Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет математики и информационных технологий

Кафедра алгебры и дискретной математики

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Направление подготовки 02.03.01 Математика и компьютерные науки

**Улучшение алгоритма кольцевой факторизации
для построения простых чисел**

ОГУ 02.03.01. 1319. 022 ОО

Заведующий кафедрой
канд. физ.-мат. наук, доцент — О.А. Пихтилькова

Руководитель
канд. пед. наук, доцент — Л.Б. Усова

Студент — А.Д. Петин

Оренбург 2019

Утверждаю
заведующий кафедрой алгебры и дискретной математики
_____ О.А. Пихтилькова
«01» февраля 2019 г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

студенту Петину Алексею Дмитриевичу

по направлению подготовки 02.03.01 Математика и компьютерные науки

1 Тема ВКР Улучшение алгоритма кольцевой факторизации для построения простых чисел.
2 Срок сдачи студентом ВКР «17» июня 2019 г.
3 Цель и задачи ВКР:
Цель – реализовать модификацию алгоритма кольцевой факторизации для построения простых чисел на основе решета Эратосфена.
Задачи: осуществить теоретический анализ различных подходов к поиску простых чисел, выбрать и реализовать наиболее эффективные алгоритмы поиска последовательных простых чисел, реализовать алгоритм кольцевой факторизации на основе решета Эратосфена, улучшить алгоритм кольцевой факторизации, провести сравнительный анализ эффективности реализованных алгоритмов поиска последовательных простых чисел.
4 Исходные данные к ВКР: статьи, теоретико-числовые результаты в области построения больших простых чисел.
5 Перечень вопросов, подлежащих разработке: осуществить обзор современной литературы по теме исследования, описать некоторые существующие алгоритмы просеивания последовательных простых чисел, реализовать выбранные алгоритмы, описать методы модификации алгоритмов просеивания простых чисел, реализовать модификацию алгоритма кольцевой факторизации на основе решета Эратосфена, провести сравнительный анализ эффективности реализованных алгоритмов поиска последовательных простых чисел по памяти и времени работы.
6 Перечень графического (иллюстративного) материала: рисунки и таблицы, необходимые для изложения материала.

Дата выдачи и получения задания

Руководитель ВКР «01» февраля 2019 г. Л.Б. Усова

Студент «01» февраля 2019 г. Петин А.Д. Петин

Аннотация

В работе рассмотрены эффективные способы поиска последовательных простых чисел.

Структура данной работы выглядит следующим образом:

– в первой главе рассматриваются теоретические аспекты алгоритмов поиска последовательных простых чисел, рассматривается алгоритм кольцевой факторизации и другие способы модификации алгоритмов просеивания простых чисел;

– вторая глава посвящена вопросам программной реализации алгоритмов просеивания простых чисел. Особое внимание уделено модификации решета Эратосфена с помощью кольцевой факторизации и других методов, описанных в теоретической главе;

– третья глава посвящена сравнительному анализу реализованных алгоритмов просеивания простых чисел по времени работы и объему потребляемой оперативной памяти.

Работа выполнена на 65 страницах с использованием 16 источников и содержит 30 рисунков, 4 таблицы и 1 приложение.

Annotation

The paper discusses effective ways to search for consecutive primes.

The structure of this work is as follows:

- The first chapter discusses the theoretical aspects of the search algorithms for consecutive primes, considers the wheel factorization algorithm and other ways to modify the sifting algorithms of prime numbers;
- the second chapter is devoted to issues of software implementation of algorithms for sifting prime numbers. Particular attention is paid to the modification of the sieve of Eratosthenes using wheel factorization and other methods described in the theoretical chapter;
- the third chapter is devoted to a comparative analysis of the implemented algorithms for sifting prime numbers by operation time and the amount of RAM consumed.

The work is done 65 pages using 16 sources and contains 30 figures, 4 tables and 1 appendix.

Содержание

Введение ... 6
1 Способы поиска простых чисел ... 8
1.1 Решето Эратосфена ... 8
1.2 Решето Сундарама ... 9
1.3 Решето Аткина ... 10
1.4 Кольцевая факторизация ... 11
1.5 Способы модификации алгоритмов просеивания простых чисел ... 13
2 Программная реализация алгоритмов поиска последовательных простых чисел ... 14
2.1 Выбор программных средств ... 14
2.2 Интерфейс приложения ... 15
2.3 Реализации решета Эратосфена и его модификаций ... 17
2.4 Реализация и улучшение алгоритма сегментированного решета Эратосфена с битовым сжатием, кольцевой факторизацией и параллельным просеиванием ... 24
2.5 Класс измерения времени ... 32
2.6 Реализация решета Сундарама и его оптимизация ... 33
2.7 Реализация решета Аткина и его оптимизация ... 35
3 Сравнительный анализ эффективности реализованных алгоритмов просеивания ... 39
Заключение ... 52
Список использованных источников ... 54
Приложение А ... 55
Результаты тестирования алгоритмов на отрезке от 1 до $2 * 10^9$. Оптимизация выключена ... 55
Результаты тестирования алгоритмов на отрезке от 1 до $2 * 10^9$. Оптимизация включена ... 57
Результаты тестирования алгоритмов на отрезке от 1 до 10^{10} ... 59
Результаты тестирования алгоритмов на отрезке от 1 до 10^{11} ... 60
Результаты тестирования алгоритмов на отрезке $[10^{12}, 10^{12} + 10^{10}]$ ... 62
Результаты тестирования алгоритмов на отрезке $[10^{14}, 10^{14} + 10^{10}]$ ... 63
Результаты тестирования алгоритмов на отрезке $[10^{16}, 10^{16} + 10^{10}]$ ... 64
Результаты тестирования алгоритмов на отрезке $[10^{18}, 10^{18} + 10^{10}]$ ... 65

Введение

Проблема нахождения простых чисел существует с давних времен. Древнегреческий математик Эратосфен первым предложил эффективный алгоритм нахождения всех простых чисел до заданного числа n. В настоящее время существует множество различных алгоритмов просеивания простых чисел. Наиболее известные из них – это решето Сундарама и решето Аткина.

Современная криптография применяет эти алгоритмы с различными модификациями для генерации последовательных простых чисел. Однако, стоит отметить, что в современной криптографии чаще всего используются комбинации или различные модификации алгоритмов.

Практически вся асимметрично-ключевая криптография базируется на некоторых положениях теории чисел, включая теории, связанные с простыми числами. Большие простые числа являются основой для генерации криптографических ключей, которые используются для проведения защищенных транзакций. В частности, они используются для генерации ключей в самой популярной криптографической системе RSA, в криптосистеме Рабина, Эль-Гамаля и других системах шифрования с открытым ключом. Сами алгоритмы шифрования применяются, например, для аутентификации на вебсайтах, использующих цифровые сертификаты. При помощи ассиметричных шифров также реализуется цифровая подпись.

Поэтому развитие методов поиска и построения простых чисел, основанных на эффективных алгоритмах просеивания, имеет огромное фундаментальное и прикладное значение для решения проблем обеспечения информационной безопасности.

На данный момент наибольшим из известных простых чисел является число Мерсенна $M_{82\,589\,933} = 2^{82\,589\,933} - 1$. Оно содержит $24\,862\,048$ десятичных цифр. А самые большие известные простые близнецы – это $2\,003\,663\,613 * 2^{195000} \pm 1$. Они состоят из $58\,711$ цифр. В мире простых чисел существует еще много других удивительных рекордов.

Объект исследования в этой работе – алгоритмы последовательного построения простых чисел.

Предмет исследования – способы улучшения алгоритмов последовательного построения простых чисел.

Цель работы – реализовать и улучшить алгоритм кольцевой факторизации для построения простых чисел на основе решета Эратосфена.

В соответствии с поставленной целью сформулированы следующие задачи:

1. осуществить теоретический анализ различных подходов к поиску простых чисел;

2. выбрать и реализовать наиболее эффективные алгоритмы поиска последовательных простых чисел;

3. реализовать алгоритм кольцевой факторизации на основе решета Эратосфена;

4. улучшить алгоритм кольцевой факторизации;

5. провести сравнительный анализ эффективности реализованных алгоритмов поиска последовательных простых чисел.

Апробация результатов. Результаты проводимой работы докладывались на:

– XLI студенческой научной конференции ОГУ. Секция «Фундаментальная и прикладная математика» (Оренбург, 2-9 апреля 2019 г.);

Публикации. По теме исследования опубликована одна работа ([1] в списке использованных источников).

Личный вклад автора. Результаты, выносимые на защиту, получены автором самостоятельно.

Объем и структура работы. Работа состоит из введения, трех глав, разбитых на подглавы, заключения, списка использованных источников, одного приложения. Определения, теоремы, формулы, примеры, рисунки и таблицы имеют сквозную нумерацию. Полный объем работы 65 страниц, библиография включает 16 наименований.

1 Способы поиска простых чисел

1.1 Решето Эратосфена

Название алгоритма говорит о принципе его работы, то есть решето подразумевает фильтрацию, а в данном случае фильтрацию всех чисел за исключением простых. По мере прохождения списка нужные числа остаются, а ненужные (составные) исключаются [2].

Возьмем массив P длины N по умолчанию заполненный нулями (пометим как не вычеркнутые). Теперь будем последовательно просматривать элементы $P[k]$, начиная с $k = 2$. Если $P[k] = 0$, то просматриваемое число простое. Тогда заполним единицами (то есть вычеркнем) все последующие ячейки массива, номера которых кратны k [3, 4]. В результате получается массив, в котором ячейки содержат 0 тогда и только тогда, когда номер ячейки – простое число.

Можно сэкономить много времени, если останавливать процесс просеивания на $k = \sqrt{N}$, где N – это максимальное число в просеиваемом интервале. Так можно поступить потому, что у составного числа, меньшего n, по крайней мере, один из делителей не превосходит $\sqrt{N}$.

Еще немного операций можно сэкономить, если по той же причине начинать вычеркивать кратные k, начиная не с $2k$, а с номера k^2. Действительно, ведь все составные числа меньшие, чем k^2 были вычеркнуты ранее. Асимптотическая сложность решета Эратосфена оценивается как $O(N \log\log N)$ операций [5]. Причем множитель $\log\log N$ растет крайне медленно. Потребление памяти же составит $O(N)$.

1.2 Решето Сундарама

Алгоритм предусматривает исключение из ряда натуральных чисел от 1 до N всех чисел вида:

$$i + j + 2ij,$$

где индексы $i \leq j$ пробегают все натуральные значения, для которых $i + j + 2ij \leq N$, а именно значения:

$$i = 1, 2, \dots, \left\lfloor \frac{\sqrt{2N + 1} - 1}{2} \right\rfloor,$$

$$j = i, i + 1, \dots, \left\lfloor \frac{N - i}{2i + 1} \right\rfloor.$$

Затем каждое из оставшихся чисел необходимо умножить на 2 и увеличить на единицу. Полученная в результате последовательность представляет собой все простые числа в отрезке $[3,\ 2N + 1]$.

Алгоритм работает с нечетными натуральными числами большими единицы, представленными в виде $2m + 1$, где m является натуральным числом. А если число $2m + 1$ является составным, то его можно представить в виде произведения двух нечетных чисел больших единицы, то есть:

$$2m + 1 = (2i + 1)(2j + 1),$$

где i и j – натуральные числа. Это равносильно соотношению:

$$m = 2ij + i + j.$$

Таким образом, если из ряда натуральных чисел исключить все числа вида $2ij + i + j$ $(1 \leq i \leq j)$, то для каждого из оставшихся чисел m число $2m + 1$ обязательно будет простым. И, наоборот, если число $2m + 1$ является простым, то число m невозможно представить в виде $2ij + i + j$ и, таким образом ни одно простое число (кроме 2) не может быть исключено в процессе работы алгоритма [6].

Асимптотическая сложность и потребление памяти решета Сундарама сопоставимо с решетом Эратосфена.

1.3 Решето Аткина

Алгоритм решета Аткина основан на следующей теореме [7].

Теорема. Пусть n – натуральное число, которое не делится ни на какой полный квадрат. Тогда верно следующее:

1. Если n представимо в виде $4k + 1$, то оно просто тогда и только тогда, когда число натуральных решений уравнения $4x^2 + y^2 = n$ нечетно.

2. Если n представимо в виде $6k + 1$, то оно просто тогда и только тогда, когда число натуральных решений уравнения $3x^2 + y^2 = n$ нечетно.

3. Если n представимо в виде $12k - 1$, то оно просто тогда и только тогда, когда число натуральных решений уравнения $3x^2 - y^2 = n$, для которых
$x > y$, нечетно.

Из элементарной теории чисел следует, что все простые числа, большие 3, имеют вид $12k + 1$ (случай 1), $12k + 5$ (снова 1), $12k + 7$ (случай 2) или $12k + 11$ (случай 3).

Для инициализации алгоритма заполним массив P нулями. Теперь для каждой пары (x, y) , где $0 < x, y < \sqrt{n}$, инкрементируем значения в ячейках $P[4x^2 + y^2]$, $P[3x^2 + y^2]$, а также, если $x > y$, то и в $P[3x^2 - y^2]$. В конце вычислений номера ячеек вида $6k \pm 1$, содержащие нечетные числа, – это или простые, или делятся на квадраты простых.

В качестве заключительного этапа пройдемся по предположительно простым номерам последовательно и вычеркнем кратные их квадратам.

Асимптотическая сложность решета Аткина оценивается как $O(n)$ операций. Потребление памяти так же составит $O(n)$.

1.4 Кольцевая факторизация

Первую оптимизацию решета предложил еще сам Эратосфен. Так как из всех четных чисел простым является только число 2, то можно сэкономить почти половину памяти и времени, если выписывать и высеивать только нечетные числа.

Более развитая оптимизация, основывающаяся на этой идее (так называемая кольцевая факторизация [8, 9] (wheel factorization)) опирается на то, что все простые числа, кроме нескольких первых можно задать одной или несколькими арифметическими прогрессиями. Например:

1. Все простые числа кроме 2 находятся среди членов арифметической прогрессии $a_k = 2k + 1$.
2. Все простые числа кроме 2 и 3 находятся среди членов арифметических прогрессий: $a_k = 6k + 1$ и $b_k = 6k + 5$.
3. Все простые числа кроме 2, 3 и 5 находятся среди членов арифметических прогрессий:

$$a_k = 30k + 1, b_k = 30k + 7, c_k = 30k + 11, d_k = 30k + 13,$$
$$e_k = 30k + 17, f_k = 30k + 19, g_k = 30k + 23, h_k = 30k + 29.$$

Количество исключаемых простых чисел называется порядком кольцевой факторизации. Каждый последующий порядок кольцевой факторизации в сравнении с предыдущим исключает еще $\frac{1}{p}$ долю от всех чисел, где p – это наибольшее исключенное простое число [10, 11]. В общем случае доля оставшихся чисел после применения кольцевой факторизации порядка k составляет:

$$\prod_{i=1}^{k} \frac{p_i - 1}{p_i},$$

где p_i – это -ое простое число. А количество арифметических прогрессий, которые нужно составить:

$$\prod_{i=1}^{k} (p_i - 1).$$

Таблица 1 – Сравнение различных порядков кольцевой факторизации

Порядок кольцевой факторизации	Количество арифметических прогрессий	Доля оставшихся чисел
0	1	100%
1	1	50%
2	2	33.33%
3	8	26.67%
4	48	22.86%
5	480	20.78%
6	5 760	19.18%
7	92 160	18.05%
8	1 658 880	17.1%
9	36 495 360	16.36%
10	1 021 870 080	15.79%

Из таблицы видно, что с ростом порядка кольцевой факторизации доля оставшихся чисел сокращается все меньше. При этом количество арифметических прогрессий сильно возрастает. А значит, увеличиваются накладные расходы для составления и хранения в памяти всех арифметических прогрессий. По этой причине начиная с некоторого момента увеличение порядка кольцевой факторизации, будет лишь замедлять работу алгоритма. По этой причине на практике обычно используется кольцевая факторизация 2-5 порядка.

1.5 Способы модификации алгоритмов просеивания простых чисел

Один из самых популярных способов улучшения алгоритма решета заключается в разделении просеиваемого интервала на сегменты. Одно большое решето заменяется на последовательность маленьких и каждое из них просеивается отдельно. Для этого требуется предварительно подготовить список простых чисел до $\sqrt{n}$, где n – это максимальное число в заданном интервале. Список простых чисел до корня можно составить обычным решетом. Это потребует $O(\sqrt{n})$ памяти. А простые найденные в процессе высевание сегментов можно сразу отдавать в выходной поток.

По сегментное просеивание позволяет не только асимптотически уменьшить объем потребляемой памяти, но и существенно увеличить скорость работы решета. Ускорение по времени достигается за счет того, что при уменьшении объемов потребляемой памяти эффективнее используется кэш процессора и реже происходит обращение к оперативной памяти.

Также важно не делать сегменты меньше $O(\sqrt{n})$ элементов. Это не дает никакого выигрыша в асимптотике потребления памяти, но из-за накладных расходов все сильнее теряется производительность.

Сегментированное решето в отличие от обычного очень хорошо распараллеливается по итерациям цикла. Порядок выполнения итераций просеивания сегмента никак не влияет на итоговый результат.

Так как все простые числа необходимые для просеивания сегментов уже хранятся в отдельном массиве. Распараллеливание сегментированного решета позволяет добиться дополнительного ускорения, пропорционального количеству используемых процессоров.

Также обычно решето реализуется с использованием массива булевых элементов. Этот тип данных может принимать только значения true или false. Казалось бы, для хранения одного элемента массива достаточно 1 бита памяти, что было бы просто замечательно. Но в действительности под него выделяется 8 бит, так как память адресуется только целыми байтами.

Это приводит к необходимости сделать еще одну оптимизацию по объему потребляемой памяти, называемую битовым сжатием. Принцип очень простой. Вместо массива булевых элементов используется любой целочисленный массив. А обработка массива происходит путем обращения напрямую к отдельным битам элементов массива. В этом случае чтение и запись необходимого бита представляет собой ряд битовых операций.

Такой подход позволяет уменьшить объем потребляемой памяти еще в 8 раз. А благодаря кэш-фактору уменьшения объема занимаемой памяти скорость работы программы возрастает примерно вдвое. Хотя на первый взгляд, кажется, что стоит ожидать замедления работы алгоритма из-за необходимости выполнять дополнительные операции при обращении к битам.

2 Программная реализация алгоритмов поиска последовательных простых чисел

2.1 Выбор программных средств

В качестве среды для разработки программы выбрана Visual Studio 2019. Данная среда программирования позволяет разрабатывать консольные приложения и приложения с графическим интерфейсом. В том числе поддерживается технология Windows Forms.

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, дизайнер классов и дизайнер схемы базы данных.

Visual Studio позволяет создавать и подключать сторонние плагины (дополнения) для расширения функциональности практически на каждом уровне. В том числе добавление поддержки систем контроля версий исходного кода, таких как: Subversion, SourceSafe.

Так же добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

Visual Studio включает следующие компоненты:

1. Visual Basic .NET, а до его появления – Visual Basic;
2. Visual C++;
3. Visual C# (включен начиная с Visual Studio .NET);
4. Visual F# (включен начиная с Visual Studio 2010).

Данная среда достаточно проста и удобна для понимания.

Язык программирования выберем C#. Разрабатываемая программа поиска последовательных простых чисел будет написана в Windows Forms.

В программе будут использоваться такие средства языка C#, как:

1. Конструктор форм;
2. Классы, структуры;
3. Типы данных byte, int, uint, long, ulong, double, string, DateTime, bool;
4. Операторы for, if-else, switch-case;
5. Функции (методы);
6. События элементов формы;
7. Средства библиотеки TPL для параллельных вычислений;
8. Математические операции класса Math.

2.2 Интерфейс приложения

Разработку программы следует начать с создания удобного интерфейса. С помощью конструктора форм была спроектирована форма, представленная на рисунке 1.

