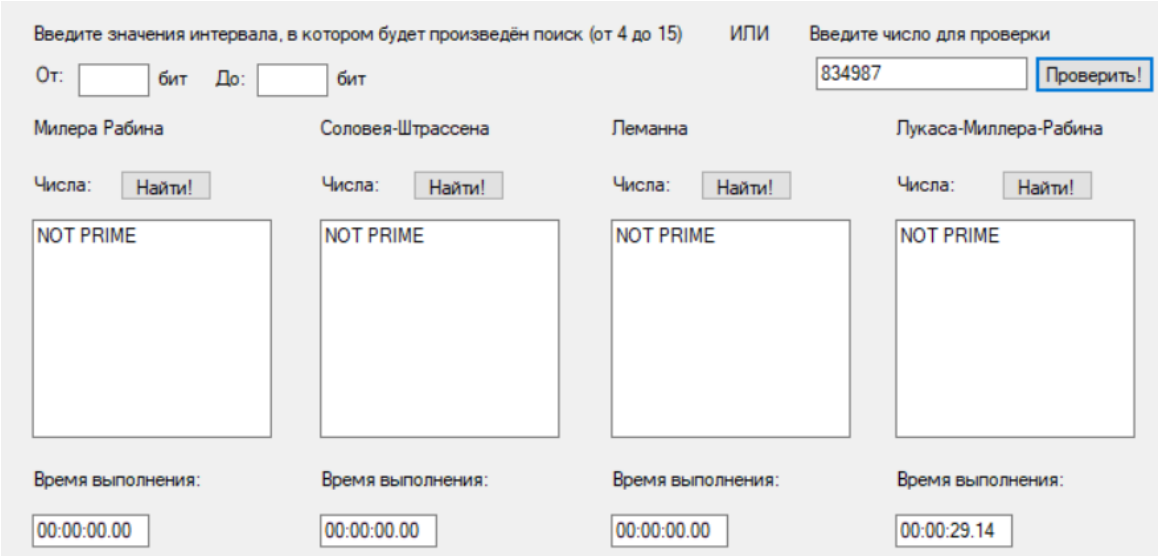

Рисунок 12 – Результат проверки одного числа

Также было необходимо получить истинный список всех простых чисел до 15 бит включительно, чтобы потом сравнивать найденные алгоритмами числа с эталонными. Для этого был использован язык Python и

детерминированный алгоритм решета Эратосфена [10]. Этот алгоритм составляет список всех чисел от 4 до 15 бит, и потом удаляет из него все числа, кратные 2, 3, и так далее. Благодаря этому алгоритму в отдельный файл были построчно сохранены все простые числа. Для такого большого количества чисел алгоритм выполнялся дольше, чем с вероятностными методами, однако это позволило собрать базу истинно простых чисел. С помощью этого файла в будущем стало возможным сравнение выбранных алгоритмов опираясь не только на время выполнения, но и на корректность выполнения алгоритма.

Таким образом, была реализована программа для исследования алгоритма Лукаса – Миллера – Рабина и для сравнения его с другими алгоритмами. Она позволяет искать простые числа в заданном интервале и проверять на простоту конкретное число. Благодаря этой программе стало возможным исследование алгоритма, его тестирование на различных диапазонах.

4. Тестирование алгоритмов

Тестирование алгоритмов поиска простых чисел будет произведено следующими способами:

1) на интервалах по 2 бита каждый запускается алгоритм с разным количеством итераций и проверяется количество ошибок, в алгоритмах используются случайные базы;

2) на интервалах по 2 бита каждый запускается алгоритм с разным количеством итераций и проверяется количество ошибок, в алгоритмах используются одинаковые базы;

3) на интервале 4-14 бит запускается алгоритм с разными базами и проверяется зависимость базы алгоритма и количества ошибок;

4) на интервале 4-14 бит запускается алгоритм с одинаковыми базами и проверяется время выполнения алгоритмов;

5) проверка алгоритмов на различные псевдопростые числа.

Результаты этих тестов будут приведены в следующих главах.

Тестирование проводилось с помощью ноутбука с процессором Intel Core i5-8300H CPU @ 2.30GHz, 4 ядра с оперативной памятью 8 ГБ.

4.1. Тестирование со случайными базами

Во время тестирования каждый из алгоритмов искал простые числа на заданном диапазоне. Изменялось количество итераций, за которые алгоритму предлагалось проверить числа. Был произведён подсчет ошибок, сделанных алгоритмом, суммируются количество чисел, которые алгоритм ошибочно посчитал простыми, и количество чисел, которые алгоритм посчитал составными и упустил. Это тестирование позволит понять, как зависит количество ошибок от количества итераций. В таблице 1 представлены результаты проведенного тестирования для каждого алгоритма.

Таблица 1 – Результаты тестирования алгоритмов на количество ошибок со случайными базами

Интервал, бит	Количество итераций	Количество ошибок алгоритма Миллера–Рабина	Количество ошибок алгоритма Соловея–Штрассена	Количество ошибок алгоритма Леманна	Количество ошибок алгоритма Лукаса–Миллера–Рабина
4-6	1	0	0	4	0
	2	0	0	2	0
	3	0	0	0	0
6-8	1	1	1	14	0
	2	0	0	6	0
	3	0	0	2	0
	4	0	0	1	0
	5	0	0	0	0
8-10	1	2	2	63	0
	2	0	1	23	0
	3	0	0	11	0
	4	0	0	6	0
	5	0	0	4	0
	6	0	0	2	0
	7	0	0	0	0
10-12	1	5	8	233	0
	2	0	1	105	0
	3	0	0	56	0
	4	0	0	25	0
	5	0	0	12	0
	6	0	0	5	0
	7	0	0	2	0
12-14	1	12	15	873	5
	2	1	3	444	5
	3	0	1	268	5

Продолжение таблицы 1

Интервал, бит	Количество итераций	Количество ошибок алгоритма Миллера–Рабина	Количество ошибок алгоритма Соловея–Штрассена	Количество ошибок алгоритма Леманна	Количество ошибок алгоритма Лукаса–Миллера–Рабина
	4	0	0	143	5
	5	0	0	70	5
	6	0	0	34	5
	7	0	0	15	5
	8	0	0	7	5
	9	0	0	3	5
	10	0	0	2	5
	11	0	0	1	0

Объединив результаты всех интервалов был составлен общий график. На рисунке 13 представлен график зависимости количества ошибок от количества итераций для алгоритмов Миллера – Рабина, Соловея – Штрассена и Лукаса – Миллера – Рабина.

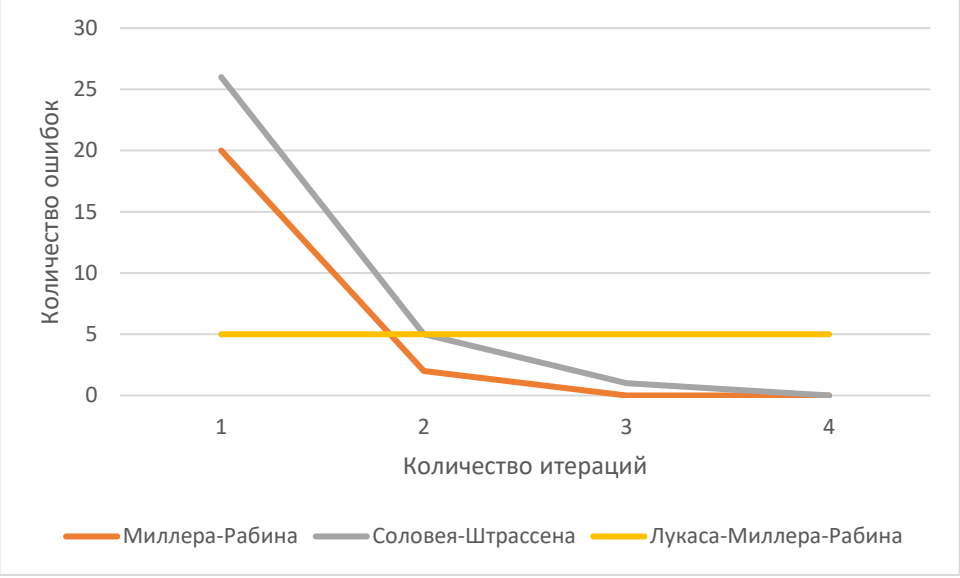

Рисунок 13 – Зависимость количества ошибок от количества итераций на интервале 4-14 бит для трех алгоритмов

Отдельно приведён график для алгоритма Леманна на рисунке 14.