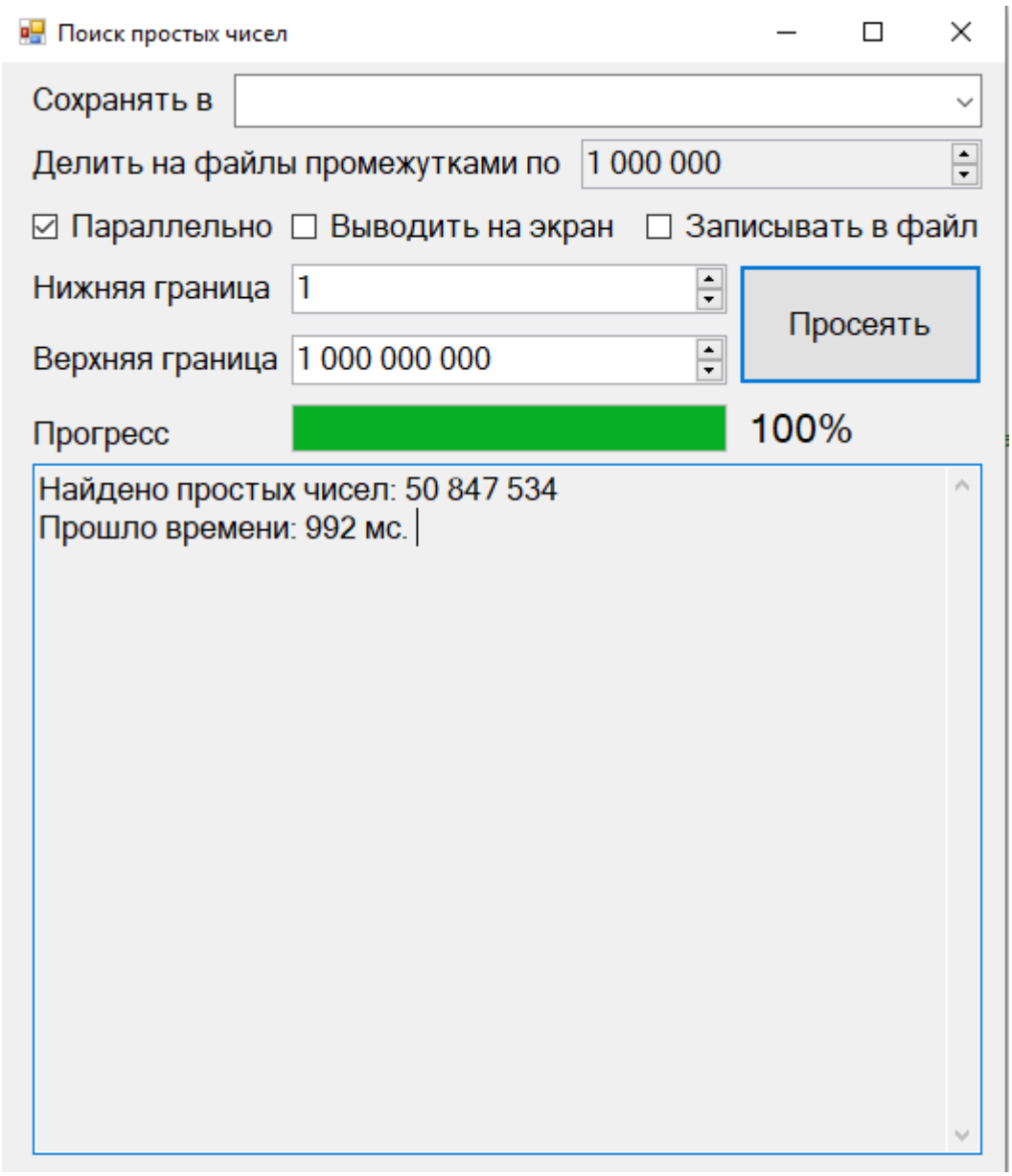

Рисунок 1 – Интерфейс приложения

В форму были добавлены такие элементы, как:

1. NumericUpDown – для ввода верхней и нижней границы, а также выбора размера файлов, при записи простых чисел;
2. ComboBox – отображает и позволяет выбирать путь к папке, в которой буду сохраняться текстовые файлы с простыми числами;

3. CheckBox – для выбора режима работы программы;
4. Label – для информирования пользователя о назначении элементов;
5. Button – для запуска и остановки вычислений;
6. ProgressBar – отображает ход вычислений;
7. TextBox – для вывода результатов просеивания и вывода простых чисел, если установлена соответствующая галочка.

Для более удобного восприятия все числа выводятся с разделителями между тысячами в виде пробела. Также ограничена возможность установки галочки лишь в одном CheckBox, так как одновременное использование нескольких режимов не предполагается. В NumericUpDown для выбора размера файлов убрана возможность ввода числа самостоятельно. Вместо этого пользователь может выбирать размер файл с помощью стрелочек в этом элементе. Размер файла изменяется всегда в 10 раз в большую или меньшую сторону. В NumericUpDown для ввода верхней и нижней границы установлены ограничения на минимальное значение, равное 1, а также максимальное значение, равное $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$. Ограничения на максимальное значение связаны с использованием стандартного 64 битного беззнакового типа данных ulong, так как использование длинной арифметики не планируется. В ComboBox убрана возможность ввода текста и выбора из списка. Вместо этого вызывается диалоговое окно для выбора папки, в которой будут сохраняться файлы. Также предусмотрена возможность отмены просеивания. После запуска вычислений текст на элементе Button меняется на «Отмена» и повторное нажатие этой кнопки вызывает отмену текущих вычислений. С целью оптимизации проверка на то, была ли запрошена отмена вычислений производится не слишком часто. Поэтому при значениях верхней границы близких к максимальным отмена может быть произведена не сразу, а спустя некоторое время. На этот случай текст на элементе button меняется на «Подождите» и кнопка становится не доступной, чтобы избежать дальнейшего ее нажатия. Также при запуске просеивания все элементы формы, кроме кнопки отмены становятся недоступны, чтобы избежать бессмысленных случайных нажатий. После завершения просеивания или в случае отмены все элементы формы вновь становятся доступны, а в текстовом поле отображается результат. Также предусмотрено изменение размеров и местоположения элементов формы при изменении размеров самой формы.

2.3 Реализации решета Эратосфена и его модификаций

Следующим шагом в разработке программы будет создание функций, реализующих алгоритм решета Эратосфена с модификациями, описанными в подглавах 1.4 и 1.5.

Для начала рассмотрим самую базовую реализацию алгоритма решета Эратосфена, описанного в подглаве 1.1:

```
public static bool[] BaseReshetoEratosfena(uint N){
            if (N < 2) return new bool[0];
            bool[] P = new bool[N + 1];
            for (ulong i = 2, max = (ulong)Math.Sqrt(N); i <= max;i++)
                if (!P[i])
                    for (ulong j = i * i; j <= N; j += i)
                        P[j] = true;
            return P;}
```

Такой маленький фрагмент кода уже позволяет достаточно быстро просеивать простые числа на интервале от 1 до $2^{31} - 2 = 2\,147\,483\,646$. К сожалению, большее количество элементов в массиве содержаться не может. К тому же даже если мы исхитримся обойти это ограничение, используя массив массивов, необходимое количество оперативной памяти будет расти непомерно быстро. Так уже при $N = 2\,147\,483\,646$ требуется 2 гигабайта оперативной памяти. Поэтому для поиска больших простых чисел такая реализация не подходит.

Теперь рассмотрим тот же алгоритм, но использующий битовое сжатие для уменьшения расхода памяти в 8 раз:

```
public static ulong[] ReshetoEratosfena(ulong N) {
    if (N < 2) return new ulong[0];
    ulong Length = N - 2, p = 0, c = 1, j = 0;
    ulong[] P = new ulong[Length / 64 + 1];
    Length = (ulong)P.Length * 64;
    uint maxi = (uint)Math.Min((Math.Sqrt(N) + 1) / 64 + 2, P.Length);
            for (uint i = 0 ; i < maxi; i++, c += 64)
                for (int k = 0; k < 64; )
                    if ((P[i] & 1UL << k++) == 0)
                        for (p = c + (uint)k, j = p * p - 2;
                            j < Length; j += p)
                            P[j / 64] |= 1UL << (int)(j % 64);
            return P;}
```

В этой реализации вместо булевых элементов мы используем биты 64 битного числа. Чтобы отмечать число, как составное, используются битовые операции для установки соответствующего бита в значение 1. На первый взгляд

кажется, что такая реализация позволит уменьшить потребление оперативной памяти ровно в 8 раз, но при этом должно увеличиться время работы алгоритма, так как приходится дополнительно использовать битовые операции для манипуляции над битами. Однако тесты показали, что на практике битовое сжатие также приводит и к ускорению работы алгоритма примерно в 2 раза. Объяснить это можно тем, что процессор гораздо быстрее обращается к данным, расположенным в оперативной памяти, за счет уменьшения объема памяти, занимаемой этими данными. Но такая модификация лишь немного улучшает ситуацию с использованием оперативной памяти, позволяя просеивать простые числа на интервале от 1 до $2^{37} - 63 = 137\,438\,953\,409$. И то работать с таким большим диапазоном будет возможно только, если у вас имеется 16 гигабайт оперативной памяти. Чтобы действительно существенно уменьшить расход оперативной памяти, необходимо улучшить алгоритм так, чтобы изменилась асимптотика использования памяти. Этого можно достичь, используя сегментированное решето. Рассмотрим базовую реализацию такого решета:

```
public static ulong BaseSegmentReshetoEratosfena
(ulong Nmin, ulong Nmax) {
        if (Nmax < 2 || Nmax < Nmin) return 0;
        if (Nmax == 2 || (Nmax == 3 && Nmin == 3)) return 1;
        if (Nmax == 3) return 2;
        uint SqrtN = (uint)Math.Sqrt(Nmax),
        Cache = Math.Max(70000, SqrtN), i;
        uint[] Prime = BaseReshetoEratosfenaPrime(SqrtN);
        bool[] Segment;
        ulong start = Math.Max(Nmin, SqrtN + 1),
        last = start + Cache - 1, j, max, R, n = 0;
        for (; last <= Nmax; start += Cache, last += Cache)
        {
            Segment = new bool[Cache];
            for (i = 0, max = (ulong)Math.Sqrt(last);
                 i < Prime.Length && Prime[i] <= max; i++)
                for (R = start % Prime[i], j = R == 0 ?
                0 : Prime[i] - R; j < Cache; j += Prime[i])
                    Segment[j] = true;
            for (i = 0; i < Cache; i++)
                if (!Segment[i])
                    n++;
        }
        Segment = new bool[Nmax - start + 1];
        {//Выполнить итерацию цикла выше для последнего сегмента}
        return n;}
```

Используя сегментированное решето, мы уже можем задавать не только верхнюю границу, но и нижнюю. Что очень важно, ведь далеко не всегда нужно

искать все простые числа, начиная с единицы. Суть сегментированного решета заключается в том, чтобы сначала найти все простые числа, на которые теоретически могут раскладываться составные числа в заданном диапазоне. Все такие простые числа не могут превосходить $\sqrt{\text{Nmax}}$, поэтому первым этапом алгоритма будет поиск всех простых чисел до $\sqrt{\text{Nmax}}$ с помощью обычного, не сегментированного решета. Теперь имея необходимую базу простых чисел, мы можем начать просеивать заданный диапазон, исключая из него все числа кратные простым из массива Prime. Причем нет необходимости просеивать сразу весь диапазон, лучше всего его разделить на небольшие сегменты. В идеале сегменты должны выбираться такого размера, чтобы массивы Prime и Segment полностью помещались в кэш процессора. Тогда обращение к нужным элементам происходит на порядок быстрее и соответственно алгоритм выполняется за меньшее время. Но в тоже время не следует делать сегменты меньше, чем $\sqrt{\text{Nmax}}$, так как в этом случае для некоторых простых чисел из массива Prime в массиве Segment может не найтись ни одного кратного составного. То есть некоторые итерации цикла, пробегающего по индексу i не принесут никакой пользы, а лишь в пустую потратят процессорное время. В итоге использование сегментированного решета позволяет уменьшить расход памяти с $O(n)$ до $O(\sqrt{n})$, что позволяет просеивать простые числа на интервале от 1 до $2^{62} - 1 = 4\,611\,686\,018\,427\,387\,903$. Правда размер массива Segment нужно также ограничить числом $2^{31} - 1 = 2\,147\,483\,647$ (максимальное количество элементов в массиве). При работе с максимальной верхней границей расход памяти на массив Prime составит $4 * 105\,097\,565 = 420\,390\,260$ Б ≈ 400 МБ. $105\,097\,565$ – это количество простых чисел до $\sqrt{\text{Nmax}}$, что показано на рисунке 2. А расход памяти на массив Segment при его размерах в $2\,147\,483\,647$ элементов составит 2 гигабайта. В случае острой нехватки памяти, можно еще больше ограничить размеры массива Segment в ущерб производительности. То есть даже такая простейшая реализация сегментированного решета позволяет находить простые числа длиной до 19 знаков, используя от 1 до 2 гигабайт оперативной памяти.

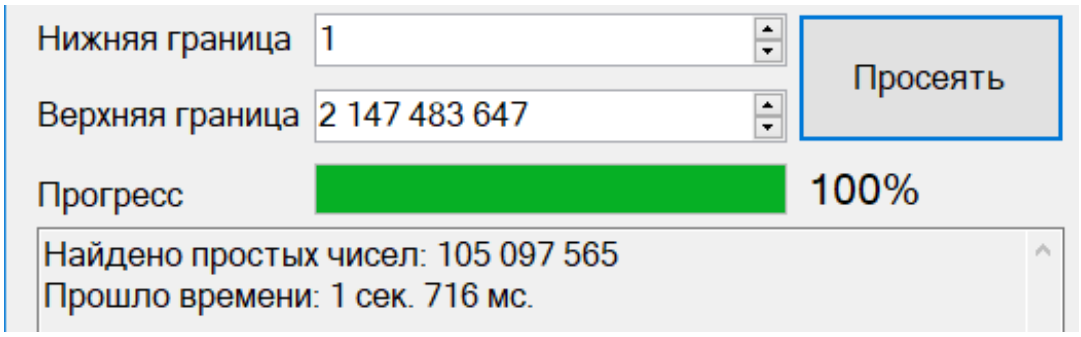

Рисунок 2 – Количество простых чисел до $2^{31} - 1$

Теперь рассмотрим реализацию решета Эратосфена с использованием кольцевой факторизации 2-го порядка (также использовано битовое сжатие):

```
public static ulong[] ReshetoEratosfenaFactor2(ulong N)
{
    if (N < 2) return new ulong[0];
    ulong Length = N < 6 ? 1 : (N / 3 - 1), p, d, c = 0, j1,j2;
    ulong[] P = new ulong[Length / 64 + 1];
    Length = 64 * (uint)P.Length;
    uint maxi = (uint)Math.Min(Math.Sqrt(N)/192 + 2, P.Length);
    for (uint i = 0; i < maxi; i++, c += 64) {
        for (int k = 0; k < 64; ) {
            if ((P[i] & 1UL << k++) == 0) {
                p = c + (uint)k;
                j1 = 3 * p * p;
                d = 6 * p;
                j2 = j1 + d;
                if ((k % 2) == 1) {
                    j2++;
                    j1 += 4 * p;
                    d += 4;}
                else {
                    j1 += 2 * p - 1;
                    d += 2;}
                for (; j2 < Length; j1 += d, j2 += d) {
                    P[j1 / 64] |= 1UL << (int)(j1 % 64);
                    P[j2 / 64] |= 1UL << (int)(j2 % 64);}
                for (; j1 < Length; j1 += d)
                    P[j1 / 64] |= 1UL << (int)(j1 % 64);}}}
    return P;
}
```

Особенность такой реализации в том, что порядковый номер бита, равный $64i + k$, обозначает не число, которое мы отмечаем, как составное или простое. Вместо этого порядковый номер бита обозначает номер члена последовательности натуральных чисел из которой были исключены $1, 2, 3$ и также все натуральные числа, кратные 2 и 3. Эта последовательность начинается так:
$5,\ 7,\ 11,\ 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49, 53, 55, 59, 61, 65, 67,$
$71, 73, 77, 79, 83, 85, 89, 91, 95, 97, 101, 103, 107, 109, 113, 115, 119 \ldots$.
Стоит отметить, что среди первых 40 членов этой последовательности всего 12 являются составными числами. Получить член этой последовательности, зная его номер, можно, используя следующую формулу:

$$a_i = \begin{cases} 3i + 1, & i - \text{четное}, \\ 3i + 2, & i - \text{нечетное}, \end{cases}$$

где нумерация i начинается с 1. При работе с такой последовательностью вместо последовательности всех натуральных чисел есть некоторые особенности. Например, при вычеркивании из этой последовательности всех чисел, кратных некоторому простому числу, необходимо пробегать по членам двух последовательностей j1 и j2, первые члены которых находятся по специальным формулам. При этом шаг в этих последовательностях равен $d = 2a_i$, то есть удвоенному простому числу, кратные которому вычеркиваются. Это неочевидно, но такой метод вычеркивания чисел, кратных заданному простому аналогичен тому, как это делается в обычной реализации решета Эратосфена без кольцевой факторизации.

Во всех предыдущих реализациях вычеркивались все числа кратные простому числу p, начиная с p^2 и с шагом p. Это все числа вида: $p^2,\ p(p+1),\ p(p+2), \dots,\ p(p+k), \dots$. То есть, чтобы вычеркнуть из последовательности натуральных чисел все числа кратные заданному простому числу p необходимо вычеркивать числа, получаемые в результате умножения числа p на само себя и на все последующие после него члены последовательности натуральных чисел. Применив это правило к последовательности a_i, получим, что для нечетного i необходимо вычеркивать числа вида:

$$(3i+2)^2, \quad (3i+2)(3i+4), \quad (3i+2)(3i+8),$$
$$(3i+2)(3i+10), \quad (3i+2)(3i+14), \quad (3i+2)(3i+16), \dots.$$

Эту последовательность чисел удобно разделить на 2 подпоследовательности:

$$\mathrm{j1} = (3i+2)^2, \quad (3i+2)(3i+2+6*1),$$
$$(3i+2)(3i+2+6*2), \quad (3i+2)(3i+2+6k), \dots,$$

$$\mathrm{j2} = (3i+2)(3i+4), \quad (3i+2)(3i+4+6*1),$$
$$(3i+2)(3i+4+6*2), \quad (3i+2)(3i+4+6k), \dots.$$

После раскрытия скобок и некоторых преобразований получим:

$$\mathrm{j1} = 3(3i^2+4i+1)+1, \quad 3(3i^2+4i+1+(6i+4)*1)+1,$$
$$3(3i^2+4i+1+(6i+4)*2)+1, \quad 3(3i^2+4i+1+(6i+4)*k)+1 \dots,$$

$$\mathrm{j2} = 3(3i^2+6i+2)+2, \quad 3(3i^2+6i+2+(6i+4)*1)+2,$$
$$3(3i^2+6i+2+(6i+4)*2)+2, \quad 3(3i^2+6i+2+(6i+4)*k)+2 \dots.$$

Теперь видно, что члены этих последовательностей также являются членами последовательности a_i, причем в скобках указан их порядковый номер в последовательности a_i. В связи с этим удобно будет переписать

последовательности j1 и j2 так, чтобы их члены обозначали не составное число, которое вычеркивается, а порядковый номер этого числа в последовательности a_i. После этих преобразований получим:

$$\mathrm{j1} = 3i^2 + 4i + 1, \quad 3i^2 + 4i + 1 + (6i + 4) * 1,$$
$$3i^2 + 4i + 1 + (6i + 4) * 2, \quad 3i^2 + 4i + 1 + (6i + 4) * k \ldots,$$

$$\mathrm{j2} = 3i^2 + 6i + 2, \quad 3i^2 + 6i + 2 + (6i + 4) * 1,$$
$$3i^2 + 6i + 2 + (6i + 4) * 2, \quad 3i^2 + 6i + 2 + (6i + 4) * k \ldots .$$

Можно заметить, что первые члены этих последовательностей получается по формулам:

$$\mathrm{j1}_0 = 3i^2 + 4i + 1,$$

$$\mathrm{j2}_0 = 3i^2 + 6i + 2.$$

Все последующие члены этих последовательностей получаются увеличением предыдущего члена на величину $(6i + 4) = 2(3i + 2) = 2a_i = d$.