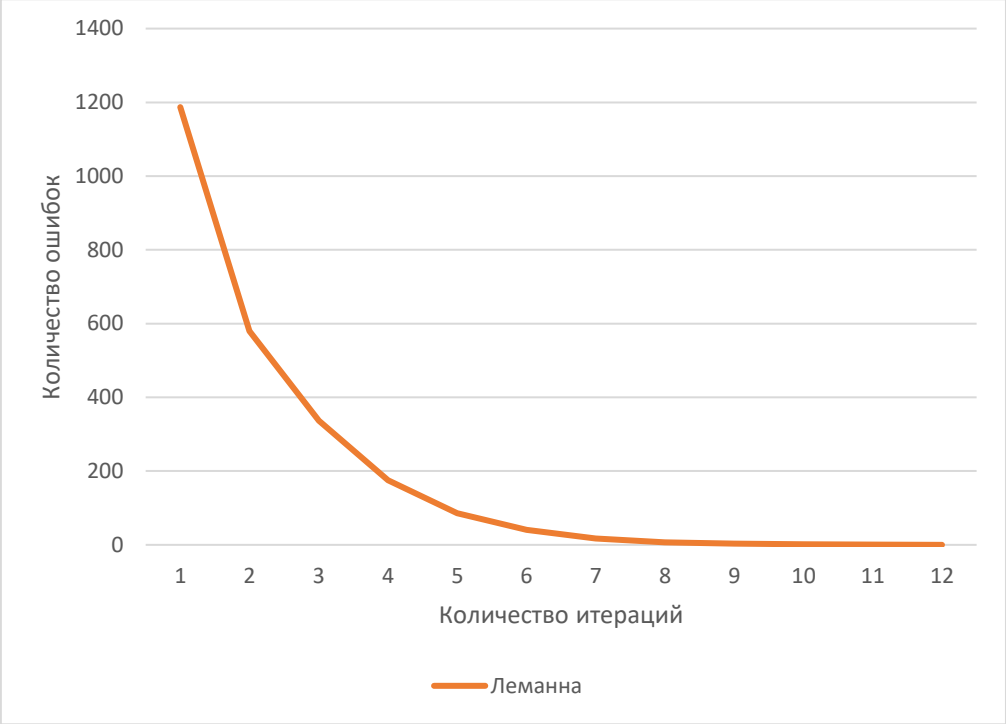

Рисунок 14 – Зависимость количества ошибок от количества итераций на интервале 4-14 бит для алгоритма Леманна

По графику можно сделать вывод: количество ошибок экспоненциально уменьшается с ростом итераций для вероятностных алгоритмов. Алгоритм Лукаса – Миллера – Рабина на 1 итерации совершает меньше ошибок, чем другие вероятностные алгоритмы. На всём интервале он совершил 5 ошибок, посчитал простыми 5 составных чисел, это LMR-псевдопростые числа.

4.2. Тестирование с одинаковыми базами

В предыдущем эксперименте для каждого алгоритма выбиралась случайная база, которая могла стать свидетелем простоты какого-либо числа. Этот эксперимент подобен предыдущему, однако база для алгоритмов будет одинаковая. Если производится 1 итерация, то базой выступает число 2, если 2 итерации – числа 2 и 3, и так далее берутся все числа из последовательного ряда простых чисел. Результаты тестирования приведены в таблице 2.

Таблица 2 – Результаты тестирования алгоритмов на количество ошибок с использованием одинаковых баз

Интервал, бит	Количество итераций	Количество ошибок алгоритма Миллера–Рабина	Количество ошибок алгоритма Соловея–Штрассена	Количество ошибок алгоритма Леманна	Количество ошибок алгоритма Лукаса–Миллера–Рабина
4-6	0	0	0	0	0
6-8	0	0	0	0	0
8-10	1	0	1	2	0
	2	0	0	1	0
	3	0	0	0	0
10-12	1	3	5	5	0
	2	0	1	2	0
	3	0	1	1	0
	4	0	0	1	0
	5	0	0	1	0
12-14	1	5	8	11	5
	2	0	1	3	5
	3	0	1	2	5
	4	0	0	1	5
	5	0	0	1	5
	6	0	0	1	5
	7	0	0	1	5

Объединив интервалы был составлен график зависимости количества ошибок от количества итераций, представленный на рисунке 15.

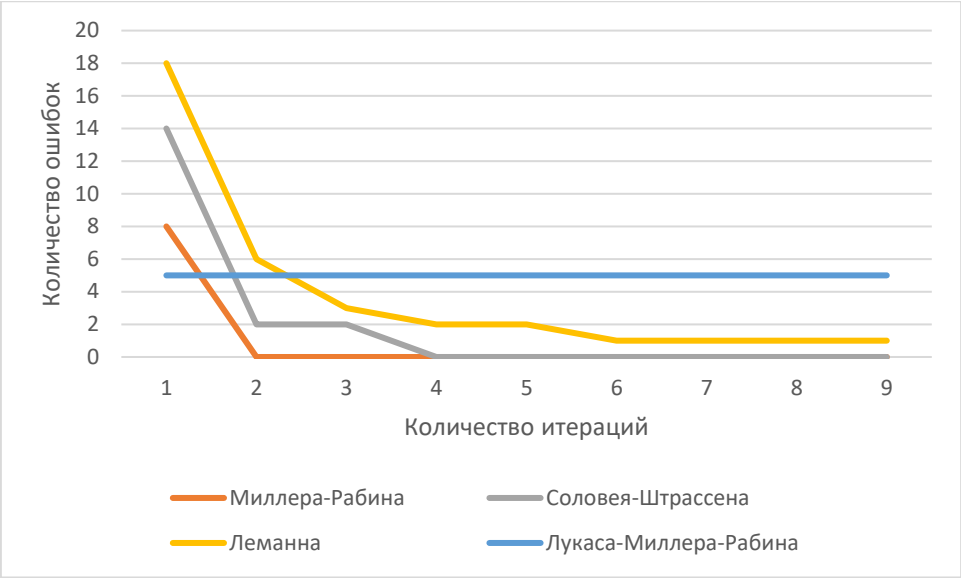

Рисунок 15 – Зависимость количества ошибок от количества итераций на интервале 4-14 бит с использованием одинаковых баз

Как видно по рисунку, алгоритм Лукаса – Миллера – Рабина справляется так же лучше, чем представленные алгоритмы, так как за 1 итерацию способен совершить меньшее количество ошибок, если найти хороший алгоритм для отбраковки LMR-псевдопростых чисел, то можно получить хороший полиномиальный детерминированный тест простоты.

4.3. Тестирование на количество ошибок

В этом эксперименте представлена попытка провести вычисления вероятностных алгоритмов за 1 итерацию с разными базами и сравнение их количества ошибок с количеством ошибок теста Лукаса – Миллера – Рабина. Для этого был задан интервал от 4 до 14 бит и подсчитано количество ошибок для вероятностных алгоритмов. Результаты представлены в таблице 3.

Таблица 3 – Результаты тестирования алгоритмов на количество ошибок с 1 итерацией и использованием различных баз

База	Алгоритм Миллера – Рабина	Алгоритм Соловея – Штрассена	Алгоритм Леманна
2	8	14	18
3	7	12	1023
5	8	14	17

Продолжение таблицы 3

База	Алгоритм Миллера – Рабина	Алгоритм Соловея – Штрассена	Алгоритм Леманна
7	6	6	945
11	19	25	2209
13	8	10	516
17	11	12	401
19	18	21	374
23	15	18	313

Среднее значение ошибок всех алгоритмов можно увидеть на рисунке 16.

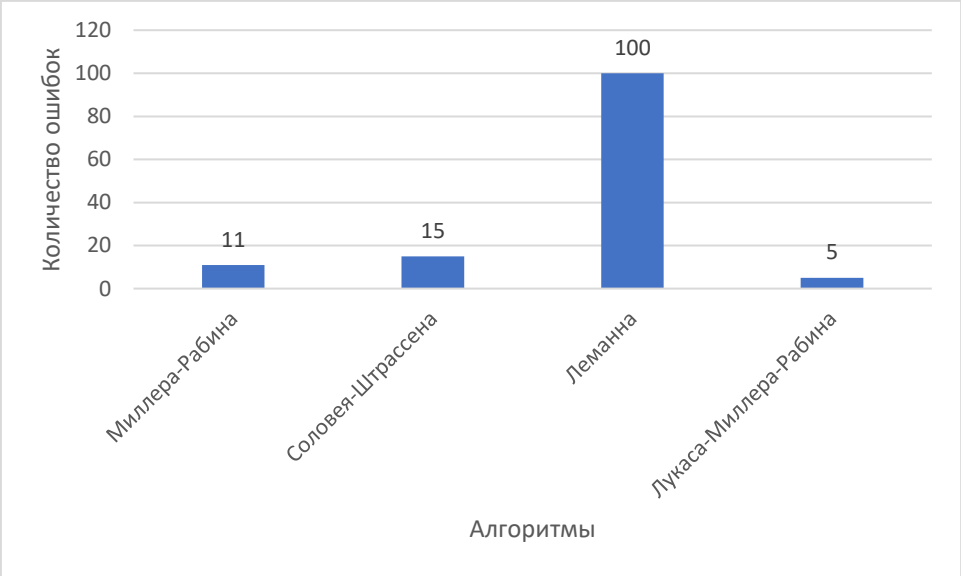

Рисунок 16 – Среднее количество ошибок за 1 итерацию на интервале 4-14 бит

Среднее количество ошибок алгоритмов за 1 итерацию больше, чем количество ошибок Лукаса – Миллера – Рабина на интервале 4-14 бит. Ближе всего справлялся алгоритмы Миллера – Рабина и Соловея – Штрассена с базой 7. Однако стоит учитывать, что был рассмотрен небольшой интервал, поэтому следует смотреть на среднее количество ошибок.

4.4. Тестирование времени работы алгоритмов

При проведении этого тестирования искались простые числа на разных интервалах и сравнивались количество итераций и затраченное время. Так как на предыдущих этапах выполнялось вычисление корректности работы алгоритма, сравнение полученных чисел со списком правильных простых чисел, что занимало много времени, поэтому на этом этапе можно брать интервалы не по 2 бит, а взять один интервал от 4 до 11 бит и увидеть время выполнения каждого алгоритма. Полученные результаты оформлены в таблице 4.