Аналогичным образом получаются формулы для четного i:

$$\mathrm{j1}_0 = 3i^2 + 2i,$$

$$\mathrm{j2}_0 = 3i^2 + 6i + 1.$$

А шаг для четного i равен $2a_i = 2(3i + 1) = (6i + 2) = d$.

Именно эти формулы и используются в коде, приведенном выше. Но записаны они в неявном виде для минимизации количества операций необходимых для нахождения первых членов последовательностей j1 и j2.

Существенный прирост в скорости работы алгоритма и уменьшение расхода памяти в 3 раза в этой реализации достигается за счет того, что $2/3$ заведомо составных чисел полностью исключается из рассмотрения, а просеивается лишь оставшаяся $1/3$ от всех чисел. Также преимущество этой реализации в том, что при вычеркивании всех кратных некоторому простому числу алгоритм в цикле пробегает сразу по двум индексам j1 и j2, вычеркивая за одну итерацию цикла сразу 2 составных числа. Это позволяет добиться параллелизма на уровне команд процессора. Ведь последовательности команд:

```
P[j1 / 64] |= 1UL << (int)(j1 % 64);
P[j2 / 64] |= 1UL << (int)(j2 % 64);
```

никак не связаны между собой и могут выполняться процессором одновременно, что дополнительно повышает скорость выполнения алгоритма.

Такая реализация решета Эратосфена позволяет просеивать простые числа до $3 * 2^{37} - 190 =$
$= 412\ 316\ 860\ 226$. При этом потребуется 16 гигабайт оперативной памяти.

Все вышеизложенные варианты реализации решета Эратосфена с модификациями позволяют существенно снизить расход оперативной памяти и сократить время просеивания. Но для достижения максимальной производительности необходим алгоритм, который объединяет все эти идеи вместе. То есть нужен алгоритм сегментированного решета, который также будет использовать битовое сжатие и кольцевую факторизацию. Именно такой алгоритм с некоторыми дополнительными улучшениями и должен использоваться в конечном варианте, разрабатываемого приложения.

2.4 Реализация и улучшение алгоритма сегментированного решета Эратосфена с битовым сжатием, кольцевой факторизацией и параллельным просеиванием

Следующий этап в разработке программного средства заключается в том, чтобы объединить все реализованные модификации решета Эратосфена в один наиболее производительный алгоритм. А затем произвести дополнительные улучшения этого алгоритма и его оптимизацию на уровне кода. Следующий фрагмент кода представляет реализацию такого алгоритма в виде функции:

```
long SF2RE(ulong Nmin, ulong Nmax)
{
    if (Nmax < 2 || Nmax < Nmin) return 0;
    if (Nmax == 2 || (Nmax == 3 && Nmin == 3)) return 1;
    if (Nmax == 3) return 2;
    ulong CountNum; long n;
    byte[] Prime =
    CreateBasePrime(ref Nmin, ref Nmax, out CountNum, out n);
    ulong CountOfSegment = (Nmax-Nmin)/CountNum, iteration = 0;
    if ((Nmax - Nmin) % CountNum != 0) CountOfSegment++;
    Timer Time = Timer.Now;
    for (; Nmin < Nmax; Nmin += CountNum) {
        ulong[] Segment =
        SievSegment(Prime, Nmin, Nmax, ref CountNum);
        for (int i = 0, max = Segment.Length - 1; i < max;)
            n += CountOfZeroBits(Segment[i++]);
        for (int k = 0, max = Segment.Length - 1,
        maxk = (int)((CountNum - 1) % 64); k <= maxk;)
            if ((Segment[max] & 1UL << k++) == 0)
                n++;
        iteration++;
        if (Time.LeftMilliseconds > 33)
        { Time = Timer.Now;
          backgroundWorker1.ReportProgress((int)
          (iteration /(double)CountOfSegment * 1000000000)); }}
    return n;
}
```

В этой реализации необходимо отдельно обрабатывать частные случаи, когда Nmax или Nmin не превосходят 3. Это связано с тем, что при использовании кольцевой факторизации 2-го порядка упускаются простые числа 2 и 3. На первом этапе этой реализации используется отдельная функция CreateBasePrime, которая генерирует массив Prime, содержащий информацию обо всех простых числах, не превосходящих $\sqrt{\text{Nmax}}$.

Далее определяется количество сегментов, которое необходимо будет просеивать. Размер сегмента устанавливается в функции CreateBasePrime.

Перед началом основного цикла, пробегающего по сегментам, создается переменная класса Timer. Класс таймер содержит набор свойств и методов, позволяющих отслеживать и отображать информацию о том, сколько времени прошло с момента создания экземпляра этого класса. Структура класса Timer будет рассмотрена позже. В цикле переменная класса Timer необходима для отслеживания количества времени, прошедшего с момента последнего вывода информации о прогрессе просеивания на форму. Это позволяет избежать потери производительности, связанной с чрезмерно частым обновлением элементов формы. Частота обновления элементов формы не превосходит 30 раз в секунду, что обеспечивает плавность обновления и практически никак не сказывается на производительности.

В теле цикла используется функция SievSegment, которая заполняет массив Segment информацией о том является ли число простым или составным для каждого числа из диапазона от Nmin до Nmax. 1 в массиве Segment указывает на то, что член последовательности a_i, рассмотренной в предыдущей подглаве, с порядковым номером $Nmin + 64i + k$ является составным числом. Поэтому для подсчета количества простых чисел, найденных в текущем сегменте, необходимо подсчитать количество нулевых бит в массиве Segment. Для достижения этой цели можно просто просматривать каждый бит, каждого элемента массива Segment и увеличивать переменную n каждый раз, когда встречается нулевой бит. Но в этой реализации для подсчета количества нулевых бит использована специальная функция CountOfZeroBits:

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
uint CountOfZeroBits(ulong n)
{
    n -= (n >> 1) & 0x5555555555555555UL;
    n = ((n >> 2) & 0x3333333333333333UL)
      + (n & 0x3333333333333333UL);
    return 64 - (uint)(((((n >> 4) + n)
      & 0x0F0F0F0F0F0F0F0FUL) * 0x0101010101010101) >> 56);
}
```

Первая строка перед заголовком функции необходима для того, чтобы заставить компилятор подставлять эту функцию в место ее вызова, а не вызывать функцию во время выполнения программы. Такой способ подсчета бит взят из [12]. Суть его заключается в том, чтобы делить последовательность из 64 бит на группы по $2, 4, 8$ и суммировать биты внутри групп. В результате сумма всех 64 битов будет находиться в старших 8 битах числа. Остается только сдвинуть число на 56 позиций вправо и вычесть сумму единичных бит из 64.

Преимущество этого метода в том, что, затратив всего 15 арифметических (битовых) операций и операций присваивания, мы получаем сумму всех бит в 64 битном числе. В то время, как наивный подход,

требующий просматривать каждый бит в отдельности затратил бы 6 операций для каждого бита. То есть $64 * 6 = 384$ операции для каждого 64 битного числа!

Рассмотрим теперь код функции SievSegment:

```
ulong[] SievSegment
(byte[] Prime, ulong Nmin, ulong Nmax, ref ulong CountNum)
{
    ulong Cache = CountNum >> 6, p = Prime[0],
          distance = Nmax - Nmin;
    if (distance < CountNum) {
        CountNum = distance;
        Cache = (uint)(((CountNum - 1) >> 6) + 1); }
    ulong[] Segment = new ulong[Cache];
    for (uint i = 0, max = (uint)Math.Sqrt((Nmin+CountNum)/3);
    p <= max; p += Prime[++i]) {
        ulong d = (p & 1) == 0 ? 6 * p + 2 : 6 * p + 4,
              R = Nmin % d, dR = d - R, j1, j2;
        if (R <= p) { j1 = p - R; j2 = dR - p - 1; }
        else {if (p >= dR) { j2 = dR + d - p - 1; j1 = dR + p;}
        else { j2 = dR + p; j1 = dR - p - 1; } }
        for (; j2 < CountNum; j1 += d, j2 += d) {
            Segment[j1 >> 6] |= 1UL << (int)(j1 & 63);
            Segment[j2 >> 6] |= 1UL << (int)(j2 & 63); }
        for (; j1 < CountNum; j1 += d)
            Segment[j1 >> 6] |= 1UL << (int)(j1 & 63); }
    return Segment;
}
```

Эта реализация использует элементы массива Prime для получения простых чисел, с помощью которых просеивается сегмент, начинающийся с Nmin и имеющий размер не более CountNum бит. Перед началом основного цикла просеивания проверяется не является ли текущий сегмент последним. Если сегмент последний, то его размер должен быть урезан до необходимого. В теле цикла в переменной p хранится порядковый номер члена последовательности a_i, кратные которому должны быть помечены, как составные в текущем сегменте. Первые члены последовательностей j1 и j2 получены с использованием формул, изложенных в предыдущей подглаве и адаптированных для сегментированного решета. Далее члены последовательностей j1 и j2 помечаются в текущем сегменте, как составные, аналогично тому, как это сделано в не сегментированном алгоритме кольцевой факторизации 2-го порядка. Операция j1 >> 6 равноценна использованию операции j1 / 64, а операция j1 & 63 равноценна j1 % 64. Такая подмена операций была выполнена с целью оптимизации. Так как операции «сдвиг» и побитовое «И» выполняются процессором значительно быстрее чем целочисленное деление и взятие остатка.

Рассмотрим функцию CreateBasePrime:

```
 byte[] CreateBasePrime
(ref ulong Nmin, ref ulong Nmax, out ulong CountNum, out long n)
 {
     uint v = Nmin < 3 ? 2U : Nmin == 3 ? 1U : 0U;
     uint SqrtN =
     (uint)Math.Min((ulong)Math.Sqrt(Nmax), 4294967291);
     ulong[] P = F2RE(SqrtN);
     n = 0; long V = 0;
     CountNum = (uint)Math.Max(8192UL * 32UL,
     (ulong)P.Length / 2UL * 64UL);//(8192 = 1KB)
     uint Length = (uint)P.Length-1, t = 64*Length+1, l = 0, s;
     for (uint i = 0; i < Length;)
         n += CountOfZeroBits(P[i++]);
     Nmin = Nmin % 6 == 2 ? Nmin / 3 + 1 : Nmin / 3;
     Nmax = Nmax % 6 == 4 ? (Nmax - 1) / 3 : (Nmax - 1) / 3 + 1;
     SqrtN = SqrtN % 6 == 4 ? (SqrtN - 1)/3 : (SqrtN - 1)/3 + 1;
     for (int k = 0; t < SqrtN; t++)
         if ((P[Length] & 1UL << k++) == 0) n++;
     byte[] Prime = new byte[n + 1];
     Prime[n] = byte.MaxValue; n = 0; t = 0;
     for (int i = 0; i < Length; i++, t += 64)
         for (int k = 0; k < 64;)
             if ((P[i] & 1UL << k++) == 0) {
                 s = t + (uint)k;
                 Prime[n++] = (byte)(s - l); l = s;
                 if (l < Nmin) V = n; }
     t = 64 * Length + 1;
     for (int k = 0; t < SqrtN; t++)
         if ((P[Length] & 1UL << k++) == 0) {
             Prime[n++] = (byte)(t - l); l = t;
             if (l < Nmin) V = n; }
     if (Nmin <= SqrtN) {
         n -= V - v;
         Nmin = SqrtN; }
     else n = 0;
     return Prime;
 }
```

Эта функция необходима для генерации всех простых чисел, на которые могут раскладываться составные числа, не превосходящие Nmax. Все такие простые числа не превосходят $\sqrt{\text{Nmax}}$. Для просеивания используется функция F2RE, которая является полным аналогом функции ReshetoEratosfenaFactor2, рассмотренной в предыдущей подглаве. Переменная CountNum устанавливает размер сегментов в битах. Ее значение используется в функции SievSegment при выделении памяти под сегмент. Далее, имея информацию обо всех простых

числах до $\sqrt{\text{Nmax}}$ в массиве P, необходимо подсчитать количество нулевых бит в этом массиве, чтобы точно определить размер массива Prime в который будет в явном виде записана информация обо всех простых числах до $\sqrt{\text{Nmax}}$. Для удобства дальнейшей работы значения Nmin, Nmax, SqrtN заменяются на соответствующий для них порядковый номер в последовательности a_i. В самом конце этой реализации проверяется не превосходит ли $\sqrt{\text{Nmax}}$ нижнюю границу Nmin. Если это так, то значение нижней границы повышается до $\sqrt{\text{Nmax}}$, а количество всех простых чисел от исходной нижней границы до $\sqrt{\text{Nmax}}$ сохраняется в переменной n.

Важной особенностью этой реализации, отличающей ее от того, как обычно реализуется сегментированное решето, является структура массива Prime. Обычно в массиве Prime хранятся все простые числа, необходимые для дальнейшего просеивания сегментов. При этом массив Prime должен целиком находиться в оперативной памяти все время работы алгоритма. Стандартный подход к хранению простых чисел предполагает хранение их в явном виде. Это означает что при наибольшем значении $\text{Nmax} = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ необходимо будет составить массив простых чисел до $\sqrt{\text{Nmax}} = 2^{32} - 1 = 4\,294\,967\,295$. На рисунке 3 показано, что таких простых чисел всего $203\,280\,221$. Так как все эти числа не превосходят $uint.MaxValue = 2^{32} - 1$, то для хранения каждого из них будет достаточно выделить 4 байта оперативной памяти. Итого при стандартном подходе потребуется до $4 * 203\,280\,221 = 813\,120\,884$ Б ≈ 775 МБ оперативной памяти только для хранения массива Prime.

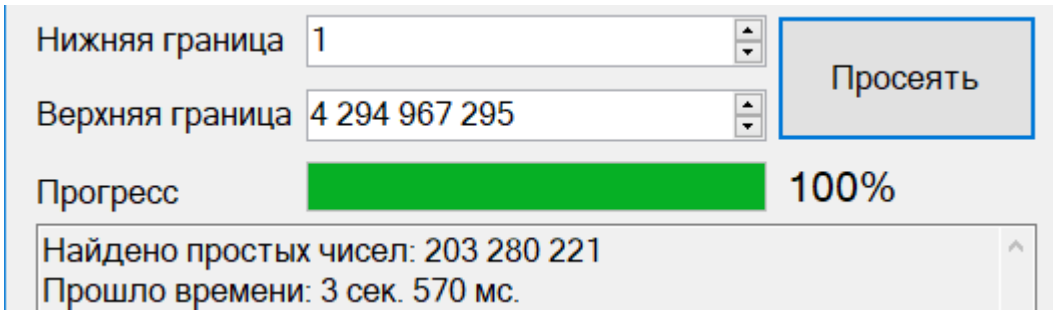

Рисунок 3 – Количество простых чисел до $2^{32} - 1$

В этой же реализации в массиве Prime простые числа не хранятся в явном виде. В первую очередь нужно отметить, что в массиве Prime хранится информация не о самих простых числах, а об их порядковых номерах в последовательности a_i. При этом алгоритм на всех этапах также работает с порядковыми номерами последовательности a_i. Для минимизации расхода оперативной памяти на массив Prime в нем хранятся разности между порядковыми номерами i соседних простых членов последовательности a_i. А при работе с массивом Prime порядковые номера членов последовательности a_i

получаются последовательным сложением элементов массива. Эта идея была предложена в [13]. Для простых чисел не превосходящих $2^{32} - 1 = 4\,294\,967\,295$ разность между соседними простыми не превосходит 360. А порядковый номер i простого числа a_i всегда примерно втрое меньше, чем само a_i. Это значит, что и разность между порядковыми номерами i простых чисел из a_i тоже примерно втрое меньше, чем разность между простыми членами a_i и не превосходит 120. Такой способ хранения информации о простых числах в массиве Prime уменьшает максимально возможное значение его элементов с $2^{32} - 1 = 4\,294\,967\,295$ до 120, что позволяет использовать всего 1 байт памяти для хранения каждого элемента массива Prime. Значения, которые можно закодировать 8 битами памяти варьируются от 0 до $2^8 - 1 = 255$, а значит разность всегда будет помещаться в 8 бит памяти. Такое улучшение позволяет уменьшить расход оперативной памяти на массив Prime ровно в 4 раза. Максимальный расход памяти на массив Prime при таком подходе составит $1 * 203\,280\,221 = 203\,280\,221$ Б ≈ 194 МБ.

В современных системах обычно имеется достаточное количество оперативной памяти, поэтому не имеет большого значения будет ли затрачено 775 МБ или 194 МБ. Но уменьшение объема данных в памяти важно не только для экономии оперативной памяти компьютера, а также для ускорения работы алгоритма. Наибольшая эффективность при работе алгоритма будет достигаться, если все данные помещаются в кэш процессора. В этом случае процессор показывает максимально возможную производительность. Поэтому уменьшение объема памяти, занимаемой массивом Prime в 4 раза увеличит частоту попадания в кэш процессора также примерно в 4 раза и обеспечит максимально эффективное использование процессора. Также согласно этому принципу выбирается размер сегментов в битах равный CountNum.

На самом деле возможно уменьшить расход памяти еще на 1/8, используя для каждого элемента массива Prime всего 7 бит памяти. Ведь 7 битами можно закодировать значения от 0 до $2^7 - 1 = 127 > 120$, а значит 7 бит также достаточно для хранения всех возможных разностей между порядковыми номерами соседних простых чисел. Но среди стандартных типов данных в языке программирования C# не существует 7 битного типа данных, а значит потребуется создание и использование собственного типа данных. Именно поэтому в данной реализации используется стандартный 8 битный тип данных byte.

Рассмотрим теперь реализацию параллельной версии улучшенного сегментированного решета Эратосфена с кольцевой факторизацией:

```
long SF2REParallel(ulong Nmin, ulong Nmax) {
    if (Nmax < 2 || Nmax < Nmin) return 0;
    if (Nmax == 2 || (Nmax == 3 && Nmin == 3)) return 1;
    if (Nmax == 3) return 2;
    ulong CountNum; long n;
```