Таблица 4 – Результаты тестирования алгоритмов на время работы

Интервал, бит	Количество итераций	Время выполнения алгоритма Миллера–Рабина, мс	Время выполнения алгоритма Соловея–Штрассена, мс	Время выполнения алгоритма Леманна, мс	Время выполнения алгоритма Лукаса–Миллера–Рабина, мс
4 - 11	1	24	4	4	9
	2	25	5	6	18
	3	26	7	8	23
	4	28	9	10	25
	5	29	10	11	30
	6	32	12	13	33
	7	35	13	15	36

График, изображающий приведённые выше данные, можно увидеть на рисунке 17.

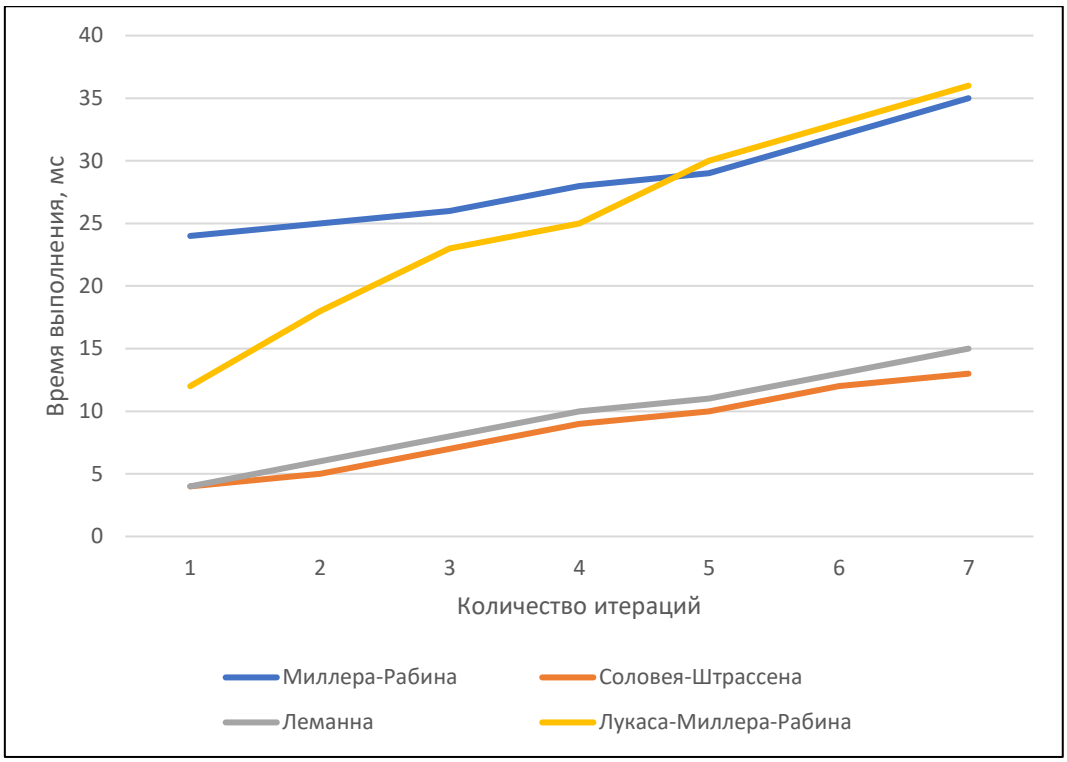

Рисунок 17 – Зависимость времени выполнения алгоритмов от количества итераций

По графику можно сделать вывод: время выполнения линейно зависит от количества итераций. Алгоритм Лукаса – Миллера – Рабина справляется примерно с такой же скоростью, как алгоритм Миллера – Рабина.

4.5. Тестирование с помощью псевдопростых чисел.

Для построения криптосистем часто нужно найти несколько простых чисел определенного количества бит, а не все числа интервала. Так как проверка большого количества чисел занимает длительное время, было принято решение провести тестирование больших псевдопростых чисел, с которыми могут быть проблемы у алгоритмов. Все числа проверялись с одной итерацией и выбирался наиболее часто встречаемый ответ алгоритма.

Существует несколько видов псевдопростых чисел:

- Кармайкла числа,
- Эйлера – Якоби – псевдопростые числа,
- числа Вудала,
- сильные псевдопростые.

Число Кармайкла – это число n, которое является составным и которое удовлетворяет условию $b^{n-1} \equiv 1\ (mod\ n)$ для всех целых b, взаимно простых с n. Такие числа не удовлетворяют только проверке тест Ферма, поэтому исследуемые алгоритмы могут посчитать их простыми.

В программу подавались следующие числа Кармайкла: 561, 1105, 1729, 2465, 2821, 6601, 8911, 41041, 62745, 63973, 75361, 101101, 126217, 172081, 188461, 278545, 340561, 825265. Была произведена 1 итерация с базой 2 в каждом алгоритме. Пример можно увидеть на рисунке 18.

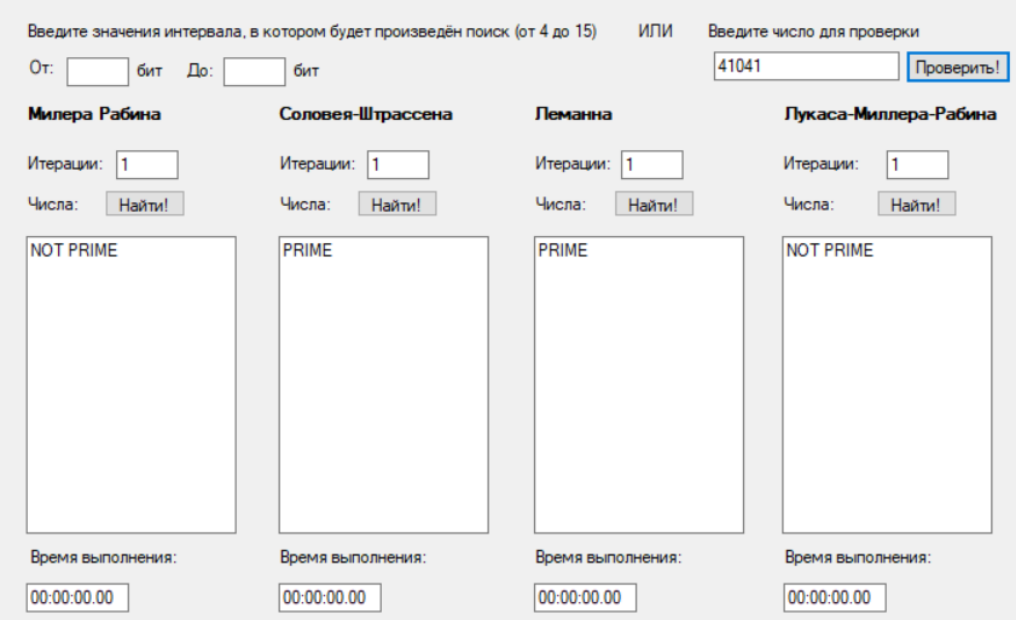

Рисунок 18 – Проверка числа 41041

Результаты тестирования приведены в таблице 5.

Таблица 5 – Результаты тестирования алгоритмов числами Кармайкла

Число	Результат алгоритма Миллера–Рабина	Результат алгоритма Соловея–Штрассена	Результат алгоритма Леманна	Результат алгоритма Лукаса–Миллера–Рабина
561	Составное	Простое	Простое	Составное
1105	Составное	Составное	Составное	Составное
1729	Составное	Простое	Простое	Составное
2465	Составное	Составное	Составное	Составное

Продолжение таблицы 5

Число	Результат алгоритма Миллера–Рабина	Результат алгоритма Соловея–Штрассена	Результат алгоритма Леманна	Результат алгоритма Лукаса–Миллера–Рабина
2821	Составное	Составное	Составное	Составное
6601	Составное	Простое	Простое	Составное
8911	Составное	Составное	Составное	Составное
41041	Составное	Простое	Простое	Составное
62745	Составное	Составное	Простое	Составное
63973	Составное	Составное	Составное	Составное
75361	Составное	Простое	Простое	Составное
101101	Составное	Составное	Составное	Составное
126217	Составное	Простое	Простое	Составное
172081	Составное	Простое	Простое	Составное
188461	Составное	Составное	Составное	Составное
278545	Составное	Составное	Составное	Составное
340561	Составное	Простое	Простое	Составное
825265	Составное	Составное	Составное	Составное

Подсчитав количество ошибок, была сделана столбчатая диаграмма, представленная на рисунке 19.

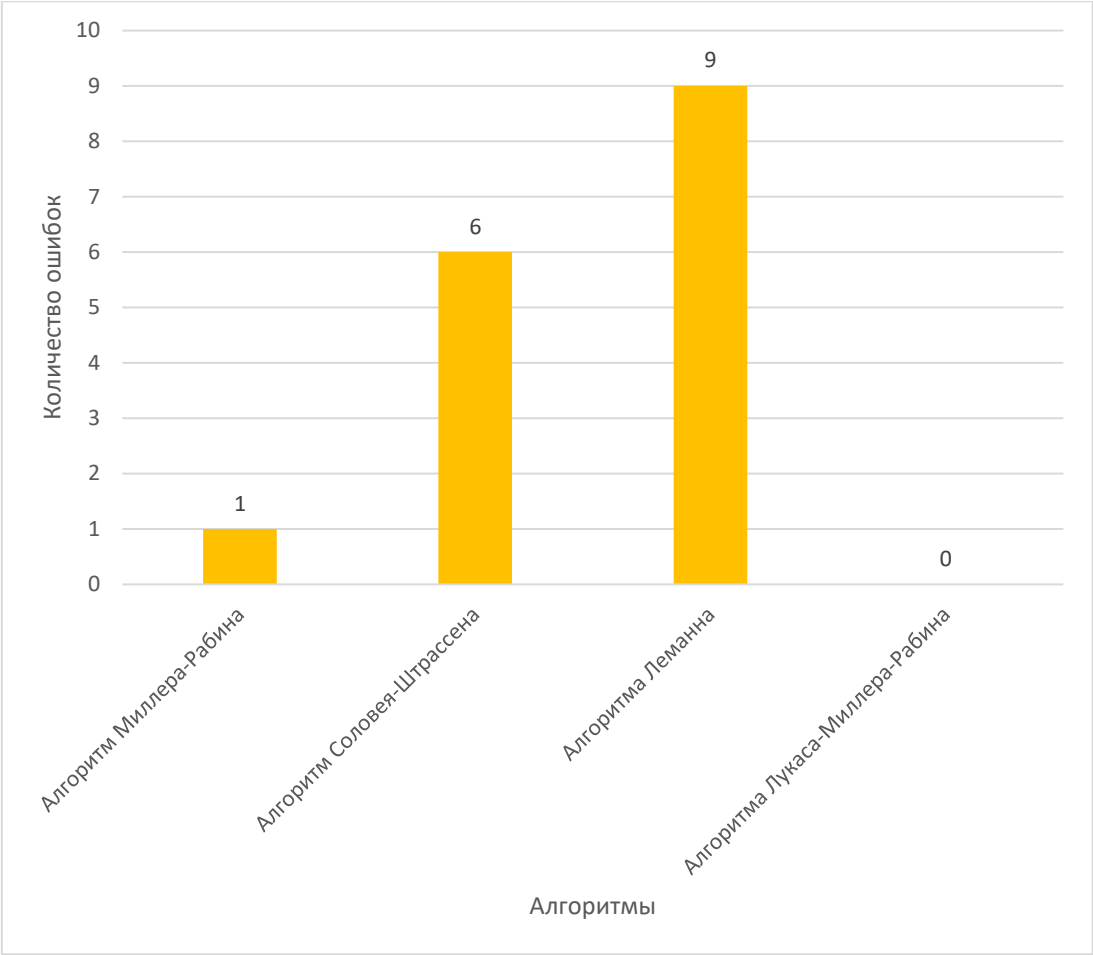

Рисунок 19 – Результат тестирования алгоритмов числами Кармайкла

Алгоритм Лукаса – Миллера – Рабина прошел проверку числами Кармайкла и на одной итерации не совершил ни одной ошибки. Хуже всего себя проявил алгоритм Леманна, он совершил 11 ошибок.

Далее была проверка алгоритмов Эйлера – Якоби – псевдопростыми числами. Эйлера – Якоби – псевдопростые числа – это составные числа, которые удовлетворяют условию $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) mod\ n$, где n – Эйлера – Якоби – псевдопростое число по основанию a, $\left(\frac{a}{n}\right)$ – символ Якоби.

Псевдопростые Эйлера — Якоби по основанию 2: 561, 1105, 1729, 1905, 2047, 2465, 3277, 4033, 4681, 6601, 8321, 8481, 10585.

Псевдопростые Эйлера — Якоби по основанию 3: 121, 703, 1729, 1891, 2821, 3281, 7381, 8401, 8911, 10585, 12403, 15457, 15841.

Результат проверки представлен в таблице 6.

Таблица 6 – Результат тестирования алгоритмов Эйлера – Якоби – псевдопростыми числами

Число	Результат алгоритма Миллера–Рабина	Результат алгоритма Соловея–Штрассена	Результат алгоритма Леманна	Результат алгоритма Лукаса–Миллера–Рабина
121	Составное	Составное	Составное	Составное
561	Составное	Простое	Простое	Составное
703	Составное	Составное	Составное	Составное
1105	Составное	Составное	Составное	Составное
1729	Составное	Простое	Простое	Составное
1891	Составное	Составное	Составное	Составное
1905	Составное	Составное	Составное	Составное
2047	Простое	Простое	Простое	Составное
2465	Составное	Составное	Составное	Составное
2821	Составное	Составное	Составное	Составное
3277	Простое	Простое	Простое	Простое
3281	Составное	Составное	Составное	Составное
4033	Простое	Простое	Простое	Составное
4681	Простое	Простое	Простое	Составное
6601	Составное	Простое	Простое	Составное
7381	Составное	Составное	Составное	Составное
8321	Простое	Простое	Простое	Составное
8401	Составное	Составное	Составное	Составное
8481	Составное	Простое	Простое	Составное
10585	Составное	Составное	Составное	Составное
12403	Составное	Составное	Составное	Составное
15457	Составное	Составное	Составное	Составное
15841	Простое	Простое	Простое	Составное

Подсчитав количество ошибок, была сделана столбчатая диаграмма, представленная на рисунке 20.

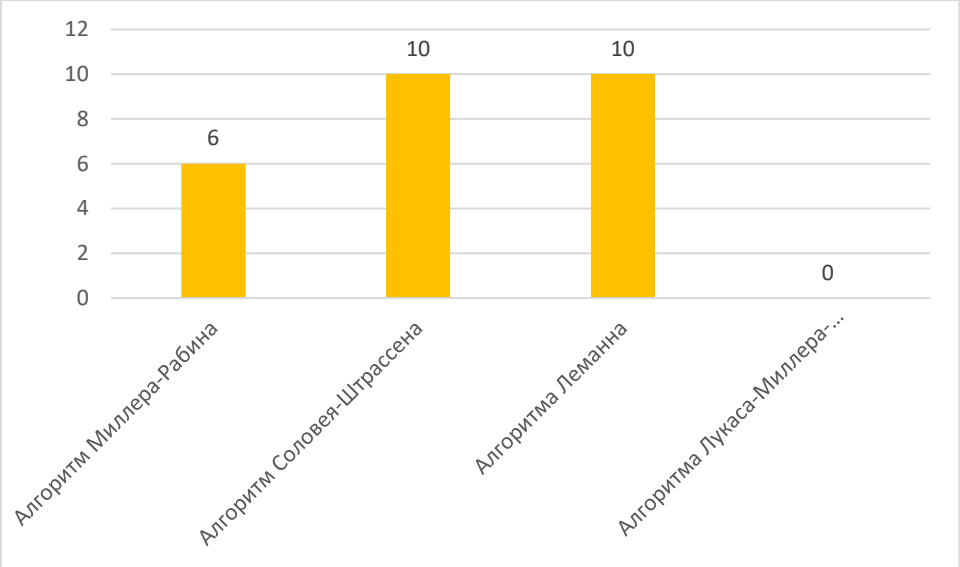

Рисунок 20 – Результат тестирования алгоритмов Эйлера – Якоби – псевдопростыми числами

Далее была проверка алгоритмов числами Вудала. Простые числа Вудала – это простые числа вида $W_n = n \cdot 2^n - 1$. Не для всякого числа n, число Вудала является простым. Но является возможным и проведение проверки на выявление составных чисел Вудала. В таблице 7 приведены значения n и то, каким числом Вудала является число, если подставить в него выбранное n.

Таблица 7 – Простые числа Вудала для n от 3 до 15

n	$W_n = n \cdot 2^n - 1$	Простое или составное
3	23	Простое
4	63	Составное
5	159	Составное
6	383	Простое
7	895	Составное
8	2047	Составное
9	4607	Составное
10	10239	Составное

Продолжение 7

n	$W_n = n \cdot 2^n - 1$	Простое или составное
11	22527	Составное
12	49151	Составное
13	106495	Составное
14	229375	Составное

С числами Вудала справились все алгоритмы.

Также алгоритмы тестировались сильно псевдопростыми числами. Это те составные числа, которые проходят проверку сильными тестами на простоту. Формально его можно описать следующим образом. Число $n = 2^s \cdot d + 1$ с нечетным d по основанию s называется сильно псевдопростым, если выполняется одно из двух условий:

1) $a^d \equiv 1 \, mod \, n$,

2) $a^{d \cdot 2r} \equiv -1 \, mod \, n$ для $0 \leq r \leq s$.

В таблице 8 приведены сильно псевдопростые числа по основанию 2 и ответ алгоритмов при вводе этих чисел.

Таблица 8 – Результат тестирования алгоритмов сильно псевдопростыми числами с основанием 2

Число	Алгоритм Миллера–Рабина	Алгоритм Соловея–Штрассена	Алгоритм Леманна	Алгоритм Лукаса–Миллера–Рабина
2047	Простое	Простое	Простое	Составное
3277	Простое	Простое	Простое	Составное
4033	Простое	Простое	Простое	Составное
4681	Простое	Простое	Простое	Составное
8321	Простое	Простое	Простое	Составное
15841	Простое	Простое	Простое	Составное

Продолжение таблицы 8

Число	Алгоритм Миллера–Рабина	Алгоритм Соловея–Штрассена	Алгоритм Леманна	Алгоритм Лукаса–Миллера–Рабина
29341	Простое	Простое	Простое	Составное
42799	Простое	Простое	Простое	Составное
49141	Простое	Простое	Простое	Составное
52633	Простое	Простое	Простое	Составное
65281	Простое	Простое	Простое	Составное
74665	Составное	Составное	Составное	Составное
80581	Простое	Простое	Простое	Составное
85489	Простое	Простое	Простое	Составное
88357	Простое	Простое	Простое	Составное
90751	Простое	Простое	Простое	Составное

По результатам видно, что вероятностные алгоритмы не справляются с сильно псевдопростыми числами. В то же время тест Лукаса – Миллера – Рабина справился без единой ошибки.