```
byte[] Prime =
CreateBasePrime(ref Nmin, ref Nmax, out CountNum, out n);
ulong CountOfSegment = (Nmax-Nmin)/CountNum, iteration = 0;
if ((Nmax - Nmin) % CountNum != 0) CountOfSegment++;
Timer Time = Timer.Now;
ParallelOptions thread = new ParallelOptions
{ MaxDegreeOfParallelism = Environment.ProcessorCount };
Parallel.For<long>(0, (int)CountOfSegment, thread, () => 0,
(ix, loop, Thisn) => {
    ulong Count = CountNum;
    ulong[] Segment = SievSegment
    (Prime, Nmin + (ulong)ix * Count, Nmax, ref Count);
    for (int i = 0, max = Segment.Length - 1; i < max;)
        Thisn += CountOfZeroBits(Segment[i++]);
    for (int k = 0, max = Segment.Length - 1,
    maxk = (int)((Count - 1) % 64); k <= maxk;)
        if ((Segment[max] & 1UL << k++) == 0) Thisn++;
    iteration++;
    if (Time.LeftMilliseconds > 33)
    { Time = Timer.Now;
      backgroundWorker1.ReportProgress((int)
      (iteration / (double)CountOfSegment * 1000000000)); }
    return Thisn; },
 (x) => Interlocked.Add(ref n, x));
return n; }
```

Эта реализация полностью аналогична предыдущей, рассмотренной ранее, за тем лишь исключением, что в качестве основного цикла, пробегающего по сегментам, используется не обычный for, а его параллельная версия Parallel.For. Параллельная версия цикла автоматически распределяет итерации цикла на все доступные процессоры. Каждый поток, созданный в этом цикле, выделяет в памяти место под свой сегмент и полностью просеивает его, поэтому при чрезмерно большом количестве потоков может просто не хватить оперативной памяти. Чтобы этого избежать максимально возможное количество потоков, работающих одновременно, ограничено количеством логических ядер процессора доступных в системе.

Параллельная версия алгоритма позволяет уменьшить время его работы пропорционально количеству ядер процессоров в системе, где он запускается. Но при этом расход памяти в параллельной версии также увеличивается пропорционально количеству ядер процессоров в системе. Размер же самих сегментов выбирается пропорционально корню квадратному из верхней границы. При просеивании максимально больших чисел, доступных в программе, с верхней границей $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ размер одного сегмента составит:

$$\frac{\sqrt{2^{64}-1}}{6} = 715\,827\,883 \text{ бит } \approx 85 \text{ МБ}.$$

Исходя из этого и зная размер базы простых чисел до 2^{32} можно сказать, что общий расход памяти в параллельной версии программы не превосходит $200 + 85 * CPU$ мегабайт, где CPU – это количество логических ядер процессоров в системе. В версии же без параллельных вычислений расход памяти не превосходит 370 МБ, так как размер сегмента установлен вдвое больше, чем для параллельной версии.

Эффективность реализованного алгоритма решета Эратосфена со всеми доступными улучшениями по сравнению с более простыми и менее оптимизированными его реализациями будет рассмотрена в главе 3.

2.5 Класс измерения времени

Помимо алгоритмов просеивания простых чисел в программе также используется множество дополнительных функций. Но особое внимание хотелось бы уделить классу `Timer`. Рассмотрим его реализацию:

```
class Timer {
     private readonly DateTime T;
     public Timer(DateTime t)
     { T = t; }
     public Timer(Timer Time)
     {    T = Time.T; }
     public static Timer Now
     { get { return new Timer(DateTime.Now); } }
     public int LeftMilliseconds
     { get { return (int)((DateTime.Now.Ticks - T.Ticks)/10000); } }
     public override string ToString(){
         TimeSpan Time = DateTime.Now - T;
         string S = "";
         if (Time.Days > 0)
             S += Time.Days + " дней ";
         if (Time.Hours > 0)
             S += Time.Hours + " часов ";
         if (Time.Minutes > 0)
             S += Time.Minutes + " минут ";
         if (Time.Seconds > 0)
             S += Time.Seconds + " сек. ";
         S += Time.Milliseconds + " мс. ";
         return S;} }
```

Этот класс был создан для удобства измерения времени выполнения алгоритмов и вывода этой информации в наглядном виде. Также он используется для контроля частоты обновления информации о прогрессе вычислений на форме. Класс содержит единственное поле, имеющее тип данных DateTime. Предусмотрено 2 конструктора с передачей в качестве параметра объекта DateTime или Timer. Для удобства также было добавлена статическое свойство класса Timer.Now, оно позволяет удобно создавать объект класса, который запоминает в качестве точки отсчета текущее время. Свойство LeftMilliseconds возвращает количество времени в миллисекундах прошедшее с момента создания объекта (если он создавался свойством Timer.Now). Для наглядного вывода информации о прошедшем времени была переопределена функция ToString(), которая выводит время, выделяя количество прошедших дней, часов, минут, секунд и миллисекунд.

2.6 Реализация решета Сундарама и его оптимизация

С целью сравнения эффективности алгоритмов просеивания последовательных простых чисел необходимо также реализовать решето Сундарама. Рассмотрим его простейшую реализацию:

```
public long Base_Siev_Sundaram(ulong m, ulong N)
{
    if (N < 2) return 0;
    int n = (int)((N - 1) / 2), Count = 1;
    bool[] P = new bool[n + 1];
    for (int i = 1; 2 * i * (i + 1) <= n; i++)
        for (int j = i; j <= (n - i) / (2 * i + 1); j++)
            P[2 * i * j + i + j] = true;
    for (uint i = 1; i <= n; i++)
        if (!P[i])
            Count++;
    return Count;
}
```

В соответствии с теорией в этой реализации все числа вида $2ij + i + j$ исключаются в двойном цикле. А размер решета берется вдвое меньше верхней границы, что уменьшает расход оперативной памяти в 2 раза в сравнении с базовой реализацией решета Эратосфена. Так как достаточно только подсчитать количество простых чисел, а не выводить их, то не требуется умножать все оставшиеся числа на 2 и прибавлять единицу.

Теперь стоит применить к этой реализации общие для алгоритмов просеивания способы оптимизации. А именно битовое сжатие, быстрый подсчет нулевых битов и оптимизация самого кода. Улучшенная реализация выглядит так:

```
public long Siev_Sundaram_Optimise(ulong m, ulong N)
{
    if (N < 2) return 0;
    uint Length = (uint)((N - 1) / 2), n = 0;
    ulong[] P = new ulong[Length / 64 + 1];
    for (uint i = 1, step = 3, maxi =
    (uint)((Math.Sqrt(N) - 1) / 2); i <= maxi; i++, step += 2)
        for (uint j = i * (step + 1); j <= Length; j += step)
            P[j >> 6] |= 1UL << (int)(j & 63);
    Length = (uint)P.Length - 1; int k = 0;
    for (int i = 0; i < Length; i++)
        n += CountOfZeroBits(P[i]);
    for (uint p = 128U * Length + 1; p <= N; p += 2)
        if ((P[Length] & 1UL << k++) == 0) n++;
    return n;}
```

Циклы были оптимизированы таким образом, чтобы не выполнять большое количество арифметических операций в каждой итерации цикла. Формула $2ij + i + j$ теперь используется неявно для вычеркивания составных чисел. Также при проверке условий окончания циклов была устранена избыточность операций. В этой реализации значение сравнивается с заранее просчитанной верхней границей. Все это в совокупности с использованием битов целочисленного массива вместо булевых элементов и использованием алгоритма быстрого подсчета нулевых бит существенно ускоряет алгоритм и уменьшает его потребление оперативной памяти в 8 раз. Более сложные способы оптимизации подобных алгоритмов не использовались при реализации решета Сундарама, так как использование этого алгоритма планируется в конечном варианте реализуемого приложения. Решето Сундарама может быть еще существенно улучшено с использованием кольцевой факторизации, сегментации просеиваемого интервала и распараллеливанием вычислений по итерациям цикла.

2.7 Реализация решета Аткина и его оптимизация

Помимо решета Эратосфена и Сундарама для сравнения алгоритмов просеивания также немаловажно реализовать решето Аткина, которое имеет преимущество в асимптотике времени работы. Простейшая его реализация выглядит примерно так:

```
public long Base_Siev_Atkin(ulong m, ulong Nm)
{
    long N = (long)Nm;
    if (N < 2) return 0;
    uint SqrtN = (uint)Math.Sqrt(N);
    bool[] Prime = new bool[N + 1];
    Prime[2] = true; Prime[3] = true;
    for (long x = 1, n; x <= SqrtN; x++)
        for (long y = 1; y <= SqrtN; y++){
            n = 4 * x * x + y * y;
            if ((n <= N) && (n % 12 == 1 || n % 12 == 5))
                Prime[n] = !Prime[n];
            n = 3 * x * x + y * y;
            if ((n <= N) && (n % 12 == 7))
                Prime[n] = !Prime[n];
            n = 3 * x * x - y * y;
            if ((x > y) && (n <= N) && (n % 12 == 11))
                Prime[n] = !Prime[n]; }
    for (uint i = 5; i <= SqrtN; i++)
        if (Prime[i])
            for (uint j = i * i; j <= N; j += i * i)
                Prime[j] = false;
    int Count = 0;
    for (uint i = 0; i <= N; i++)
        if (Prime[i])
            Count++;
    return Count;
}
```

Основное время работы алгоритма уходит на выполнения двойного цикла по x и y. Из заголовка этого цикла очевидно, что он выполняется $\sqrt{N} * \sqrt{N} = N$ итераций. Что неудивительно, ведь алгоритм имеет линейную асимптотическую сложность по времени. Этот алгоритм также возможно улучшить с использованием стандартных способов оптимизации алгоритмов просеивания последовательных простых чисел. Улучшенная реализация выглядит следующим образом:

```
public long Siev_Atkin_Optimise(ulong m, ulong Nm)
{
    long N = (long)Nm, halfN = N / 2;
    if (N < 2) return 0;
    uint SqrtN = (uint)Math.Sqrt(N);
    ulong[] Prime = new ulong[N / 128 + 1];
    for (long y = 1,x1,x2, n1, n2, n3, n4; y <= SqrtN; y += 2){
        for (x1 = 2, x2 = 6 * y, n1 = ((y * y) >> 1) + 2,
        n2 = ((y * (y + 8)) >> 1) + 10, n3 = y * (y + 3) + 1,
        y += 4, n4 = y * (y + 3) + 1; n4 <= halfN;
        n1 += x1 += 4, n2 += x1, n3 += x2 += 12, n4 +=x2 + 24){
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63);
            Prime[n2 >> 6] ^= 1UL << (int)(n2 & 63);
            Prime[n3 >> 6] ^= 1UL << (int)(n3 & 63);
            Prime[n4 >> 6] ^= 1UL << (int)(n4 & 63); }
        for (; n3 <= halfN; n3 += x2 += 12)
            Prime[n3 >> 6] ^= 1UL << (int)(n3 & 63);
        for (; n2 <= halfN; n1 += x1 += 4, n2 += x1) {
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63);
            Prime[n2 >> 6] ^= 1UL << (int)(n2 & 63); }
        for (; n1 <= halfN; n1 += x1 += 4)
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63); }
    for (long x = 1,y,n1,n2,maxx = SqrtN >> 1;x <= maxx; x+=2){
        for (y=0, n1=((x*x) << 1) + 4, n2 = ((++x*x) << 1) + 4;
        n2 <= halfN; n1 += y += 36, n2 += y) {
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63);
            Prime[n2 >> 6] ^= 1UL << (int)(n2 & 63); }
        for (; n1 <= halfN; n1 += y += 36)
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63); }
    for (long y = 2, x1, x2, n1, n2, n3, n4; y <= SqrtN; y+=4){
        for (x1 = 0, x2 = 6 * y, n1 = ((y * y) >> 1) + 1,
        n2 = ((y * (y + 4)) >> 1) + 3, n3 = y * (y + 3) + 1,
        y += 2,n4 = y*(y + 3) + 1; n4 <= halfN; n1 += x1 += 12,
        n2 += x1, n3 += x2 += 12, n4 += x2 + 12) {
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63);
            Prime[n2 >> 6] ^= 1UL << (int)(n2 & 63);
            Prime[n3 >> 6] ^= 1UL << (int)(n3 & 63);
            Prime[n4 >> 6] ^= 1UL << (int)(n4 & 63); }
        for (; n3 <= halfN; n3 += x2 += 12)
            Prime[n3 >> 6] ^= 1UL << (int)(n3 & 63);
        for (; n2 <= halfN; n1 += x1 += 12, n2 += x1) {
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63);
            Prime[n2 >> 6] ^= 1UL << (int)(n2 & 63); }
        for (; n1 <= halfN; n1 += x1 += 12)
            Prime[n1 >> 6] ^= 1UL << (int)(n1 & 63); }
    for (int i = 2; i <= SqrtN; i++)
        if ((Prime[i >> 6] & 1UL << (i & 63)) != 0)
            for (long step = ((i * (i + 1L)) << 2) + 1,
            j = step >> 1, maxj = N >> 1; j <= maxj; j += step)
```

```
                    Prime[j >> 6] &= ~(1UL << (int)(j & 63));
        uint Count = 2; int k = 0, Length = Prime.Length - 1;
        for (int i = 0; i < Length; i++)
            Count += CountOfOneBits(Prime[i]);
        for (long p = 128U * Length + 1; p <= N; p += 2)
            if ((Prime[Length] & 1UL << k++) != 0)
                Count++;
        return Count;
    }
```

Вместо массива булевых элементов теперь используются биты целочисленного массива, причем хранятся в массиве только нечетные числа, то есть использована простейшая кольцевая факторизация 1 порядка. Также добавление массива бит позволило использовать быстрый алгоритм подсчета нулевых битов. Помимо этого, была проведена глобальная оптимизация всего алгоритма и кода. Сильной перестройке подвергся основной этап алгоритма, использующий двойной цикл для подсчета количества решений уравнений. Основные его проблемы в базовой реализации были:

1. Большое количество операций умножения;
2. Избыточное количество сравнений для проверки выхода за границы;
3. Многократное вычисление остатков от деления.

Для решения первой проблемы достаточно было немного оптимизировать код, сохраняя результаты промежуточных вычисления и повторно используя их. Со второй проблемой оказалось чуть больше сложностей. Для ее решения потребовалось разбить общий двойной цикл по 3-ем уравнениям на 3 раздельных двойных цикла по каждому уравнению в отдельности. После чего стало возможным подобрать необходимые формулы, чтобы обозначить четкие границы для x и y в каждом из уравнений. После такой оптимизации больше не требовалось проверять не превысило ли n верхнюю границу, а также уменьшилось общее количество бесполезных итераций цикла.

Если первая и вторая проблемы решались относительно легко и не требовали углубления в теорию, то для решения третьей (наиболее значимой!) проблемы потребовалось изучить некоторые свойства 3-ех уравнений, решения которых подсчитываются для просеивания. Было обнаружено, что для первого уравнения $4x^2 + y^2 = n$ остаток от деления на 12 равен 1 или 5 только в двух случаях:

1. $y = 6k \pm 1, \forall\, x \ \ni N,$
2. $\begin{cases} y = 6k + 3 \\ x \neq 3k \end{cases}.$

Для второго уравнения $3x^2 + y^2 = n$ остаток от деления на 12 равен 7 только тогда, когда:

$$\begin{cases} y = 6k \pm 2 \\ x = 2k + 1 \end{cases}.$$

А для третьего уравнения $3x^2 - y^2 = n$ при $x > y$ остаток от деления на 12 равен 11 в двух случаях:

1. $\begin{cases} y = 6k \pm 1, \\ x = 2k. \end{cases}$

2. $\begin{cases} y = 6k \pm 2, \\ x = 2k + 1. \end{cases}$

Из этого следует, что если организовать циклы особым образом так, чтобы они пробегали только по тем значениям x и y для которых остаток заведомо принимает необходимое значение, тогда не будет смысла проверять равен ли остаток этому значению. То есть в этих циклах можно будет в каждой итерации сразу менять значение бита с номером n на противоположное, не выполняя при этом никаких дополнительных проверок и избыточных операций. Именно таким образом и была полностью решена 3-я проблема базовой реализации этого алгоритма. Стоит отметить, что именно ее разрешение дало колоссальный прирост в скорости работы алгоритма. Что и будет показано в следующей главе при сравнительном анализе алгоритмов просеивания простых чисел.

Стоит отметить, что решето Аткина еще может быть существенно улучшено с использованием сегментации просеиваемого интервала, кольцевой факторизации более высокого порядка и распараллеливанием вычислений. Но в рамках этой работы для сравнения эффективности алгоритмов ограничимся уже выполненной оптимизацией.

3 Сравнительный анализ эффективности реализованных алгоритмов просеивания

Для проведения сравнительного анализа были выбраны все реализованные в процессе разработки реализации решета Эратосфена, а также базовые и оптимизированные реализации решета Аткина и Сундарама. Список всех функций, представляющих реализации этих алгоритмов представлен на рисунке 4.

```
//Тестовые функции

//Базовое решето Сундарама
public long Base_Siev_Sundaram(ulong m, ulong N)...
//Оптимизированное решето Сундарама
public long Siev_Sundaram_Optimise(ulong m, ulong N)...
//Базовое решето Аткина
public long Base_Siev_Atkin(ulong m, ulong Nm)...
//Оптимизированное решето Аткина
public long Siev_Atkin_Optimise(ulong m, ulong Nm)...
//Базовое решето Эратосфена
public long Base_Siev_Eratosfen(ulong m, ulong N)...
//Решето с битовым сжатием
public long Siev_Eratosfen(ulong m, ulong N)...
//Оптимизированное решето с битовым сжатием
public long Siev_Eratosfen_Optimise(ulong m, ulong N)...
//Оптимизированное решето с битовым сжатием и быстрым подсчетом простых чисел
public long Siev_Eratosfen_Optimise_Fast_Enumerate(ulong m, ulong N)...
//Решето по нечетным числам (1 порядок кольцевой факторизации)
public long Siev_Eratosfen_Factor1(ulong m, ulong N)...
//Решето со 2 порядком кольцевой факторизации
public long Siev_Eratosfen_Factor2(ulong m, ulong N)...
//Решето с 3 порядком кольцевой факторизации
public long Siev_Eratosfen_Factor3(ulong m, ulong N)...
//Базовое сегментированное решето
public long Segment_Base_Siev_Eratosfen(ulong Nmin, ulong Nmax)...
//Сегментированное решето с битовым сжатием
public long Segment_Siev_Eratosfen(ulong Nmin, ulong Nmax)...
//Сегментированное решето по нечетным числам (1 порядок кольцевой факторизации)
public long Segment_Siev_Eratosfen_Factor1(ulong Nmin, ulong Nmax)...

//Сегментированные решета со 2-ым порядком кольцевой факторизации
//составляет базу простых чисел до корня
public long Segment_Siev_Eratosfen_Factor2_Use_Base_Prime(ulong Nmin, ulong Nmax)...
//не переписывает числа в базу простых чисел, а использует их прямо из решета до корня
public long Segment_Siev_Eratosfen_Factor2_Use_Siev_Prime(ulong Nmin, ulong Nmax)...
//вместо базы простых чисел использует интервалы между ними.
public long Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime(ulong Nmin, ulong Nmax)...
//Работает параллельно на всех доступных логических процессорах и использует интервалы
public long Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime_Parallel(ulong Nmin, ulong Nmax)
```

Рисунок 4 – Список алгоритмов, участвовавших в тестировании

Стоит пояснить названия этих функций. Слово «Siev» в названии всех функций переводится, как «Решето», чем и являются все эти алгоритмы. Приставка «Base» в названии означает, что использовалась самая простейшая реализация алгоритма. В таком виде алгоритмы обычно объясняются в

учебниках и реализуется для того, чтобы показать идею. Именно поэтому они максимально просты и прозрачны, но в тоже время наименее эффективны на практике. Слова «Atkin», «Sundaram» и «Eratosfen», очевидно, обозначают автора алгоритма, лежащего в основании. Суффикс «FactorN» говорит о том, что в данной реализации использовалась кольцевая факторизация порядка N. А приставка «Segment» сообщает, что в данной реализации используется посегментное просеивание заданного интервала чисел, а значит такой алгоритм может быть использован для просеивания больших простых чисел длиной до 20 знаков.

Отдельно нужно сказать про функцию Siev_Eratosfen_Optimise, в ней слово «Optimise» обозначает то, что алгоритм отличается от базовой версии лишь добавлением битового сжатия (использован массив битов целых чисел вместо булевых элементов) и оптимизацией на уровне кода (минимизация количества операций). При этом в этой реализации все-таки использован обычный способ подсчета количества нулевых бит. А в функцию Siev_Eratosfen_Optimise_Fast_Enumerate как раз помимо прочих оптимизаций был добавлен также алгоритм быстрого подсчета нулевых бит. Во всех же остальных реализациях, за исключением базовых, по умолчанию обязательно используется битовое сжатие и быстрый подсчет нулевых бит, хотя отдельно это и не указывается. Для решета Сундарама и Аткина слово «Optimise» обозначает комплексное улучшение алгоритма. Подробнее об этом говорится в предыдущей главе.

Сегментированные решета Эратосфена со 2-ым порядком кольцевой факторизации имеют свои особенные окончания. «Use_Base_Prime» значит, что после просеивания всех простых чисел до корня из верхней границы они будут переписаны в отдельный uint[] массив, чтобы в дальнейшем использовать их при просеивании сегментов. «Use_Siev_Prime» говорит о том, что после просеивания всех простых чисел до корня из верхней границы они не будут никуда переписываться, а сразу будут использованы для просеивания сегментов прямо из решета. Плюсы такого способы в экономии оперативной памяти на хранение базы простых чисел и экономии времени для составления этой базы и решета. А существенный минус заключается в том, что не известно, какие именно числа являются простыми в решете до корня из верхней границы. А значит, для поиска следующего простого, которое должно быть вычеркнуто из сегмента, необходимо последовательно просматривать каждый бит решета пока не будет найден следующий нулевой бит. Окончание «Use_Base_Interval_Prime» информирует, что в данной реализации составляется база простых чисел до корня из верхней границы, но хранятся эти числа не в их явном виде, а в виде интервалов между соседними номерами простых чисел в кольцевой факторизации 2-го порядка. Использование такого подхода позволяет минимизировать расход оперативной памяти для хранения базы простых чисел и при этом не снижает скорость доступа к простым числам из базы, так как при просеивании они просматриваются последовательно.

Суффикс «Parallel» используется лишь в одной функции и говорит о том, что данная реализация алгоритма распределяет вычисления на все доступные ядра процессора. Для удобства сравнения скорости выполнения алгоритмов и использования ими оперативной памяти были написаны специальные тестирующие функции, представленные на рисунке 5.