4.6. Анализ проведенного тестирования

Анализируя полученные результаты, можно прийти к следующему выводу: алгоритм Лукаса – Миллера – Рабина является наиболее оптимальным среди выбранных алгоритмов. Время его работы такое же, как время работы теста Миллера – Рабина. Этот алгоритм справляется за 1 итерацию, чего не могут добиться другие алгоритмы. Он также спокойно справляется с числами Вудала, Кармайкла, Эйлера – Якоби – псевдопростыми и сильно псевдопростыми. Стоит отметить, что этот алгоритм запускается однократно, но этот недостаток исправим, если строить различные бинарные последовательности или менять начальную пару значений в ряде Фибоначчи.

Каждый разработчик криптосистем сам выбирает алгоритм, который ему будет удобно использовать, ведь для любой системы ценно что-то определённое. Где-то необходима скорость выполнения, где-то – высокая точность. Но лучше использовать вероятностные алгоритмы высокой точности, они быстрее детерминированных и вероятность, благодаря которой они считают число простым, достаточно велика.

ЗАКЛЮЧЕНИЕ

В ходе выпускной квалификационной работы была реализована программа, в которой возможно самостоятельно протестировать и сравнить комбинированный тест простоты Лукаса – Миллера – Рабина и другие вероятностные тесты: тест Леманна, алгоритм Соловея – Штрассена и тест Миллера – Рабина. Также данное приложение позволяет проверить одно большое число на простоту сразу четырьмя алгоритмами.

Алгоритмы реализованы на языке C# с интуитивно понятным пользовательским интерфейсом на платформе Windows Forms с помощью среды разработки Microsoft Visual Studio.

В ходе работы были достигнуты следующие результаты:

1) изучены методы поиска простых чисел,

2) реализовано приложение с возможностью поиска простых чисел различными алгоритмами,

3) протестированы тесты простоты на различных интервалах и получен график времени их работы,

4) проведён анализ корректности выполнения каждого из алгоритмов,

5) алгоритмы протестированы с помощью различных псевдопростых чисел,

6) выявлены преимущества и недостатки теста Лукаса – Миллера – Рабина.

Можно выделить следующие преимущества алгоритма Лукаса – Миллера – Рабина:

– быстрая скорость выполнения алгоритма;

– высокая точность на одной итерации относительно других представленных алгоритмов;

– устойчивость алгоритма к числам Кармайкла, Вудала, Эйлера – Якоби – псевдопростым и сильно псевдопростым.

Таким образом, проанализировав выполненные задачи, можно сделать вывод, что цель была достигнута и алгоритм Лукаса – Миллера – Рабина был исследован по различным критериям.

За период выполнения выпускной квалификационной работы были приобретены следующие компетенции, указанные в таблице 9.

Таблица 9 – Приобретенные компетенции

Шифр компетенции	Расшифровка проверяемой компетенции	Расшифровка освоения компетенции
ОК-1	способность использовать основы философских знаний для формирования мировоззренческой позиции	способность сформулировать цели и задачи работы
ОК-2	способность использовать основы экономических знаний в различных сферах деятельности	применение экономических навыков для увеличения конкурентоспособности информационной системы
ОК-3	способность анализировать основные этапы и закономерности исторического развития России, ее место и роль в современном мире для формирования гражданской позиции и развития патриотизма	разработка современного продукта и продвижение его по мировому рынку
ОК-4	способность использовать основы правовых знаний в различных сферах деятельности	способность анализировать и учитывать Федеральные законы
ОК-5	способность понимать социальную значимость своей будущей профессии, обладать высокой мотивацией к выполнению профессиональной деятельности в области обеспечения информационной безопасности и защиты интересов личности, общества и государства, соблюдать нормы профессиональной этики	желание развивать профессиональные компетенции и повышать уровень квалификации с помощью получения новых знаний
ОК-6	способность работать в коллективе, толерантно воспринимая социальные, культурные и иные различия	работа вместе с научным руководителем, принятие поправок и замечаний
ОК-7	способность к коммуникации в устной и письменной формах на русском и иностранном языках для решения задач межличностного и межкультурного взаимодействия, в том числе в сфере профессиональной деятельности	способность воспринимать все замечания и поправки, читать и анализировать литературу на английском языке

Продолжение таблицы 9

Шифр компетенции	Расшифровка проверяемой компетенции	Расшифровка освоения компетенции
ОК-8	способность к самоорганизации и самообразованию	умение планировать этапы выполнения выпускной квалификационной работы, навыки самостоятельного изучения материала по заданной тематике
ОК-9	способность использовать методы и средства физической культуры для обеспечения полноценной социальной и профессиональной деятельности	использование методов физической культуры для организации полноценного режима работы
ОПК-1	способность анализировать физические явления и процессы для решения профессиональных задач	способность соотнести физические закономерности с профессиональными задачами
ОПК-2	способность применять соответствующий математический аппарат для решения профессиональных задач	умение понимать научную литературу о нейронных сетях, содержащую в себе математические термины, и использовать приобретенные знания на практике
ОПК-3	способность применять положения электротехники, электроники и схемотехники для решения профессиональных задач	применение персонального компьютера для реализации программного продукта информационной системы
ОПК-4	способность понимать значение информации в развитии современного общества, применять информационные технологии для поиска и обработки информации	овладение инструментами и навыками поиска необходимой информации в сети Интернет, а также умение ее систематизировать и использовать в рамках ВКР
ОПК-5	способность использовать нормативные правовые акты в профессиональной деятельности	применение и анализ Федеральных законов для осуществления информационной безопасности
ОПК-6	способность применять приемы оказания первой помощи, методы и средства защиты персонала предприятия и населения в условиях чрезвычайных ситуаций, организовать мероприятия по охране труда и технике безопасности	осуществление правильного использования персонального компьютера без вреда для здоровья

Продолжение таблицы 9

Шифр компетенции	Расшифровка проверяемой компетенции	Расшифровка освоения компетенции
ОПК-7	способность определять информационные ресурсы, подлежащие защите, угрозы безопасности информации и возможные пути их реализации на основе анализа структуры и содержания информационных процессов и особенностей функционирования объекта защиты	умение находить слабо защищенные части web-приложения, а также определять ряд возможных угроз, которым они могут подвергнуться
ПК-1	способность выполнять работы по установке, настройке и обслуживанию программных, программно-аппаратных (в том числе криптографических) и технических средств защиты информации	навыки, полученные в результате добавления и настройке дополнительных средств защиты в информационной системе
ПК-2	способность применять программные средства системного, прикладного и специального назначения, инструментальные средства, языки и системы программирования для решения профессиональных задач	навыки создания клиентской и серверной части информационной системы, а также добавления к ней элементов защиты информации
ПК-3	способность администрировать подсистемы информационной безопасности объекта защиты	комплекс работ по поддержанию работоспособности и актуальности информационного продукта
ПК-4	способность участвовать в работах по реализации политики информационной безопасности, применять комплексный подход к обеспечению информационной безопасности объекта защиты	определение требований к элементам защиты интерактивной системы, а также их реализация
ПК-5	способность принимать участие в организации и сопровождении аттестации объекта информатизации по требованиям безопасности информации	проверка защищенности интерактивной системы, применение необходимых средств защиты
ПК-6	способность принимать участие в организации и проведении контрольных проверок работоспособности и эффективности применяемых программных, программно-аппаратных и технических средств защиты информации	проведение тестирования интерактивной системы

Продолжение таблицы 9

Шифр компетенции	Расшифровка проверяемой компетенции	Расшифровка освоения компетенции
ПК-7	способность проводить анализ исходных данных для проектирования подсистем и средств обеспечения информационной безопасности и участвовать в проведении технико-экономического обоснования соответствующих проектных решений	способность на основе анализа интерактивной системы определять средства защиты информационной безопасности, а также объяснять свой выбор
ПК-8	способность оформлять рабочую техническую документацию с учетом действующих нормативных и методических документов	оформление отчета для ВКР с учетом действующих нормативных и методических документов
ПК-9	способность осуществлять подбор, изучение и обобщение научно-технической литературы, нормативных и методических материалов, составлять обзор по вопросам обеспечения информационной безопасности по профилю своей профессиональной деятельности	навыки поиска и изучения научно-технической литературы по созданию интерактивных систем, ее элементов безопасности, а также симметричным криптографическим алгоритмам
ПК-10	способность проводить анализ информационной безопасности объектов и систем на соответствие требованиям стандартов в области информационной безопасности	анализ соответствующих законов для проверки необходимых средств защиты информации в информационной системе
ПК-11	способность проводить эксперименты по заданной методике, обработку, оценку погрешности и достоверности их результатов	способность тестировать корректность работы разработанных алгоритмов
ПК-12	способность принимать участие в проведении экспериментальных исследований системы защиты информации	умение искать и исправлять ошибки в готовом исходном коде
ПК-13	способность принимать участие в формировании, организовывать и поддерживать выполнение комплекса мер по обеспечению информационной безопасности, управлять процессом их реализации	применение средств защиты информации при создании информационной системы
ПК-14	способность организовывать работу малого коллектива исполнителей в профессиональной деятельности	организация рабочего процесса по аналитике, проектированию и разработке информационной системы

Продолжение таблицы 9

Шифр компетенции	Расшифровка проверяемой компетенции	Расшифровка освоения компетенции
ПК-15	способность организовать технологический процесс защиты информации ограниченного доступа в соответствии с нормативными правовыми актами и нормативными методическими документами Федеральной службы безопасности Российской Федерации, Федеральной службы по техническому и экспортному контролю	защита пользовательских данных при помощи симметричных алгоритмов шифрования

СПИСОК ЛИТЕРАТУРЫ

1) Василенко О. Н. Теоретико-числовые алгоритмы в криптографии / Василенко О.Н. — Москва: МЦНМО, 2003. — 328 с. – ISBN 5940571034 – Текст : непосредственный.