```
//Тестирующие функции
public void TestSievs(ulong Nmin, ulong Nmax)
{
    TestSiev(Base_Siev_Eratosfen, Nmin, Nmax);
    TestSiev(Siev_Eratosfen, Nmin, Nmax);
    TestSiev(Siev_Eratosfen_Optimise, Nmin, Nmax);
    TestSiev(Siev_Eratosfen_Optimise_Fast_Enumerate, Nmin, Nmax);
    TestSiev(Base_Siev_Sundaram, Nmin, Nmax);
    TestSiev(Siev_Sundaram_Optimise, Nmin, Nmax);
    TestSiev(Base_Siev_Atkin, Nmin, Nmax);
    TestSiev(Siev_Atkin_Optimise, Nmin, Nmax);
    TestSiev(Siev_Eratosfen_Factor1, Nmin, Nmax);
    TestSiev(Siev_Eratosfen_Factor2, Nmin, Nmax);
    TestSiev(Siev_Eratosfen_Factor3, Nmin, Nmax);
    TestSiev(Segment_Base_Siev_Eratosfen, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen_Factor1, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen_Factor2_Use_Base_Prime, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen_Factor2_Use_Siev_Prime, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime, Nmin, Nmax);
    TestSiev(Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime_Parallel, Nmin, Nmax);
}
public void TestSiev(Func<ulong, ulong, long> Siev, ulong Nmin, ulong Nmax)
{
    if ((Nmin < 3 && Nmax < int.MaxValue) || Siev.Method.Name.Substring(0, 7) == "Segment")
    {
        textBox1.AppendText(Siev.Method.Name + ":");
        GC.Collect(GC.MaxGeneration, GCCollectionMode.Forced, true);
        Timer T = Timer.Now;
        long CountPrime = Siev.Invoke(Nmin, Nmax);
        textBox1.AppendText("\r\n          Прошло времени: " + T +
            "\r\n          Найдено простых чисел: " + CountPrime.ToString("#,#\r\n\r\n"));
    }
}
```

Рисунок 5 – Функции, необходимые для проведения тестов

Первая из них просто вызывает вторую для всех тестируемых алгоритмов. А вторая же представляет из себя функцию, принимающую в качестве параметра другую функцию и два ее аргумента для которых она должна быть вызвана. Перед вызовом функции, которую необходимо протестировать выводится ее имя и запрашивается принудительная сборка всего мусора в оперативной памяти, чтобы сделать тестирование максимально объективным. После завершения сборки мусора запускается таймер и тут же вызывается функция. Для измерения расхода оперативной памяти в каждую реализацию была добавлена строка, представленная на рисунке 6.

```
ulong[] P = REF1(N);
textBox1.AppendText("\r\n          Использовано памяти: "
    + Math.Round(8L * P.Length / Math.Pow(2, 20), 2) + " MB");
```

Рисунок 6 – Строка для вывода информации об объеме оперативной памяти, который использует алгоритм во время работы

Объем оперативной памяти измеряется умножением количества элементов массива или массивов, которые использует алгоритм на вес одного элемента в байтах. Такой способ дает очень точную оценку реального объема задействованной оперативной памяти. После завершения выполнения функции выводится информация о количестве найденных простых чисел и прошедшем времени с момента запуска функции. Количество найденных простых чисел позволяет проверить правильность работы реализованного алгоритма.

Результаты проведения тестов оформлены в виде таблице. Для экономии места в первом столбце таблицы, вызываемая функция обозначается ее порядковым номером в соответствии со следующим списком:

1. Base Siev Eratosfen – простейшая реализация решета Эратосфена;
2. Siev Eratosfen – решето с битовым сжатием, без оптимизации кода;
3. Siev Eratosfen Optimise – решето Эратосфена с битовым сжатием и максимальной оптимизацией арифметических операций;
4. Siev Eratosfen Optimise Fast Enumerate – предыдущий алгоритм, улучшенный добавлением быстрого подсчета количества простых чисел;
5. Base Siev Sundaram – простейшая реализация решета Сундарама;
6. Siev Sundaram Optimise – оптимизированное решето Сундарама;
7. Base Siev Atkin – простейшая реализация решета Аткина;
8. Siev Atkin Optimise – оптимизированное решето Аткина;
9. Siev Eratosfen Factor1 – решето Эратосфена с 1 порядком кольцевой факторизации, битовым сжатием и быстрым подсчетом простых чисел;
10. Siev Eratosfen Factor2 – решето Эратосфена со 2 порядком кольцевой факторизации, битовым сжатием и быстрым подсчетом простых чисел;
11. Siev Eratosfen Factor3 – решето Эратосфена с 3 порядком кольцевой факторизации, битовым сжатием и быстрым подсчетом простых чисел;
12. Segment Base Siev Eratosfen – простейшая реализация решета Эратосфена с посегментным просеиванием;
13. Segment Siev Eratosfen – сегментированное решето Эратосфена с битовым сжатием;
14. Segment Siev Eratosfen Factor1 – сегментированное решето Эратосфена с битовым сжатием и 1 порядком кольцевой факторизации;
15. Segment Siev Eratosfen Factor2 Use Base Prime – сегментированное решето с битовым сжатием и 2 порядком кольцевой факторизации. Составляет базу простых чисел до $\sqrt{Nmax}$;

16. Segment Siev Eratosfen Factor2 Use Siev Prime – в отличии от предыдущего алгоритма использует простые числа из решета до $\sqrt{Nmax}$;

17. Segment Siev Eratosfen Factor2 Use Base Interval Prime – вместо базы простых чисел составляет базу интервалов между соседними из них;

18. Segment Siev Eratosfen Factor2 Use Base Interval Prime Parallel – улучшение предыдущего алгоритма, распределением вычислений на все доступные ядра процессора.

Все тесты проводились на ноутбуке с двухъядерным процессором Intel Core i3-3227U 1.90 GHz и оперативной памятью 8GB DDR3 1.60 GHz. Из-за большого количества скриншотов с результатами проведения всех тестов они представлены в приложении А. В качестве первого теста найдем количество простых чисел до $2\,000\,000\,000 = 2 * 10^9 = 2$ млрд. Тестирование будем проводить сначала без использования автоматической оптимизации компилятора, а затем с автоматической оптимизацией. Это позволит отследить где именно использование автоматической оптимизации дает наибольший прирост производительности и оценить насколько хорошо оптимизирован код вручную. Ниже в таблице 2 представлены результаты проведения теста для обоих режимов компиляции.

Таблица 2 – Время работы и расход оперативной памяти алгоритмов при нахождении всех простых чисел до $2 * 10^9$ в двух режимах компиляции

	Время работы (секунд)		Использование памяти (мегабайт)
№	Оптимизация выключена	Оптимизация включена	
1	75	39	1907
2	208	28.7	238
3	59	28.4	238
4	45	22.6	238
5	86	49	954
6	51	31	119
7	152	47	1907
8	13.5	9.1	119
9	18.7	9.9	119
10	9.5	6.1	79
11	6.6	4.5	64
12	50	18.3	0.06
13	35	13.9	0.04
14	14.7	6.7	0.03
15	6.6	3.2	0.05
16	6.9	3.1	0.03
17	6.4	3.1	0.04
18	3.1	1.6	0.13

Первое, что бросается в глаза, глядя на эти результаты – это время работы 2-го алгоритма без оптимизации. Почему же он настолько плох? Его время работы почти втрое больше даже простейшего решета Эратосфена! Все дело в том, что в этой реализации при использовании битового сжатия намеренно не проводилось никакой оптимизации кода. А именно такое огромное время работы связано со строкой:

```
P[j / 64] |= 1UL << (int)(j % 64);
```

Эта строка в алгоритмы используется в самом внутреннем цикле, чтобы помечать числа, как составные. Соответственно она и выполняется наибольшее количество раз. А проблем этой строки заключается в том, что операции деления и нахождения остатка от деления чрезвычайно долго выполняются процессором. Очень легко оптимизировать алгоритм просто заменив эту строчку на:

```
P[j >> 6] |= 1UL << (int)(j & 63);
```

Обе эти строки выполняют одни и те же действия. Только в первой строке использованы сложные операции деления и остатка от деления, а во второй строке они заменены на очень быстрые операции сдвига и логического «И». Разницу между такими минимальными исправлениями можно видеть, сравнивая 2 и 3 алгоритмы – 208 и 59 секунд соответственно! А если обратить внимание на режим автоматической оптимизации при компиляции, то можно заметить, что 2-ой алгоритм уже выполняется всего за 28.7 секунд и ничуть не уступает 3-ему с его 28.4 секундами. Это вполне ожидаемо, ведь алгоритмы идентичны по своей структуре, а оптимизацию арифметических действий компиляторы давно умеют выполнять самостоятельно и обычно намного лучше, чем это удается сделать вручную.

Также хочется обратить внимание на различия между временем выполнения 3 и 4 алгоритмов. По структуре эти алгоритмы отличаются только способом подсчета количества простых чисел после того, как решето полностью завершило просеивание. Наглядно можно увидеть, что этот способ подсчета дает очень существенный прирост производительности. Если в 3 алгоритме подсчет количества простых чисел занимает львиную долю времени работы, то в 4 алгоритме время на подсчет количества простых чисел практически не ощутимо на фоне общего времени работы. 2, 3 и 4 алгоритмы специально были реализованы для наглядности преимуществ использования битового сжатия и быстрого подсчета количества простых чисел, а также важности того, какие арифметические действия используются. Во всех остальных алгоритмах, кроме базовых, по умолчанию использовано и битовое сжатие, и быстрый подсчет. Для базовых реализаций алгоритмов быстрый

подсчет количества простых чисел невозможен из-за использования массива булевых элементов.

При реализации решета Сундарама, к сожалению, не удалось добиться прорывного прироста производительности между базовой и оптимизированной реализацией. Время работы сократилось примерно на 40% для обоих режимов компиляции, а расход памяти в 8 раз за счет битового сжатия. Стоит отметить, что решето Сундарама на фоне остальных выделяет использование вдвое меньшего объема памяти в базовой реализации. Из-за особенностей структуры этого алгоритма даже простейшая его реализация сходна с применением кольцевой факторизации 1-го порядка.

Приятно удивило оптимизированное решето Аткина, показавшее колоссальный прирост производительности, в особенности для режима без автоматической оптимизации кода. Время выполнения базовой и оптимизированной реализации составляет 152 и 13.5 секунд соответственно. А расход оперативной памяти 1907 и 119 мегабайт соответственно. Разница в 11 раз по времени и в 16 раз по памяти! В режиме автоматической оптимизации различие между обычной и оптимизированной реализацией чуть скромнее. А именно 5-ти кратное ускорение работы алгоритма. В то время как базовая реализация решета Аткина оказалась худшим алгоритмом среди 3-ех решет, его оптимизированная версия уступает лишь решету Эратосфена со 2 и 3 порядком кольцевой факторизации, а также сегментированным решеткам Эратосфена со 2-ым порядком кольцевой факторизации. Вполне возможно, что дальнейшее улучшение решета Аткина с использованием сегментации просеиваемого интервала, кольцевой факторизации 2 или выше порядка и распределением вычислений на ядра процессора покажет конкурентно способные результаты в сравнении с лучшими реализациями сегментированного решета Эратосфена.

Среди несегментированных алгоритмов наиболее быстрым по времени и экономичным по памяти, как и ожидалось, оказалось решето Эратосфена с 3 порядком кольцевой факторизации. Скорее всего сегментированное решето Эратосфена с 3 порядком кольцевой факторизации также показало бы наивысшую производительность, но оно не было реализовано в виду сложности внутреннего устройства алгоритма при добавлении в него посегментного просеивания. Время работы несегментированного решета Эратосфена с 3 порядком кольцевой факторизации даже сопоставимо со временем работы лучших сегментированных решет.

Среди сегментированных решет наилучшими, как и ожидалось оказались решетки со 2 порядком кольцевой факторизации. А между собой различные реализации сегментированных решет 2-го порядка практически не отличаются по времени работы в данном тесте. Сравнение сегментированных решет по памяти не имеет смысла для таких малых входных данных.

Использование реализации с параллельными вычислениями на 2-ух ядерном процессоре вполне ожидаемо показало прирост производительности примерно в 2 раза в сравнение с не параллельной версией этого же алгоритма.

В целом из таблицы также видно, что использование битового сжатия дает 8-ми кратное уменьшение расхода памяти и при этом лишь ускоряет работу алгоритма. Применение кольцевой факторизации уменьшает расход памяти в соответствии с данными из таблицы 1. При этом ускорение по времени между различными порядками кольцевой факторизации даже больше, чем это ожидалось, согласно таблице 1. Это можно объяснить более быстрым обращением к памяти при уменьшении объема данных.

Количество простых чисел, найденных среди первых 2 миллиардов чисел одинаково у всех алгоритмов одинаково и составляет $98\,222\,287$, что подтверждает корректность их работы.

В дальнейшем при проведении тестов всегда используется режим компиляции с автоматической оптимизацией кода. Также используются лишь оптимизированные реализации алгоритмов, а именно все для которых расход памяти в таблице 2 не превосходит 120 мегабайт. Для тестирования были выбраны следующие отрезки чисел:

1. От 1 до 10^{10} $(1 - 10\,000\,000\,000)$;
2. От 1 до 10^{11} $(1 - 100\,000\,000\,000)$;
3. От 10^{12} до $10^{12} + 10^{10}$ $(1\,000\,000\,000\,000 - 1\,010\,000\,000\,000)$;
4. От 10^{14} до $10^{14} + 10^{10}$ $(100\,000\,000\,000\,000 - 100\,010\,000\,000\,000)$;
5. От 10^{16} до $10^{16} + 10^{10}$ $(10\,000\,000\,000\,000\,000 - 10\,000\,010\,000\,000\,000)$;
6. От 10^{18} до $10^{18} + 10^{10}$ $(1\,000\,000\,000\,000\,000\,000 - 1\,000\,000\,010\,000\,000\,000)$.

Ниже в таблицах 3 и 4 представлены результаты проведения всех тестов. В таблицу 3 также были добавлены результаты первого теста в режиме автоматической оптимизации кода для данных алгоритмов. Время работы указывается в секундах, а расход оперативной памяти в мегабайтах. В заголовках столбцов указан отрезок для которого запускались алгоритмы. Скриншоты с результатами также находятся в приложении А.

Таблица 3 – Время работы и расход оперативной памяти оптимизированных алгоритмов при нахождении всех простых чисел на заданных отрезках

	$[1, 2 * 10^9]$		$[1, 10^{10}]$		$[1, 10^{11}]$		$[10^{12}, 10^{12} + 10^{10}]$	
№	Время	Память	Время	Память	Время	Память	Время	Память
6	31	119	193	596	3038	5960	–	–
8	9.1	119	52	596	1032	5960	–	–
9	9.9	119	57	596	763	5960	–	–
10	6.1	79	34	397	468	3974	–	–
11	4.5	64	26	318	362	3179	–	–
12	18.3	0.06	104	0.13	1916	0.41	118	1.26
13	13.9	0.04	83	0.04	1087	0.08	133	0.24
14	6.7	0.03	39	0.04	540	0.05	83	0.12
15	3.2	0.05	17.3	0.07	235	0.14	46	0.34
16	3.1	0.03	18.3	0.04	268	0.04	54	0.08
17	3.1	0.04	16.9	0.04	203	0.06	46	0.12
18	1.6	0.13	8.1	0.13	102	0.15	19	0.2

Таблица 4 – Время работы и расход оперативной памяти сегментированных алгоритмов при нахождении всех простых чисел на заданных отрезках

	$[10^{14}, 10^{14} + 10^{10}]$		$[10^{16}, 10^{16} + 10^{10}]$		$[10^{18}, 10^{18} + 10^{10}]$	
№	Время	Память	Время	Память	Время	Память
12	202	12.07	286	117.35	406	1147
13	163	2.38	265	23.84	334	238
14	96	1.19	137	11.92	167	119
15	51	2.93	69	25.95	90	234
16	56	0.79	79	7.95	100	79
17	48	1.03	70	9.47	91	88
18	26	1.43	38	13.44	48	128

Для начала стоит сказать о количестве простых чисел, найденных на каждом из отрезков:

1. $[1, 2 * 10^9] - 98\,222\,287$;
2. $[1, 10^{10}] - 455\,052\,511$;
3. $[1, 10^{11}] - 4\,118\,054\,813$;
4. $[10^{12}, 10^{12} + 10^{10}] - 361\,840\,208$;
5. $[10^{14}, 10^{14} + 10^{10}] - 310\,208\,140$;
6. $[10^{16}, 10^{16} + 10^{10}] - 271\,425\,366$;
7. $[10^{18}, 10^{18} + 10^{10}] - 241\,272\,176$.

Эти цифры одинаковы для каждого из алгоритмов на каждом из отрезков. Что опять же говорит в пользу корректности реализации всех алгоритмов. Также прослеживается явная закономерность в том, что количество простых чисел на отрезках равной длины тем меньше, чем больше числа на этом отрезке. Помимо этого количество простых чисел соответствует их примерному количеству, следующему из теоремы о распределении простых чисел [14, 15, 16]. Эта теорема говорит о том, что количество простых чисел до N зависит от N, как:

$$\frac{N}{\ln N}.$$

Исходя из всего этого можно сделать вывод, что все алгоритмы реализованы корректно и подсчитывают реальное количество простых чисел на заданном отрезке.

Из таблицы 3 прекрасно видно, что для несегментированных алгоритмов расход памяти растет линейно, относительно верхней границы. Время работы же возрастает нелинейно и очевидно зависит не только от количества просеиваемых чисел, а также от объема памяти, который занимают массивы данных. Хранение больших массивов данных в памяти сильно замедляет обращение процессора к этим данным и соответственно тормозит работу алгоритма.

Среди несегментированных алгоритмов наиболее эффективным оказалось решето Эратосфена. Даже его простейшая оптимизация, предполагающая просеивание только нечетных чисел практически не уступает оптимизированной реализации решета Аткина, а при просеивании огромных массивов чисел даже превосходит его, работая примерно на 25% быстрее. Наиболее эффективной реализацией для любых объемов данных оказалось решето Эратосфена с 3 порядком кольцевой факторизации. Оно работает вдвое быстрее, чем оптимизированное решето Аткина. А на больших объемах данных даже превосходит решето Аткина почти втрое!

Решето Сундарама, к сожалению, показало крайне плохие результаты для всех входных данных на которых проводилось тестирование. Его расход памяти соответствует расходу памяти оптимизированного решета Аткина и решета Эратосфена с 1 порядком кольцевой факторизации, а время работы при этом минимум втрое больше чем у других решет. Если сравнивать решето Сундарама с фаворитом среди несегментированных решет, то оно проигрывает по скорости работы в 7-8 раз!

На рисунке 7 представлены графики зависимостей времени работы от количества просеиваемых чисел для первых 3 отрезков из таблицы 3. По горизонтали отложено количество просеиваемых чисел в логарифмической шкале, то есть $\log_{10} N$. По вертикали отложено время работы алгоритмов в

секундах. Числом на рисунке обозначается номер алгоритма, для которого построен график.

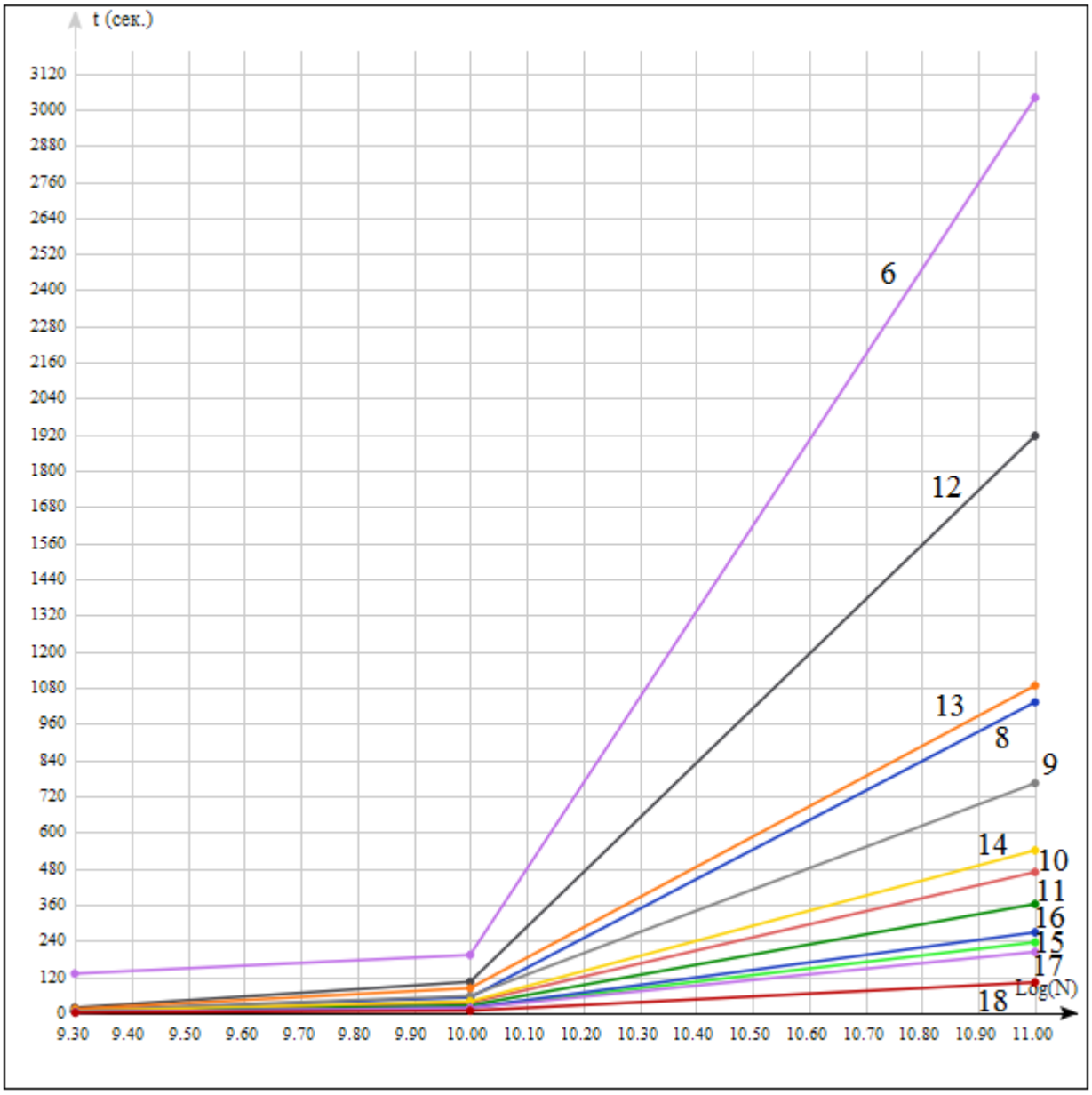

Рисунок 7 – Графики зависимостей времени работы алгоритмов от количества просеиваемых чисел

На рисунке 8 представлены графики зависимостей времени работы алгоритмов посегментного просеивания от длины просеиваемых чисел для всех отрезков длиной 10^{10} чисел из таблиц 3 и 4. Количество просеиваемых чисел на этих графиках одинаково. По горизонтали же отложено значение верхней границы просеиваемого отрезка в логарифмической шкале, то есть $\log_{10} N$. По вертикали отложено время работы алгоритмов в секундах. Числом на рисунке обозначается номер алгоритма, для которого построен график.

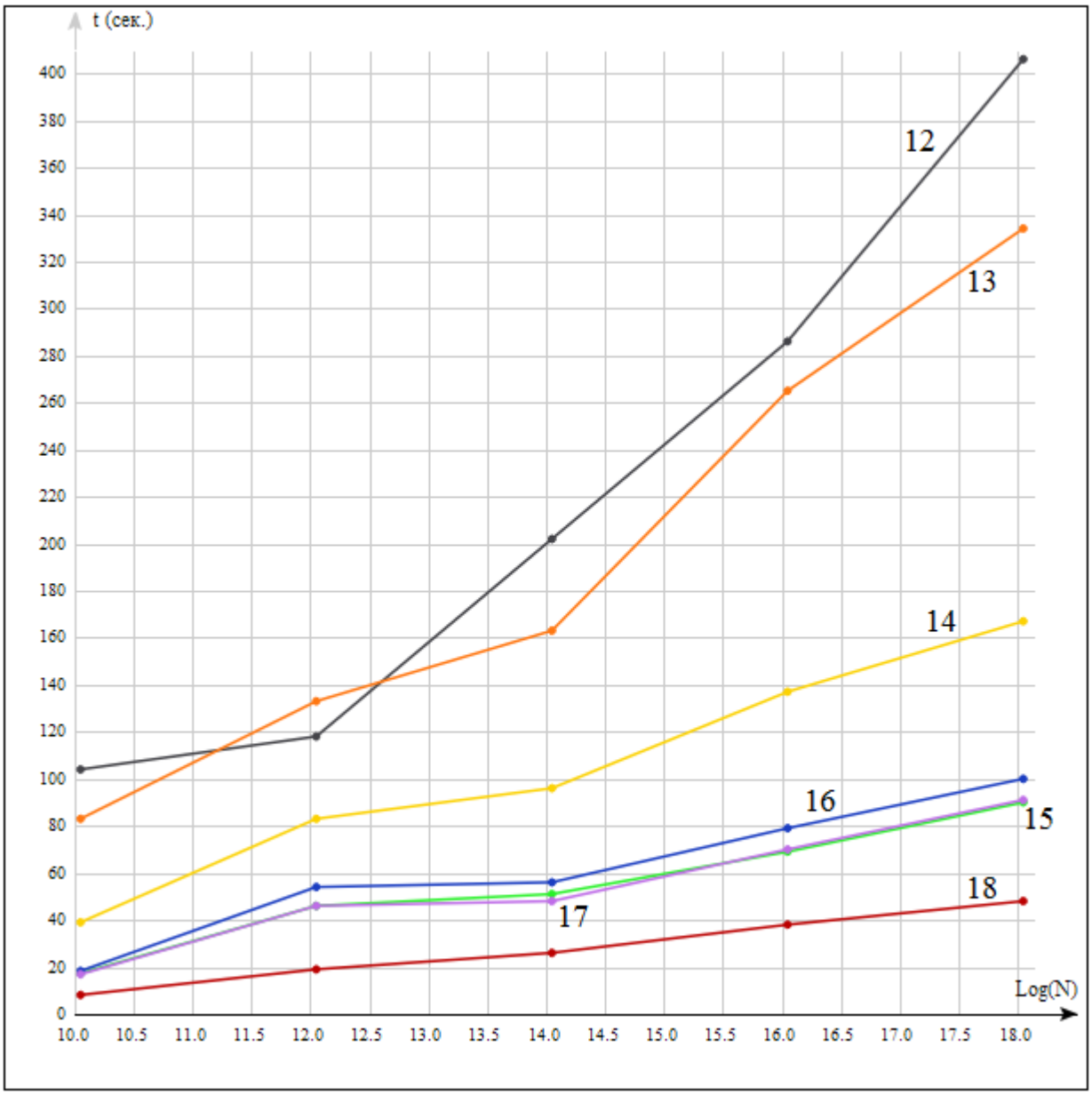

Рисунок 8 – Графики зависимостей времени работы от длины просеиваемых чисел для сегментированных алгоритмов

Говоря о сегментированных решетах нужно отметить, что в целом все они работают быстрее чем их несегментированные аналоги. При этом расход памяти для них становится существенным лишь для простых чисел длиной в 19 десятичных знаков. В то время как для несегментированных алгоритмов расход памяти существенный уже для чисел длиной 10-11 знаков. Для чисел длиной более 12 знаков расход памяти несегментированных алгоритмов возрастает критически сильно, хотя сегментированные алгоритмы для таких чисел обходятся сотней килобайт оперативной памяти. Именно благодаря асимптотическому сжатию размера данных, которыми оперируют

сегментированные алгоритмы они и работают значительно быстрее несегментированных аналогов. Для простых чисел длиной до 15 знаков все данные с которыми работает сегментированный алгоритм помещаются в кэш процессора, что и обеспечивает максимально быстрое обращение к памяти.

Использование кольцевой факторизации и битового сжатия еще более актуально для сегментированных алгоритмов, так как позволяет добиться максимального сжатия данных и соответственно улучшить попадание всех данных в кэш процессора, а значит ускорить обращение к этим данным и сократить итоговое время работы алгоритма. Это подтверждается данными таблиц 3 и 4. Из них явно видно огромное преимущество сегментированных решет Эратосфен со 2 порядком кольцевой факторизации и битовым сжатием.

Среди сегментированных решеток Эратосфена со 2 порядком кольцевой факторизации не наблюдается явного лидера по скорости выполнения. Можно только отметить, что алгоритм не использующий никакую базу простых чисел до корня из верхней границы немного уступает на больших объемах данных двум другим реализациям, где эта база используется. Это как раз связано с тем, что в нем необходимо каждый раз искать следующее простое число в решете до корня из верхней границы. И чем больше эта верхняя граница тем больше размер решета и тем реже в нем встречаются простые числа, а значит больше времени занимает их поиск.

Если говорить об использовании оперативной памяти, то фаворитом среди последних четырех алгоритмов является реализация, не использующая базу простых чисел. Так как способ хранения их в виде решета является наиболее компактным, хотя и совсем не намного превосходит способ хранения этих чисел в виде интервалов между номерами соседних простых. А вот алгоритм, использующий базу простых чисел явно расходует слишком много оперативной памяти. А именно втрое больше чем две другие реализации. При этом он не имеет никакого ощутимого преимущества по скорости выполнения над алгоритмом, хранящим базу простых чисел в виде интервалов.

Также стоит отметить, что алгоритм, использующий параллельные вычисления во всех тестах превосходит свой обычный аналог примерно в 2 раза. А в некоторых тестах даже более чем в 2 раза, что удивительно! Возможно это связано с тем, что для него размер каждого решета выбирается вдвое меньше чем для не параллельного аналога, а возможно это связано с особенностями процессора на котором проводились тесты.

Из всего вышеизложенного можно сделать вывод, что последние 2 алгоритма показывают оптимальное соотношения расхода оперативной памяти и времени выполнения. Именно поэтому они и используются в разрабатываемом приложении.

Заключение

Проведенное исследование по теме «Улучшение алгоритма кольцевой факторизации для построения простых чисел» позволяет сделать следующие выводы.

Первая задача, поставленная в данной работе, заключалась в том, чтобы проанализировать различные подходы к поиску простых чисел. В результате выполнения первой задачи установлено, что асимптотически наиболее быстрым является решето Аткина. При этом огромное значение имеет использование различных способов оптимизации алгоритмов просеивания. А лучше всего оптимизациям поддается решето Эратосфена.

Вторая задача, сформулированная в работе, предполагала выбор и реализацию наиболее эффективных алгоритмов поиска простых чисел. При выполнении этой части работы были реализованы и улучшены решета Сундарама, Аткина и Эратосфена. При их оптимизации использовались такие способы как: битовое сжатие, быстрый подсчет нулевых битов, оптимизация на уровне кода. Для решета Аткина также был разработан собственный метод оптимизации алгоритма, учитывающий его особенности. Этот метод меняет структуру основных циклов алгоритма таким образом, что удается полностью избавиться от тяжелых для процессора операций умножения и нахождения остатка от деления в основном теле цикла. Тестирование, проведенное в третьей главе работы, показало, что в режиме компиляции без автоматической оптимизации кода этот метод улучшения решета Аткина дает прирост скорости работы более чем в 10 раз!

Третья задача заключалась в реализации алгоритма кольцевой факторизации на основе решета Эратосфена. В работе был реализован алгоритм кольцевой факторизации 2-го и 3-его порядка. В сочетании с сегментированным решетом был реализован алгоритм кольцевой факторизации 2-го порядка.

Четвертой задачей в работе было улучшение алгоритма кольцевой факторизации. Для сегментированного решета Эратосфена со 2 порядком кольцевой факторизации также было использовано битовое сжатие. Помимо этого, для него был реализован способ быстрого подсчета количества простых чисел, ускоряющий эту часть алгоритма как минимум в 25 раз! Также был разработан и реализован собственный способ улучшения алгоритма, позволяющий снизить расход памяти на хранение базы простых чисел в 4 раза! Использование этого подхода уменьшает общий расход оперативной памяти алгоритмом примерно втрое и делает его практически не ощутимым при просеивании любых 64 битных чисел. Дополнительно ко всем внесенным улучшениям был реализован способ распараллеливания вычислений на все доступные ядра процессора, что позволяет ускорить алгоритм пропорционально количеству физических ядер процессоров в системе.