2) Crandall R. The prime numbers: a computational perspertive / R. Crandall, C. Pomerance. – sec.ed. Springer–Verlag, Berlin, 2005. – 604 с. – ISBN: 0387252827 – Текст : непосредственный.

3) Rabin M.O. Probabilistic algorithm for testing primality / Rabin M.O. – Текст : непосредственный // Journal of Number Theory. – Т.12, вып. 1 – с. 128–138.

4) Аграновский А.В. Практическая криптография: алгоритмы и их программирование / А.В. Аграновский, Р.А. Хади. – Москва: Солон-Пресс, 2009. – 256 с. – ISBN 5-98003-002-6 – Текст : непосредственный.

5) Ишмухаметов, Ш. Т. On the Number of Witnesses in the Miller–Rabin Primality Test / Ш. Т. Ишмухаметов, Б. Г. Мубараков, Р. Г. Рубцова. – DOI 10.3390/sym12060890. – Текст : электронный // Symmetry. – 2020. – № 12(6). – С. 890-902. – URL: https://www.mdpi.com/2073-8994/12/6/890 (дата обращения: 11.11.2022).

6) Ишмухаметов Ш.Т. Об одном тесте простоты натуральных чисел / Ш.Т. Ишмухаметов, Р.Г. Рубцова, Р.Р. Хуснутдинов – Текст : непосредственный // Известия высших учебных заведений. Математика. – 2022. – №2. – с. 83-87.

7) Шнайер, Б. Прикладная криптография. Протоколы, алгоритмы и исходный код на С : [монография] / Б. Шнайер. – 2-е издание. – Москва: Вильямс, 2016. – 28 с. – ISBN 978-5-9908462-4-1. – Текст : непосредственный.

8) Простые числа: Математик Джеймс Мэйнард о теореме Евклида, гипотезе Римана и современных исследованиях тайн простых чисел [Электронный ресурс]. —2016. — URL: https://habr.com/ru/post/191240/ (дата обращения 3.05.2023).

9) Дикарев, С. С. Исследование алгоритмов генерации простых чисел / С. С. Дикарев, Е. Н. Рябухо, Т. В. Турка. — Текст : непосредственный // Молодой ученый. — 2015. — № 10. — С. 6-9. — URL: https://moluch.ru/archive/90/18929/ (дата обращения: 06.04.2023).

10) Росанова, К. А. Эти сложные простые числа! / К. А. Росанова, Я. О. Воронцова, А. М. Гаврилова, О. В. Шмелева. — Текст : непосредственный // Юный ученый. — 2016. — № 6.1. — С. 40-41. — URL: https://moluch.ru/young/archive/9/625/ (дата обращения: 14.03.2023).

ПРИЛОЖЕНИЕ

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Diagnostics;
using System.Numerics;
using System.IO;

namespace VKR
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        //Функция быстрого возведения в степень по модулю
        public static BigInteger powmod(BigInteger a, BigInteger
b, BigInteger n)
        {
            if (b == 0)
                return 1;
            BigInteger c = powmod(a, b / 2, n);
            if (b % 2 == 0)
            {
                return (c * c) % n;
            }
            else
            {
                return (a * c * c) % n;
            }
        }
        //Функция генерации всех чисел между двумя числа
определенного количества бит
        public static List<int> GenerateAllNumbers(int
bitCount1, int bitCount2)
        {
            int minNumber = (int)Math.Pow(2, bitCount1 - 1);
            int maxNumber = (int)Math.Pow(2, bitCount2) - 1;
            List<int> nums = new List<int>();
            for (int i = minNumber; i <= maxNumber; i++)
            {
                nums.Add(i);
            }
            return nums;
```

```
        }
        //Функция для получения эталонных простых чисел из
интервала
        public static List<int> Prime_numbers(int num1, int
num2)
        {
            string filePath =
"E:/educ/дипломушка/кодик/primes.txt";
            string content = File.ReadAllText(filePath);
            string[] nnumbers = content.Replace("[",
"").Replace("]", "").Split(',');
            List<int> array_of_all_numbers = new List<int>();

            for (int i = 0; i < nnumbers.Length; i++)
            {
                if (int.TryParse(nnumbers[i].Trim(), out int
value))
                {
                    if (value >= num1 && value <= num2)
                    { array_of_all_numbers.Add(value); }
                }
            }
            return array_of_all_numbers;
        }

        //Функция поиска наибольшего общего делителя
        public static BigInteger GCD(BigInteger x, BigInteger y)
        {
            return y == 0 ? x : GCD(y, x % y);
        }
        //Функция нахождения символа Якоби
        public static int Jacobi(BigInteger a, BigInteger b)
        {
            if (GCD(a, b) != 1) { return 0; }
            int r = 1;
            if (a < 0)
                a = -a;
            if (b % 4 == 3) { r = -r; }
            while (a != 0)
            {
                int t = 0;
                while (a % 2 == 0)
                {
                    t++;
                    a /= 2;
                }
                if (t % 2 != 0)
                {
                    if (b % 8 == 3 || b % 8 == 5) { r = -r; }
                }
                if (a % 4 == 3 && b % 4 == 3) { r = -r; }
                BigInteger c = a;
                a = b % c;
```

```
                b = c;
            }
            return r;
        }
        //вспомогательная функция для нахождения символа Якоби
        private static int Jacobi_Exp(BigInteger n)
        {
            if (n % 2 == 0)
                return 1;
            return -1;
        }
        //Функция нахождения символа Лежандра
        private static int Legendre_symbol(BigInteger a,
BigInteger n)
        {
            if (a == 0)
                return 0;
            if (a < 0)
                return Legendre_symbol(a *= 1, n) *
Jacobi_Exp((n - 1) / 2);
            if (a % 2 == 0)
                return Legendre_symbol(a / 2, n) *
Jacobi_Exp((BigInteger.Pow(n, 2) - 1) / 8);
            if (a == 1)
                return 1;
            if (a < n)
                return Legendre_symbol(n, a) * Jacobi_Exp((a -
1) / 2 * (n - 1) / 2);
            return Legendre_symbol(a % n, n);
        }
        // Функция произведения матриц типа BigInteger
        private static BigInteger[,]
MatrixMultiply(BigInteger[,] a, BigInteger[,] b)
        {
            BigInteger[,] r = new BigInteger[a.GetLength(0),
b.GetLength(1)];
            for (int i = 0; i < a.GetLength(0); i++)
            {
                for (int j = 0; j < b.GetLength(1); j++)
                {
                    for (int k = 0; k < b.GetLength(0); k++)
                    {
                        r[i, j] += a[i, k] * b[k, j];
                    }
                }
            }
            return r;
        }

        //Вычисление n-ого члена Фибоначчи
        private static BigInteger Fibonacci(BigInteger n)
        {
            if (n == 0)
```

```
                return 0;
            else if (n == 1 || n == 2)
                return 1;
            else
            {
                BigInteger[,] a = new BigInteger[,] { { 1, 1 },
{ 1, 0 } };
                BigInteger[,] b = new BigInteger[,] { { 1, 1 },
{ 1, 0 } };
                for (int i = 1; i < n; i++)
                {
                    a = MatrixMultiply(a, b);
                }
                return a[0, 1];
            }
        }
        //Функция теста Миллера-Рабина
        public static bool Miller_Rabin(BigInteger number, int
iterations)
        {
            BigInteger t = number - 1;
            int s = 0;
            while (t % 2 == 0)
            {
                t /= 2;
                s += 1;
            }
            for (int j = 0; j < iterations; j++)
            {

                RNGCryptoServiceProvider rngCSP = new
RNGCryptoServiceProvider();
                byte[] c = new
byte[number.ToByteArray().LongLength];
                BigInteger a;
                do
                {
                    rngCSP.GetBytes(c);
                    a = new BigInteger(c);
                } while (a < 2 || a > number - 2);

                /* версия для реализации определенной базы
алгоритма:
                List<BigInteger> a_numbers = new
List<BigInteger>() { 2,3,5,7,11,13,17,19,23,29};
                BigInteger a = a_numbers[j];
                */
                BigInteger x = powmod(a, t, number);
                if (x == 1 || x == number - 1) { continue; }
                for (int i = 0; i < s-1; i++)
                {
                    x = powmod(x, 2, number);
                    if (x == 1) { return false; }
```