Провести сравнительный анализ эффективности реализованных алгоритмов по памяти и по времени стало пятой задачей работы. При выполнении этой части работы было проведено тестирование 18 различных реализаций решет. Среди несегментированных алгоритмов наиболее эффективным оказалось решето Эратосфена с 3 порядком кольцевой факторизации. Но даже решето Эратосфена с 1 порядком кольцевой факторизации превосходит по скорости работы оптимизированное решето Аткина. Решето Сундарама же многократно уступает всем реализованным алгоритмам. Результаты тестирования также показали, что для просеивания больших простых чисел обязательным является использование сегментированного решета. Среди сегментированных решет оптимальное соотношение скорости работы и объема задействованной оперативной памяти показало решето, использующее компактный способ хранения базы простых чисел. Применение кольцевой факторизации по результатам тестирования оказалось не только очень эффективным методом уменьшения объема потребляемой памяти, а также крайне эффективным способом ускорения работы алгоритма. Прирост в скорости работы оказался даже больше, чем ожидалось из теоретических оценок!

Список использованных источников

1 Петин, А. Д. Улучшение алгоритма кольцевой факторизации для построения простых чисел, Шаг в науку – № 4, 2019. – С. 1-7.

2 Коутинхо, С. Введение в теорию чисел. Алгоритм RSA, Постмаркет, 2001. – С. 105-109.

3 Sorenson, J. An Introduction to Prime Number Sieves, Department of Computer Sciences University of Wisconsin-Madison, 1990.

4 O'Neill, M. E. The Genuine Sieve of Eratosthenes, Journal of Functional Programming, Published online by Cambridge University, 2008.

5 Pritchard, P. Linear prime-number sieves: a family tree, Sci. Comput. Programming, 1987. P. 17-35.

6 Решето Сундарама [Электронный ресурс]. - 2009. - Режим доступа: https://ru.wikipedia.org/wiki/Решето_Сундарама. - 25.03.2019.

7 Atkin A. O. L., Bernstein D. J. Prime sieves using binary quadratic forms, Math. Comp. 73, 2004.

8 Минаев В. А., Васильев Н. П., Лукьянов В. В., Никонов С. А., Никеров Д. В. Индексные алгоритмы вычисления простых чисел с использованием метода кольцевой факторизации, ВЕСТНИК – № 4, 2013.

9 Минаев В. А., Сычев М. П., Никонов С. А. и Никеров Д. В., Реализация индексных алгоритмов поиска простых чисел с помощью параллельных вычислений, Вестник МГТУ им. Н.Э. Баумана. Серия «Приборостроение». – №6 (105). – 2015.

10 Минаев, В. А. Простые числа: новый взгляд на закономерности формирования, М.: Логос, 2011.

11 Минаев, В. А. Теорема о полном множестве простых чисел. М.: НИЯУ МИФИ, 2011.

12 Обстоятельно о подсчёте единичных битов [Электронный ресурс]. - 2016. - Режим доступа: https://habr.com/ru/post/276957/. - 21.02.2019.

13 Компрессия больших массивов простых чисел [Электронный ресурс]. - 2018. - Режим доступа: https://habr.com/ru/post/417753/. - 07.02.2019.

14 Диамонд, Г. Элементарные методы в изучении распределения простых чисел, УМН 45:2(272), 1990.

15 Постников А. Г., Романов Н. П. Упрощение элементарного доказательства А. Сельберга асимптотического закона распределения простых чисел, УМН, 10:4(66), 1955.

16 Selberg, A. An Elementary Proof of the Prime Number Theorem, Ann. Math. 50, 1949.

Приложение А
(рекомендуемое)

Результаты тестирования алгоритмов на отрезке от 1 до $2 * 10^9$. Оптимизация выключена

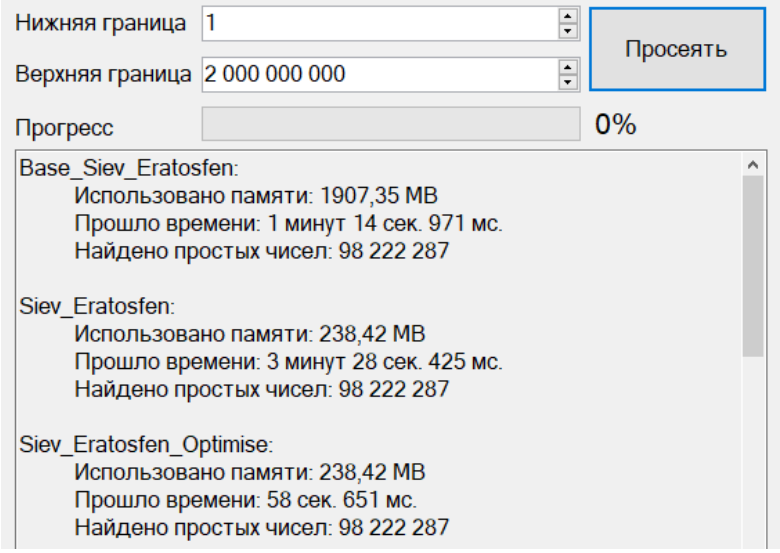

Рисунок А.1 – Результаты тестирования 1, 2 и 3 алгоритмов

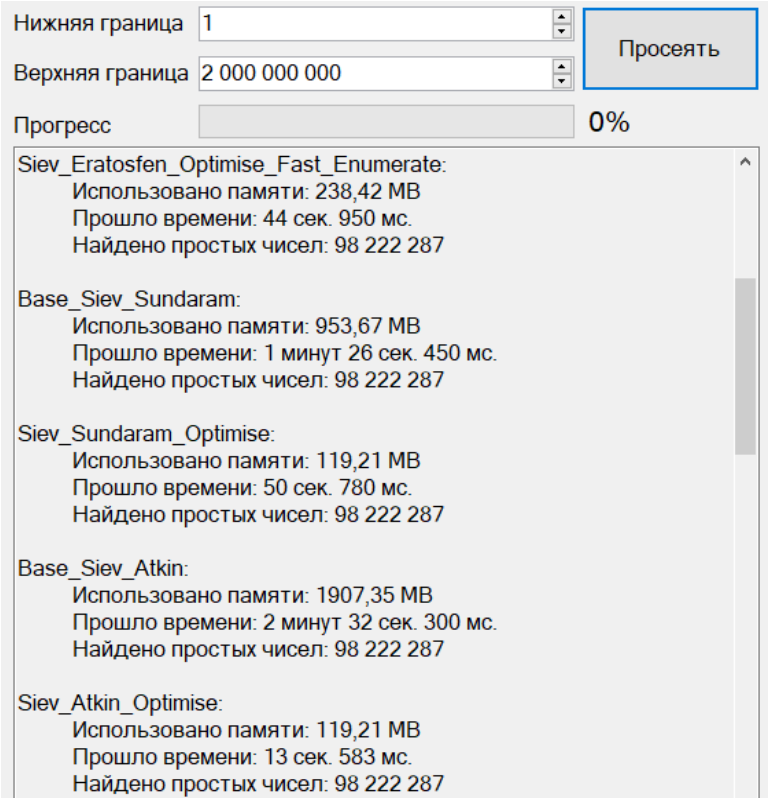

Рисунок А.2 – Результаты тестирования 4, 5, 6, 7 и 8 алгоритмов

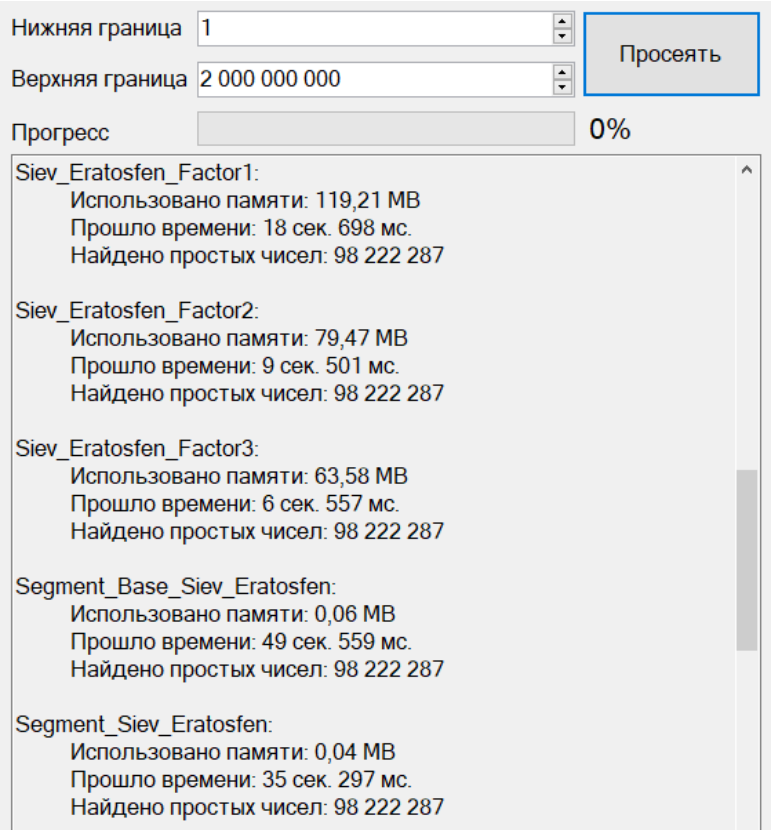

Рисунок А.3 – Результаты тестирования 9, 10, 11, 12 и 13 алгоритмов

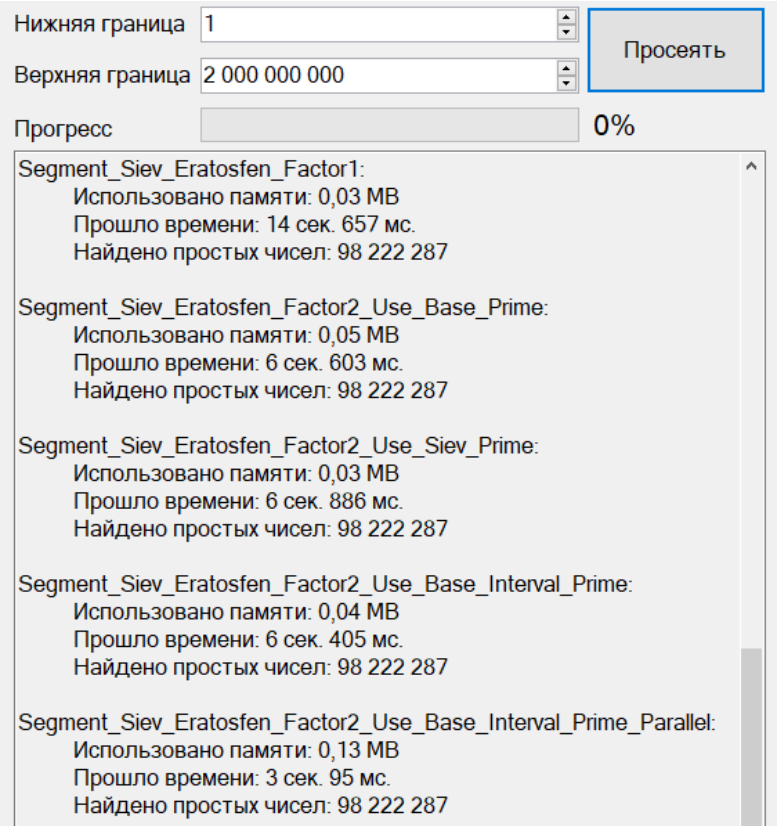

Рисунок А.4 – Результаты тестирования 14, 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке от 1 до $2 * 10^9$. Оптимизация включена

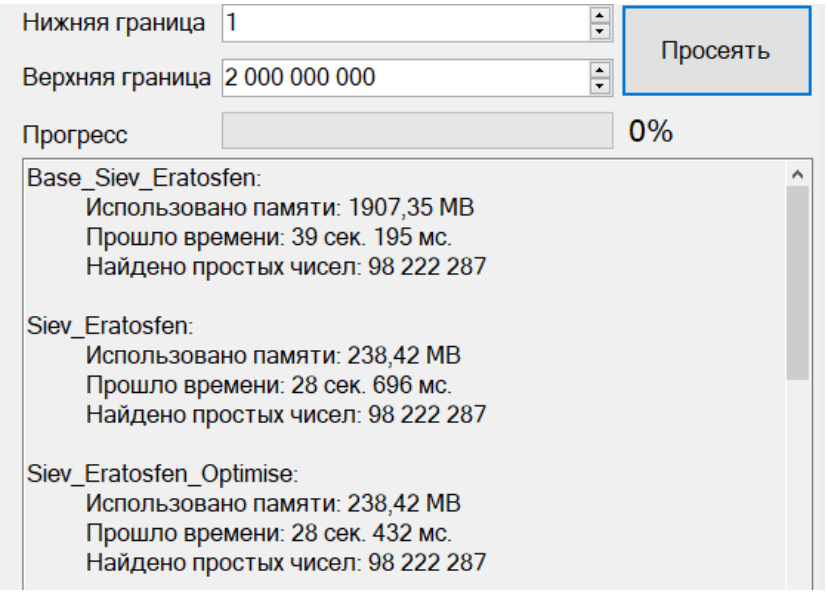

Рисунок А.5 – Результаты тестирования 1, 2 и 3 алгоритмов

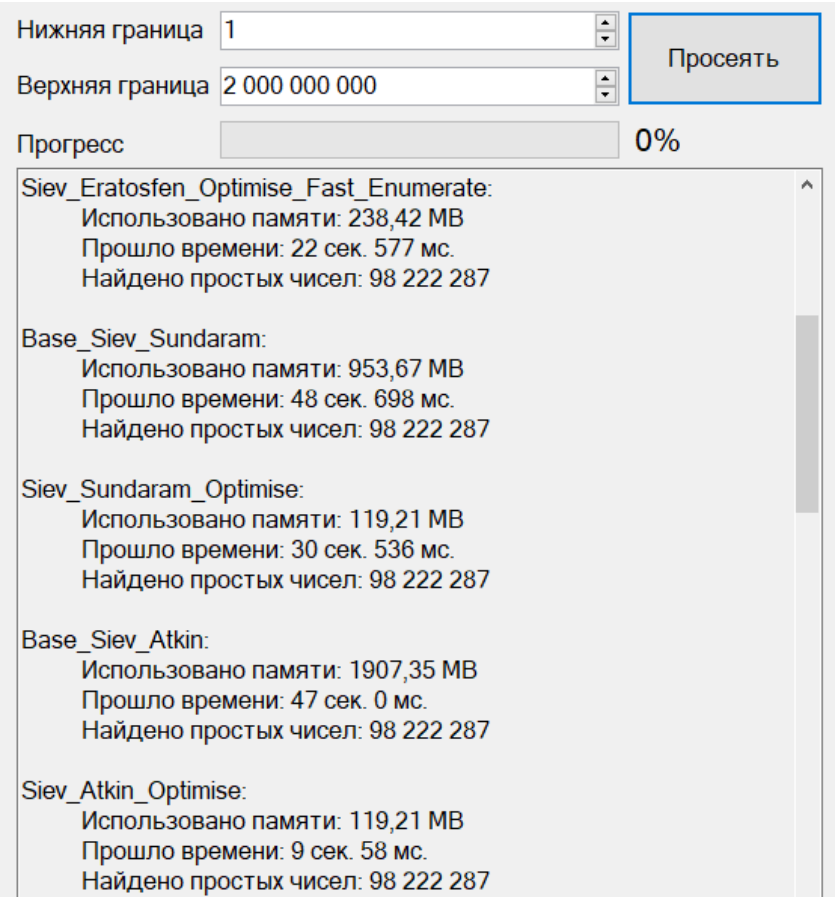

Рисунок А.6 – Результаты тестирования 4, 5, 6, 7 и 8 алгоритмов

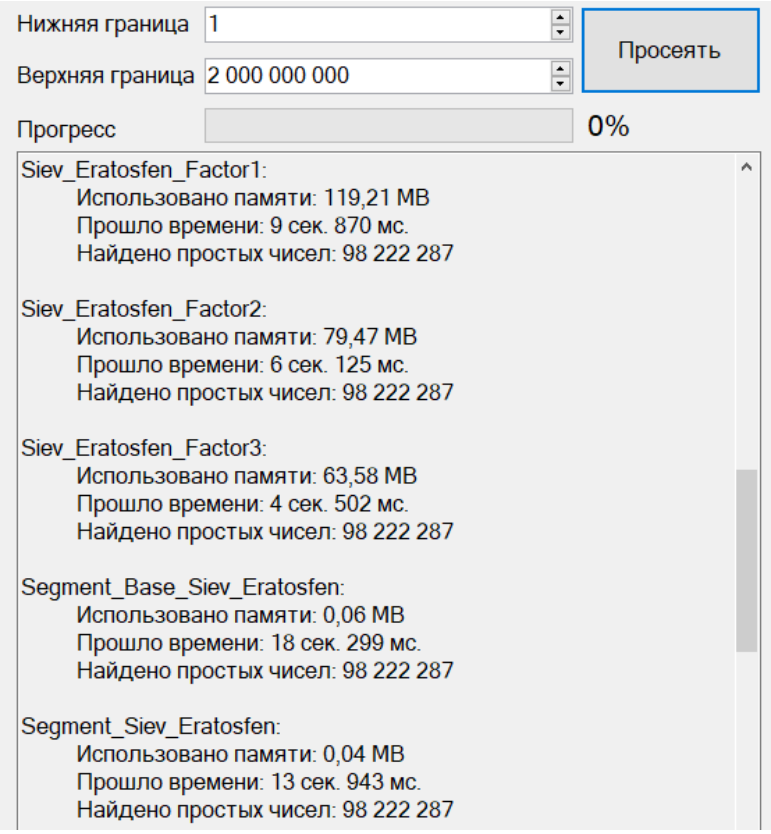

Рисунок А.7 – Результаты тестирования 9, 10, 11, 12 и 13 алгоритмов

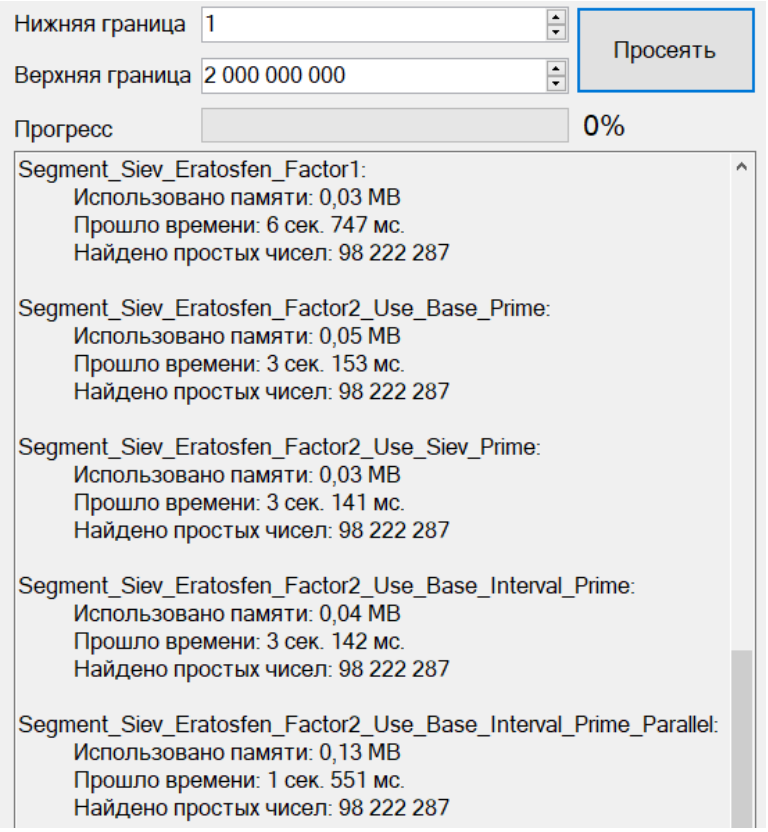

Рисунок А.8 – Результаты тестирования 14, 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке от 1 до 10^{10}

Нижняя граница 1

Верхняя граница 10 000 000 000

Просеять

Прогресс 0%

Siev_Sundaram_Optimise:
Использовано памяти: 596,05 MB
Прошло времени: 3 минут 13 сек. 47 мс.
Найдено простых чисел: 455 052 511

Siev_Atkin_Optimise:
Использовано памяти: 596,05 MB
Прошло времени: 52 сек. 357 мс.
Найдено простых чисел: 455 052 511

Рисунок А.9 – Результаты тестирования 6 и 8 алгоритмов

Нижняя граница 1

Верхняя граница 10 000 000 000

Просеять

Прогресс 0%

Siev_Eratosfen_Factor1:
Использовано памяти: 596,05 MB
Прошло времени: 57 сек. 16 мс.
Найдено простых чисел: 455 052 511

Siev_Eratosfen_Factor2:
Использовано памяти: 397,36 MB
Прошло времени: 34 сек. 389 мс.
Найдено простых чисел: 455 052 511

Siev_Eratosfen_Factor3:
Использовано памяти: 317,89 MB
Прошло времени: 25 сек. 967 мс.
Найдено простых чисел: 455 052 511

Segment_Base_Siev_Eratosfen:
Использовано памяти: 0,13 MB
Прошло времени: 1 минут 44 сек. 179 мс.
Найдено простых чисел: 455 052 511

Segment_Siev_Eratosfen:
Использовано памяти: 0,04 MB
Прошло времени: 1 минут 23 сек. 185 мс.
Найдено простых чисел: 455 052 511

Рисунок А.10 – Результаты тестирования 9, 10, 11, 12 и 13 алгоритмов

Нижняя граница 1

Верхняя граница 10 000 000 000

Просеять

Прогресс 0%

Segment_Siev_Eratosfen_Factor1:
Использовано памяти: 0,04 MB
Прошло времени: 38 сек. 595 мс.
Найдено простых чисел: 455 052 511

Segment_Siev_Eratosfen_Factor2_Use_Base_Prime:
Использовано памяти: 0,07 MB
Прошло времени: 17 сек. 296 мс.
Найдено простых чисел: 455 052 511

Segment_Siev_Eratosfen_Factor2_Use_Siev_Prime:
Использовано памяти: 0,04 MB
Прошло времени: 18 сек. 311 мс.
Найдено простых чисел: 455 052 511

Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime:
Использовано памяти: 0,04 MB
Прошло времени: 16 сек. 858 мс.
Найдено простых чисел: 455 052 511

Segment_Siev_Eratosfen_Factor2_Use_Base_Interval_Prime_Parallel:
Использовано памяти: 0,13 MB
Прошло времени: 8 сек. 109 мс.
Найдено простых чисел: 455 052 511

Рисунок А.11 – Результаты тестирования 14, 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке от 1 до 10^{11}

Нижняя граница 1

Верхняя граница 100 000 000 000

Просеять

Прогресс 0%

Siev_Sundaram_Optimise:
Использовано памяти: 5960,46 MB
Прошло времени: 50 минут 38 сек. 23 мс.
Найдено простых чисел: 4 118 054 813

Siev_Atkin_Optimise:
Использовано памяти: 5960,46 MB
Прошло времени: 17 минут 12 сек. 49 мс.
Найдено простых чисел: 4 118 054 813

Рисунок А.12 – Результаты тестирования 6 и 8 алгоритмов

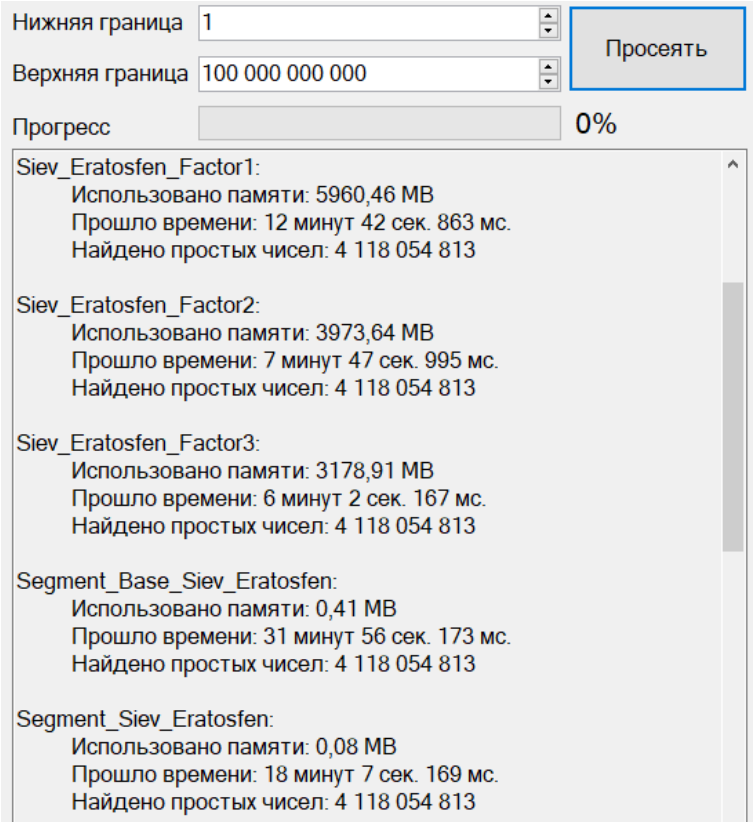

Рисунок А.13 – Результаты тестирования 9, 10, 11, 12 и 13 алгоритмов

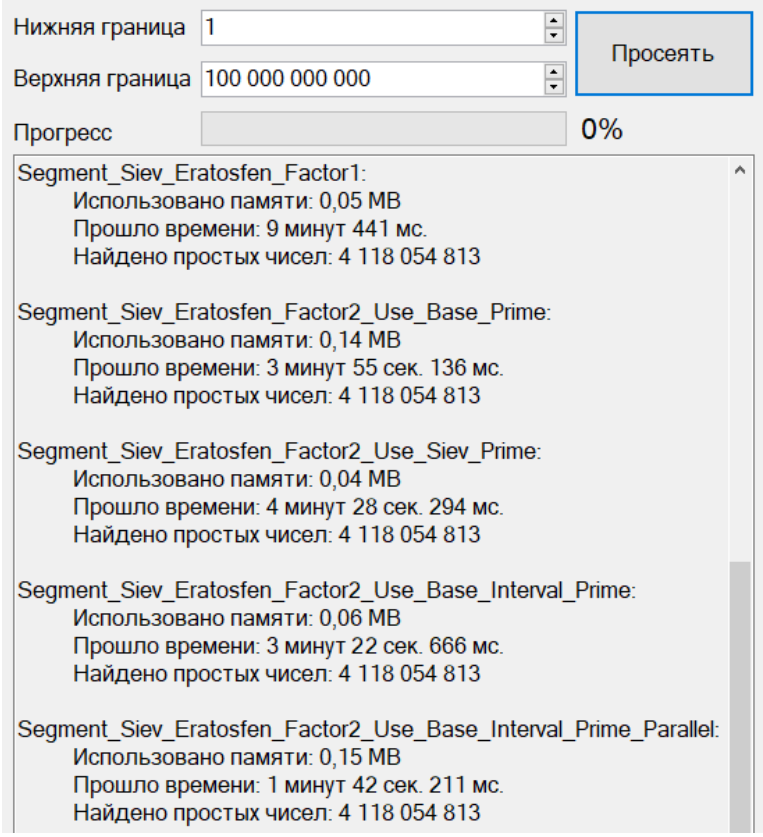

Рисунок А.14 – Результаты тестирования 14, 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке $[10^{12}, 10^{12} + 10^{10}]$

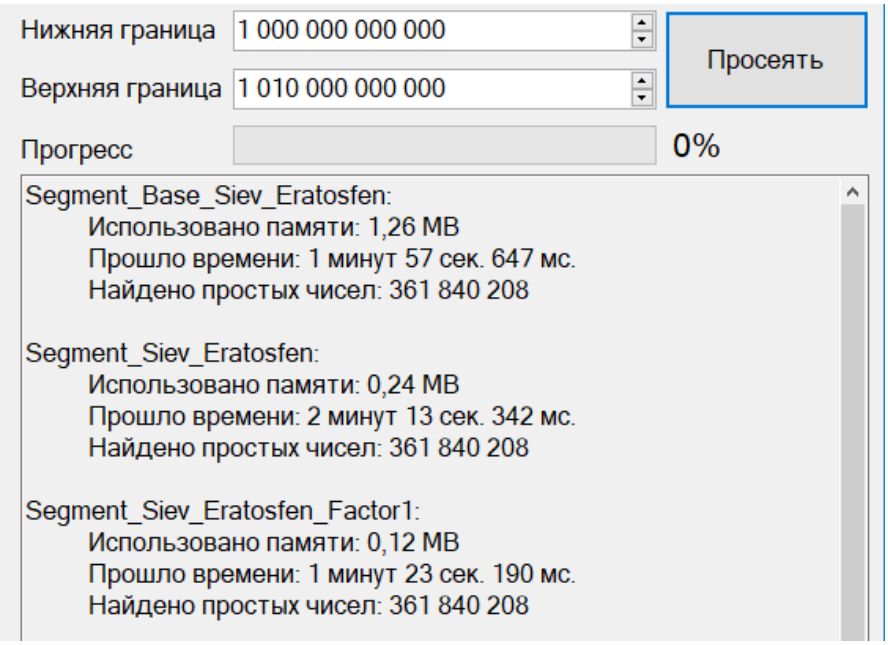

Рисунок А.15 – Результаты тестирования 12, 13 и 14 алгоритмов

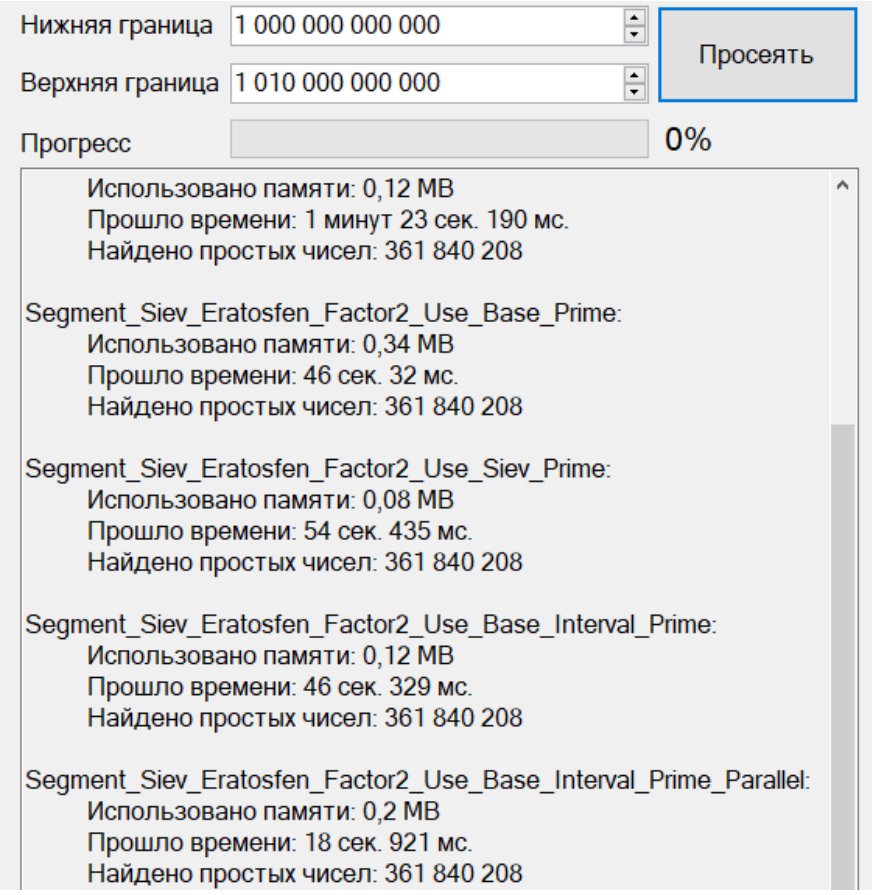

Рисунок А.16 – Результаты тестирования 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке $[10^{14}, 10^{14} + 10^{10}]$

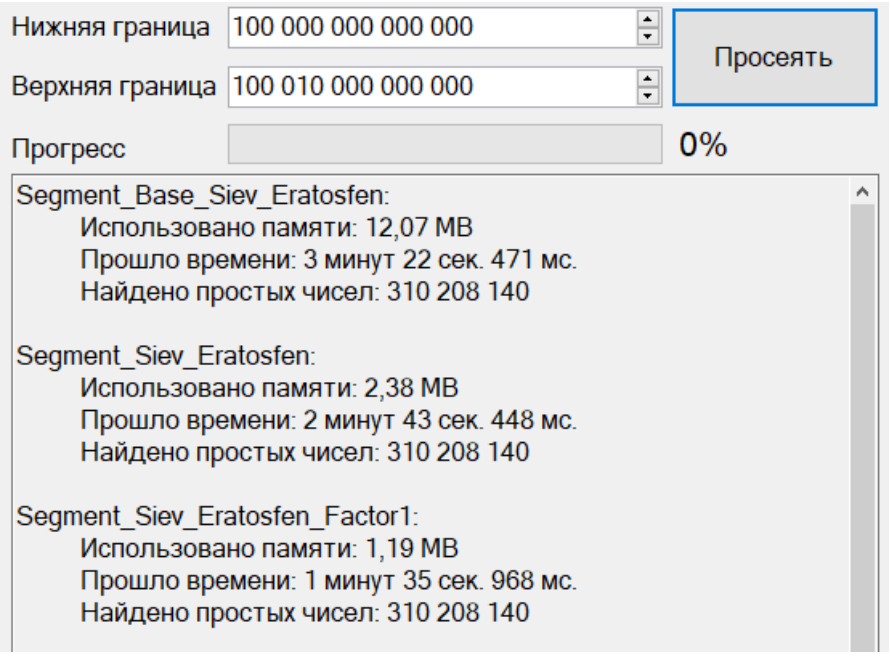

Рисунок А.17 – Результаты тестирования 12, 13 и 14 алгоритмов

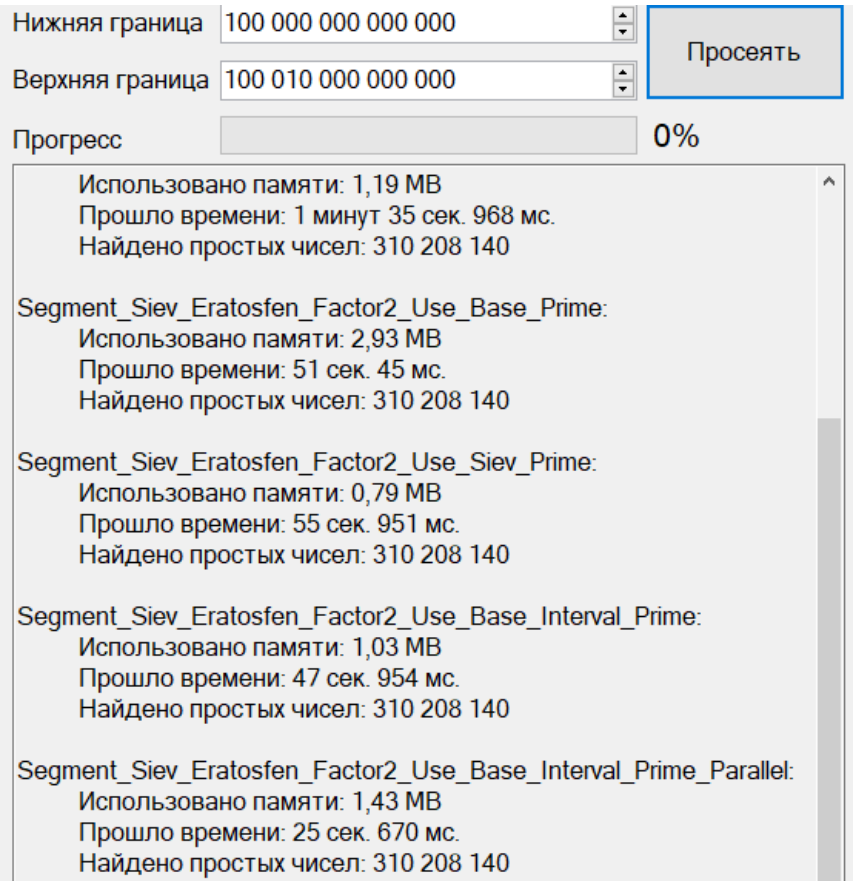

Рисунок А.18 – Результаты тестирования 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке $[10^{16}, 10^{16} + 10^{10}]$

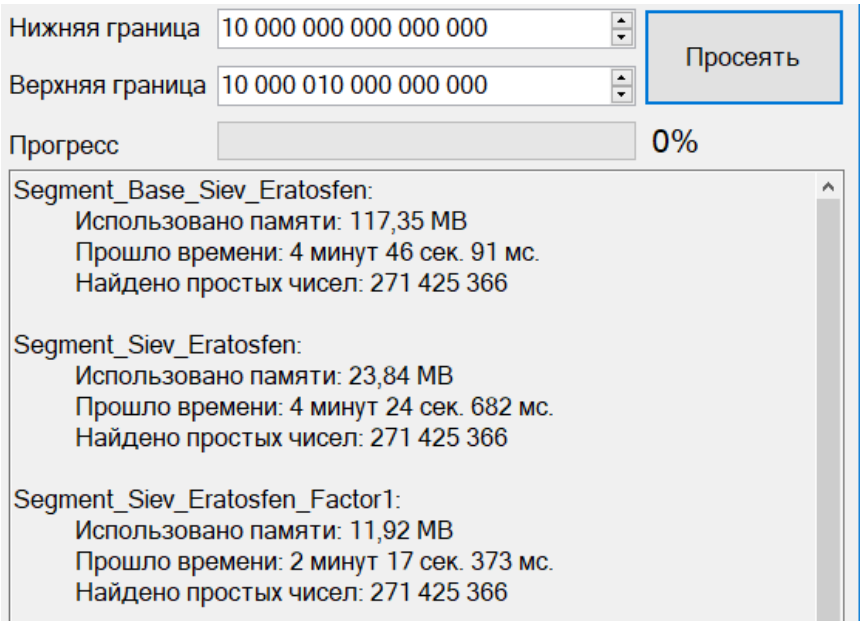

Рисунок А.19 – Результаты тестирования 12, 13 и 14 алгоритмов

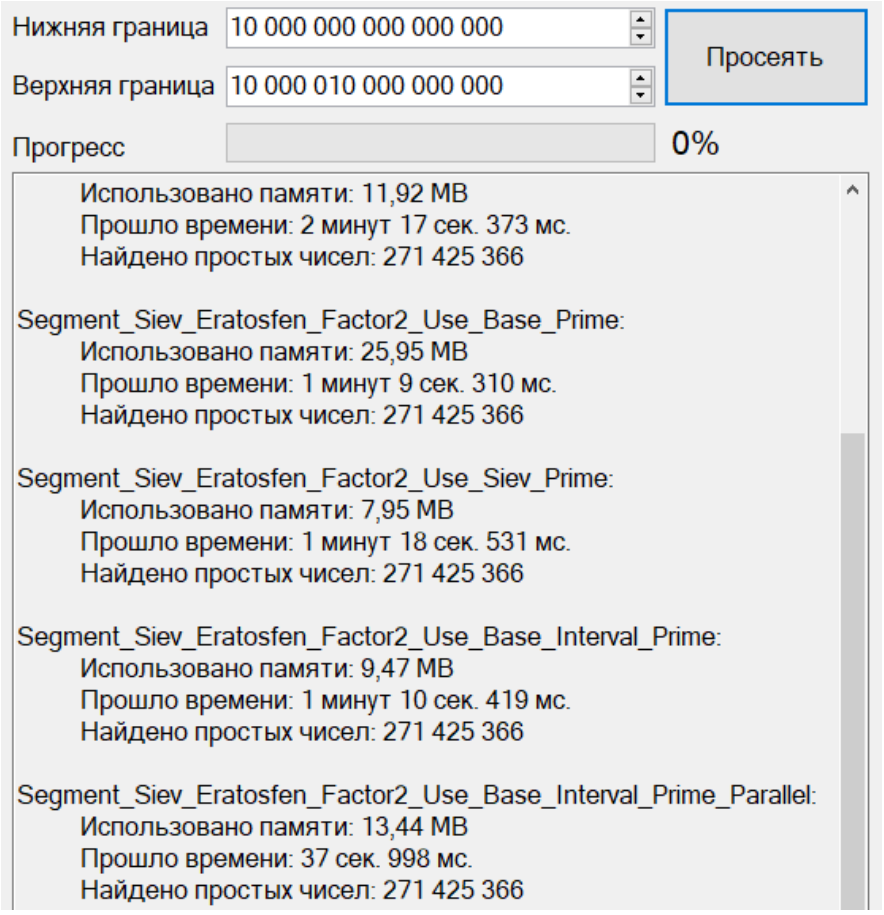

Рисунок А.20 – Результаты тестирования 15, 16, 17 и 18 алгоритмов

Результаты тестирования алгоритмов на отрезке $[10^{18}, 10^{18} + 10^{10}]$

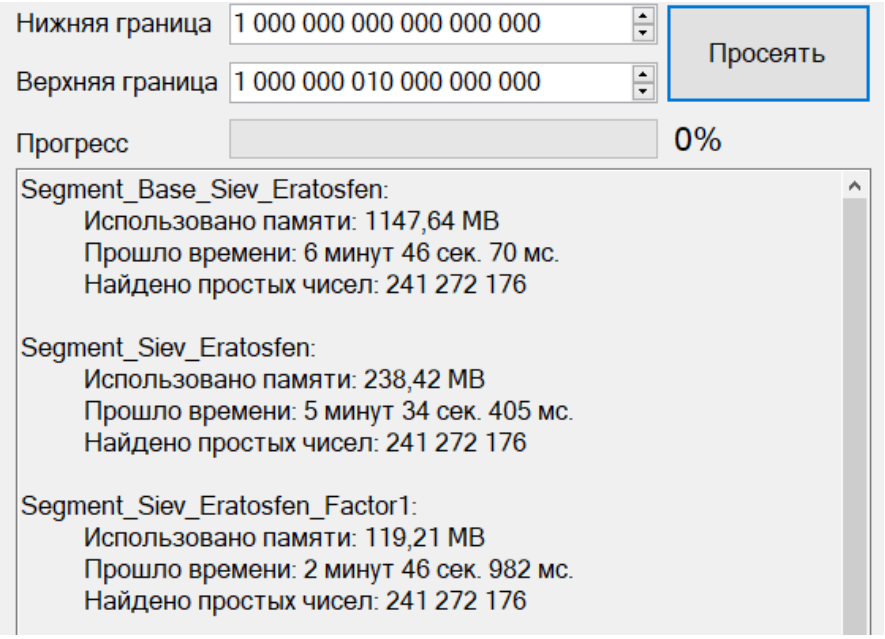

Рисунок А.21 – Результаты тестирования 12, 13 и 14 алгоритмов

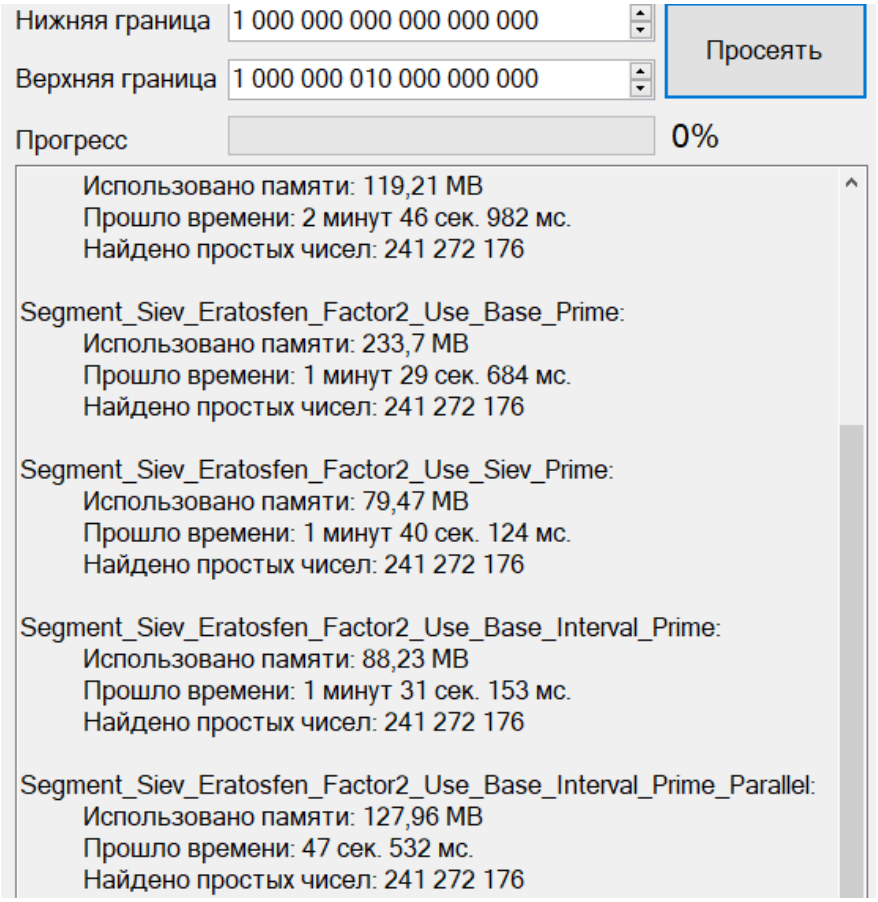

Рисунок А.22 – Результаты тестирования 15, 16, 17 и 18 алгоритмов