```
                    if (x == number - 1) { break; }

                }
                if (x != number - 1)
                    return false;
            }
            return true;
        }
        //Функция теста Лукаса-Миллера-Рабина
        public static bool Lucas_Miller_Rabin(BigInteger number)
        {
            if (number > 5)
            {
                int e = Jacobi(number, 5);
                BigInteger t = number - e;
                BigInteger s = 0;
                while (t % 2 == 0)
                {
                    t /= 2;
                    s += 1;
                }
                BigInteger Ft = Fibonacci(t);
                if (Ft % number == 0) return true;
                for (int i = 0; i < s; i++)
                {
                    BigInteger m = BigInteger.Pow(2, i) * t;
                    if ((Fibonacci(m - 1) + Fibonacci(m + 1)) %
number == 0) return true;
                }
                return false;
            }
            else
            {
                if (number == 2 || number == 3 || number == 5)
                    return true;
                else return false;
            }
        }
        //Функция теста Соловея-Штрассена
        public static bool Solov_Shtrass(BigInteger number, int
iterations)
        {
            if (number <= 2)
                return false;
            int i = 0;
            while (i < iterations)
            {
                /* версия для реализации определенной базы
алгоритма:
                List<BigInteger> a_numbers = new
List<BigInteger>() { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
                BigInteger a = a_numbers[i];
                */
```

```
                RNGCryptoServiceProvider rngCSP = new
RNGCryptoServiceProvider();
                byte[] c = new
byte[number.ToByteArray().LongLength];
                BigInteger a;
                do
                {
                    rngCSP.GetBytes(c);
                    a = new BigInteger(c);
                } while (a < 2 || a > number - 2);

                if (GCD(a, number) > 1) { return false; }
                if (powmod(a, (number - 1) / 2, number) !=
Legendre_symbol(a, number) && powmod(a, (number - 1) / 2,
number) != Legendre_symbol(a, number) + number)
                { return false; }
                i += 1;
            }
            return true;
        }
        //Функция теста Леманна
        public static bool Lemann(BigInteger number, int
iterations)
        {
            for (int i = 0; i < iterations; i++)
            {
                /* версия для реализации определенной базы
алгоритма:
                List<BigInteger> a_numbers = new
List<BigInteger>() { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
                BigInteger a = a_numbers[i];
                */

                RNGCryptoServiceProvider rngCSP = new
RNGCryptoServiceProvider();
                byte[] c = new
byte[number.ToByteArray().LongLength];
                BigInteger a;
                do
                {
                    rngCSP.GetBytes(c);
                    a = new BigInteger(c);
                } while (a < 2 || a > number - 1);

                if (GCD(a, number) == 1)
                {
                    BigInteger k = (number - 1) / 2;
                    BigInteger r = powmod(a, k, number);
                    if (r != 1 && r != number - 1) { return
false; }
                }
            }
```

```
            return true;
        }
        //Функция для кнопки, выполняющая поиск простых чисел
алгоритмом Миллера-Рабина
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                if (textBox1.Text == "" || textBox2.Text == "")
                {
                    MessageBox.Show("Не введены значения!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                else
                {
                    int num1 = int.Parse(textBox1.Text);
                    int num2 = int.Parse(textBox2.Text);
                    bool flag = false;
                    if (num1 >= num2) { MessageBox.Show("Конец
интервала не больше начала!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                    else
                    {
                        if (num1 < 4) { MessageBox.Show("Введено
слишком маленькое значение для начала интервала", "Ошибка",
MessageBoxButtons.OK, MessageBoxIcon.Error); }
                        else
                        {
                            if (num1 > 20) {
MessageBox.Show("Введено слишком большое значение для начала
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                            else
                            {
                                if (num2 < 5) {
MessageBox.Show("Введено слишком маленькое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                else
                                {
                                    if (num2 > 30) {
MessageBox.Show("Введено слишком большое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                    else
                                    {
                                        flag = true;
                                    }
                                }
                            }
                        }
                    }
                    if (flag)
```

```
                    {
                        List<int> numbers =
GenerateAllNumbers(num1, num2); //все числа интервала
                        List<int> true_prime =
Prime_numbers((int)numbers[0], (int)numbers[numbers.Count - 1]);
//эталонные простые
                        List<int> arr = new List<int>();
//массив для найденных простых чисел
                        List<int> arrmis1 = new List<int>();
//массив для чисел, которые алгоритм посчитал простыми, а они
такими не явл
                        List<int> arrmis2 = new List<int>();
//массив для чисел, которые алгоритм пропустил
                        int itter = int.Parse(textBox12.Text);

                        Stopwatch stopWatch = new Stopwatch();
                        stopWatch.Start();
                        for (int i = 0; i < numbers.Count; i++)
                        {
                            if (numbers[i] % 2 != 0 &&
Miller_Rabin(numbers[i], itter))
                            {
                                arr.Add(numbers[i]);
                            }
                        }
                        stopWatch.Stop();

                        IEnumerable<int> c =
true_prime.Except(arr);
                        arrmis1 = new List<int>(c);
                        IEnumerable<int> c1 =
arr.Except(true_prime);
                        arrmis2 = new List<int>(c1);

                        textBox3.Text = "Всего найдено " +
arr.Count + " простых чисел. Среди них " + arrmis1.Count + "
ошибочно отнесены к простым:";
                        foreach (int item in arrmis1)
                        {
                            textBox3.Text += " " + item;
                        }
                        textBox3.Text += ". Не найдено " +
arrmis2.Count + " простых чисел:";
                        foreach (int item in arrmis2)
                        {
                            textBox3.Text += " " + item;
                        }
                        textBox3.Text += ". Найденные простые
числа: ";
                        foreach (int item in arr)
                        {
                            textBox3.Text += " " + item;
                        }
```

```
                        TimeSpan ts = stopWatch.Elapsed;
                        string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                            ts.Hours, ts.Minutes, ts.Seconds,
                            ts.Milliseconds / 10);
                        textBox4.Text = elapsedTime;
                    }
                }
            }
            catch (FormatException)
            {
                MessageBox.Show("Введите значения правильно!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            catch (OverflowException)
            {
                MessageBox.Show("Введены слишком большие
значения!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }
        }
        //Функция для кнопки, выполняющая поиск простых чисел
алгоритмом Леманна
        private void button3_Click(object sender, EventArgs e)
        {
            try
            {
                if (textBox1.Text == "" || textBox2.Text == "")
                {
                    MessageBox.Show("Не введены значения!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                else
                {
                    int num1 = int.Parse(textBox1.Text);
                    int num2 = int.Parse(textBox2.Text);
                    bool flag = false;
                    if (num1 >= num2) { MessageBox.Show("Конец
интервала не больше начала!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                    else
                    {
                        if (num1 < 4) { MessageBox.Show("Введено
слишком маленькое значение для начала интервала", "Ошибка",
MessageBoxButtons.OK, MessageBoxIcon.Error); }
                        else
                        {
                            if (num1 > 20) {
MessageBox.Show("Введено слишком большое значение для начала
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                            else
```

```
                            {
                                if (num2 < 5) {
MessageBox.Show("Введено слишком маленькое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                else
                                {
                                    if (num2 > 30) {
MessageBox.Show("Введено слишком большое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                    else
                                    {
                                        flag = true;
                                    }
                                }
                            }
                        }
                    }
                    if (flag)
                    {
                        List<int> numbers =
GenerateAllNumbers(num1, num2); //все числа интервала
                        List<int> true_prime =
Prime_numbers((int)numbers[0], (int)numbers[numbers.Count - 1]);
//эталонные простые
                        List<int> arr = new List<int>();
//массив для найденных простых чисел
                        List<int> arrmis1 = new List<int>();
//массив для чисел, которые алгоритм посчитал простыми, а они
такими не явл
                        List<int> arrmis2 = new List<int>();
//массив для чисел, которые алгоритм пропустил
                        int itter = int.Parse(textBox14.Text);

                        Stopwatch stopWatch = new Stopwatch();
                        stopWatch.Start();
                        for (int i = 0; i < numbers.Count; i++)
                        {
                            if (numbers[i] % 2 != 0 &&
Lemann(numbers[i], itter))
                            {
                                arr.Add(numbers[i]);
                            }
                        }
                        stopWatch.Stop();

                        IEnumerable<int> c =
true_prime.Except(arr);
                        arrmis1 = new List<int>(c);
                        IEnumerable<int> c1 =
arr.Except(true_prime);
                        arrmis2 = new List<int>(c1);
```

```
                        textBox7.Text = "Всего найдено " +
arr.Count + " простых чисел. Среди них " + arrmis1.Count + "
ошибочно отнесены к простым:";
                        foreach (int item in arrmis1)
                        {
                            textBox7.Text += " " + item;
                        }
                        textBox7.Text += ". Не найдено " +
arrmis2.Count + " простых чисел:";
                        foreach (int item in arrmis2)
                        {
                            textBox7.Text += " " + item;
                        }
                        textBox7.Text += ". Найденные простые
числа: ";
                        foreach (int item in arr)
                        {
                            textBox7.Text += " " + item;
                        }

                        TimeSpan ts = stopWatch.Elapsed;
                        string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                            ts.Hours, ts.Minutes, ts.Seconds,
                            ts.Milliseconds / 10);
                        textBox8.Text = elapsedTime;
                    }
                }
            }
            catch (FormatException)
            {
                MessageBox.Show("Введите значения правильно!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            catch (OverflowException)
            {
                MessageBox.Show("Введены слишком большие
значения!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }
        }
        //Функция для кнопки, выполняющая поиск простых чисел
алгоритмом Соловея-Штрассена
        private void button2_Click_1(object sender, EventArgs e)
        {
            try
            {
                if (textBox1.Text == "" || textBox2.Text == "")
                {
                    MessageBox.Show("Не введены значения!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
```

```
                else
                {
                    int num1 = int.Parse(textBox1.Text);
                    int num2 = int.Parse(textBox2.Text);
                    bool flag = false;
                    if (num1 >= num2) { MessageBox.Show("Конец
интервала не больше начала!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                    else
                    {
                        if (num1 < 4) { MessageBox.Show("Введено
слишком маленькое значение для начала интервала", "Ошибка",
MessageBoxButtons.OK, MessageBoxIcon.Error); }
                        else
                        {
                            if (num1 > 20) {
MessageBox.Show("Введено слишком большое значение для начала
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                            else
                            {
                                if (num2 < 5) {
MessageBox.Show("Введено слишком маленькое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                else
                                {
                                    if (num2 > 30) {
MessageBox.Show("Введено слишком большое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                    else
                                    {
                                        flag = true;
                                    }
                                }
                            }
                        }
                    }
                    if (flag)
                    {
                        List<int> numbers =
GenerateAllNumbers(num1, num2); //все числа интервала
                        List<int> true_prime =
Prime_numbers((int)numbers[0], (int)numbers[numbers.Count - 1]);
//эталонные простые
                        List<int> arr = new List<int>();
//массив для найденных простых чисел
                        List<int> arrmis1 = new List<int>();
//массив для чисел, которые алгоритм посчитал простыми, а они
такими не явл
                        List<int> arrmis2 = new List<int>();
//массив для чисел, которые алгоритм пропустил
```

```
                        int itter = int.Parse(textBox13.Text);

                        Stopwatch stopWatch = new Stopwatch();
                        stopWatch.Start();
                        for (int i = 0; i < numbers.Count; i++)
                        {
                            if (numbers[i] % 2 != 0 &&
Solov_Shtrass(numbers[i], itter))
                            {
                                arr.Add(numbers[i]);
                            }
                        }
                        stopWatch.Stop();

                        IEnumerable<int> c =
true_prime.Except(arr);
                        arrmis1 = new List<int>(c);
                        IEnumerable<int> c1 =
arr.Except(true_prime);
                        arrmis2 = new List<int>(c1);

                        textBox5.Text = "Всего найдено " +
arr.Count + " простых чисел. Среди них " + arrmis1.Count + "
ошибочно отнесены к простым:";
                        foreach (int item in arrmis1)
                        {
                            textBox5.Text += " " + item;
                        }
                        textBox5.Text += ". Не найдено " +
arrmis2.Count + " простых чисел:";
                        foreach (int item in arrmis2)
                        {
                            textBox5.Text += " " + item;
                        }
                        textBox5.Text += ". Найденные простые
числа: ";
                        foreach (int item in arr)
                        {
                            textBox5.Text += " " + item;
                        }

                        TimeSpan ts = stopWatch.Elapsed;
                        string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                            ts.Hours, ts.Minutes, ts.Seconds,
                            ts.Milliseconds / 10);
                        textBox6.Text = elapsedTime;
                    }
                }
            }
            catch (FormatException)
            {
```

```
                MessageBox.Show("Введите значения правильно!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            catch (OverflowException)
            {
                MessageBox.Show("Введены слишком большие
значения!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }
        }
        //Функция для кнопки, выполняющая поиск простых чисел
алгоритмом Лукаса-Миллера-Рабина
        private void button4_Click(object sender, EventArgs e)
        {
            try
            {
                if (textBox1.Text == "" || textBox2.Text == "")
                {
                    MessageBox.Show("Не введены значения!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
                }
                else
                {
                    int num1 = int.Parse(textBox1.Text);
                    int num2 = int.Parse(textBox2.Text);
                    bool flag = false;
                    if (num1 >= num2) { MessageBox.Show("Конец
интервала не больше начала!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                    else {
                        if (num1 < 4) { MessageBox.Show("Введено
слишком маленькое значение для начала интервала", "Ошибка",
MessageBoxButtons.OK, MessageBoxIcon.Error); }
                        else
                        {
                            if (num1 > 20) {
MessageBox.Show("Введено слишком большое значение для начала
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                            else
                            {
                                if (num2 < 5) {
MessageBox.Show("Введено слишком маленькое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                else
                                {
                                    if (num2 > 30) {
MessageBox.Show("Введено слишком большое значение для конца
интервала", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error); }
                                    else
                                    {
```

```
                                    flag = true;
                                }
                            }
                        }
                    }
                }
                if (flag)
                {
                    List<int> numbers =
GenerateAllNumbers(num1, num2); //все числа интервала
                    List<int> true_prime =
Prime_numbers((int)numbers[0], (int)numbers[numbers.Count-1]);
//эталонные простые
                    List<int> arr = new List<int>();
//массив для найденных простых чисел
                    List<int> arrmis1 = new List<int>();
//массив для чисел, которые алгоритм посчитал простыми, а они
такими не явл
                    List<int> arrmis2 = new List<int>();
//массив для чисел, которые алгоритм пропустил

                    Stopwatch stopWatch = new Stopwatch();
                    stopWatch.Start();
                    for (int i = 0; i < numbers.Count; i++)
                    {
                        if (numbers[i] % 2 != 0 &&
numbers[i] % 5 != 0 && Lucas_Miller_Rabin(numbers[i]))
                        {
                            arr.Add(numbers[i]);
                        }
                    }
                    stopWatch.Stop();

                    IEnumerable<int> c =
true_prime.Except(arr);
                    arrmis1 = new List<int>(c);
                    IEnumerable<int> c1 =
arr.Except(true_prime);
                    arrmis2 = new List<int>(c1);

                    textBox9.Text = "Всего найдено " +
arr.Count + " простых чисел. Среди них " + arrmis1.Count + "
ошибочно отнесены к простым:";
                    foreach(int item in arrmis1)
                    {
                        textBox9.Text += " " + item;
                    }
                    textBox9.Text += ". Не найдено " +
arrmis2.Count + " простых чисел:";
                    foreach (int item in arrmis2)
                    {
                        textBox9.Text += " " + item;
                    }
```

```
                        textBox9.Text += ". Найденные простые
числа: ";
                        foreach (int item in arr)
                        {
                            textBox9.Text += " " + item;
                        }

                        TimeSpan ts = stopWatch.Elapsed;
                        string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                            ts.Hours, ts.Minutes, ts.Seconds,
                            ts.Milliseconds / 10);
                        textBox10.Text = elapsedTime;
                    }
                }
            }
            catch (FormatException)
            {
                MessageBox.Show("Введите значения правильно!",
"Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
            catch (OverflowException)
            {
                MessageBox.Show("Введены слишком большие
значения!", "Ошибка", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }
        }
        //Функция для кнопки, выполняющая проверку числа каждым
алгоритмом
        private void button5_Click(object sender, EventArgs e)
        {
            Stopwatch stopWatch = new Stopwatch();
            BigInteger number = int.Parse(textBox11.Text);

            int it1 = int.Parse(textBox12.Text);
            int it2 = int.Parse(textBox13.Text);
            int it3 = int.Parse(textBox14.Text);

            stopWatch.Start();
            if (Lucas_Miller_Rabin(number) && number % 5 != 0)
            { textBox9.Text = "PRIME"; }
            else { textBox9.Text = "NOT PRIME"; }
            stopWatch.Stop();

            TimeSpan ts = stopWatch.Elapsed;
            string elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts.Hours, ts.Minutes, ts.Seconds,
                ts.Milliseconds / 10);
            textBox10.Text = elapsedTime;

            Stopwatch stopWatch1 = new Stopwatch();
```

```
            stopWatch1.Start();
            if (Miller_Rabin(number, it1) && number % 5 != 0)
            { textBox3.Text = "PRIME"; }
            else { textBox3.Text = "NOT PRIME"; }
            stopWatch1.Stop();

            TimeSpan ts1 = stopWatch1.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts1.Hours, ts1.Minutes, ts1.Seconds,
                ts1.Milliseconds / 10);
            textBox4.Text = elapsedTime;

            Stopwatch stopWatch2 = new Stopwatch();
            stopWatch2.Start();
            if (Lemann(number, it3) && number % 5 != 0)
            { textBox7.Text = "PRIME"; }
            else { textBox7.Text = "NOT PRIME"; }
            stopWatch2.Stop();

            TimeSpan ts2 = stopWatch2.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts2.Hours, ts2.Minutes, ts2.Seconds,
                ts2.Milliseconds / 10);
            textBox8.Text = elapsedTime;

            Stopwatch stopWatch3 = new Stopwatch();
            stopWatch3.Start();
            if (Solov_Shtrass(number, it2) && number % 5 != 0)
            { textBox5.Text = "PRIME"; }
            else { textBox5.Text = "NOT PRIME"; }
            stopWatch3.Stop();

            TimeSpan ts3 = stopWatch3.Elapsed;
            elapsedTime =
String.Format("{0:00}:{1:00}:{2:00}.{3:00}",
                ts3.Hours, ts3.Minutes, ts3.Seconds,
                ts3.Milliseconds / 10);
            textBox6.Text = elapsedTime;
        }
    }
}
```