**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(национальный исследовательский университет)»

**Институт (Филиал)** №8 “Компьютерные науки и прикладная математика” **Кафедра** 806

**Группа** М80-407Б-18  **Направление подготовки** 01.03.02 Прикладная математика и информатика

**Профиль** Информатика

**Квалификация бакалавр**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРА**

На тему: Алгоритмы факторизации больших чисел

Автор ВКРБ Гамов Павел Антонович\_\_\_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Руководитель Ухов Петр Александрович (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Консультант (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Рецензент (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

**К защите допустить**

Заведующий кафедрой 806 Крылов Сергей Сергеевич (\_\_\_\_\_\_\_\_\_)

(№ каф) (фамилия, имя, отчество полностью)

\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_г.

Москва 2022

**РЕФЕРАТ**

Выпускная квалификационная работа бакалавра содержит: 40 страниц, 7 рисунков, 10 использованных источников, 1 приложение.

ФАКТОРИЗАЦИЯ, МЕТОД ПОЛЛАРДА, КВАДРАТИЧНОЕ РЕШЕТО, КРИПТОГРАФИЯ

Цель работы: исследование и анализ различных методов факторизации чисел. Программная реализация ρ-метода Полларда и алгоритма квадратичного решета, сравнение их между собой, а также поиск возможных улучшений каждого из алгоритмов.

В работе рассматривается программная реализация ρ-метода Полларда, его возможных улучшений, программная реализация метода квадратичного решета, его улучшений, а также сравнения данных методов между собой.

Разработаны программные последовательная и параллельная реализации ρ-метода Полларда, программная реализация алгоритма квадратичного решета и улучшения этапов его работы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ……………………………………………………………………..….4

Теоретическая часть

Основные обозначения и понятия (гладкие числа, факторизация ферма, простые числа, модульная арифметика)

Алгоритм Полларда (блок схема принцип работы алгоритма, метрики и для чего используется, привести больше понятий по принципу работы, от чего зависит скорость факторизации, что он расчитан на числа с маленькими составными числами)

Агоритм квадратичного решета (предыстория кем был открыт, награды, принцип работы, этапы алгоритма, матрика и доказательство метрики, сказать что зависит от длины факторизируемого числа)

Практическая часть

Реализация алгоритма Полларда (привести листинг кода, прогнать алгоритм пару раз на сете чисел, показать графики времени работы алгоритма, в том числе реализация многопроцессорного ускорения)

Реализация квадратичного решета (классическая реализация алгоритма, этапы, где дыры и узкие места, указать на то что можно улучшить, привести метрики работы на сете чисел)

Ускорение квадратичного решета (решето аткина, многопроцессорное просеивание, битовые срезы, описание как это устроено и почему оно ваще работает, как я к этому пришел, основные этапы того как я приходил к таким решениям)

1. Модульная арифметика…………………………………………………….….5
2. ρ метод Полларда………………………………………………………………5
3. Метод квадратичного решета………………………………………………....8
   1. Нахождение простых чисел. Формирование факторной базы………….9
   2. Отбор факторной базы …………………………………………………….12
   3. Символ Лежандра………………………………………………………….12
   4. Теорема Тоннели-Шенкса…………………………………...…………….12
   5. Генерирующий полином…………………………………………….…….14
   6. Этап просеивания…….…………………………………………………….14
   7. Составление матрицы...…………………………………………………….16
   8. Решение матрицы в конечном поле……………………………………….16

ЗАКЛЮЧЕНИЕ…………………………………………………………………….18

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ…..………………………….20

ПРИЛОЖЕНИЕ…………………………………………………………………….21

ВВЕДЕНИЕ

В настоящее время в любых сетях используется шифрование информации для защиты от вмешательства третьих лиц. Для шифрования используются различные криптосистемы, такие как RSA. Тот факт, что задача факторизации больших чисел есть задача предполагаемой большой вычислительной сложности, лежит в основе различных криптосистем. Факторизация или разложение на множители целых чисел – одна из древнейших проблем теории чисел. Методы факторизации целых чисел затрагивают такие области математики, как теория чисел, модульная арифметика, решение матричных уравнений. В настоящей работе рассматриваются и анализируются некоторые известные методы факторизации. При решении задачи факторизации затрагивается ряд других задач, непосредственно связанных с ней, таких как тесты на простоту и генерация больших простых чисел.

Пусть число n целое и составное, под термином факторизация будем иметь в виду нахождение чисел отличных от единицы и самого числа, произведение которых будет давать нам искомое число n. В случае, когда таких чисел нет, число n будет считаться простым, имеющим в качестве делителя само себя и единицу. В данной работе будут рассмотрены алгоритмы факторизации чисел, а также теоремы и алгоритмы, необходимые для проведения факторизации чисел.

1 Модульная арифметика

Прежде чем переходить к непосредственной факторизации, стоит оговорить о некоторых понятиях, которые будут в дальнейшем использоваться в разборе принципов работы тех или иных алгоритмов.

Пусть Z обозначает множество целых чисел, а – целое число. Условие, что целое число a делится нацело на b, будем записывать через . Говорят, что два целых числа a и b сравнимы по модулю p, записывается если

2 ρ метод Полларда

ρ - алгоритм — предложенный Джоном Поллардом в 1975 году алгоритм, служащий для факторизации целых чисел. Данный алгоритм основывается на алгоритме Флойда поиска длины цикла в последовательности и некоторых следствиях из парадокса дней рождения. Алгоритм наиболее эффективен при факторизации составных чисел с достаточно малыми множителями в разложении. Сложность алгоритма оценивается как O(N^(1/4)).

ρ - алгоритм Полларда строит числовую последовательность, элементы которой образуют цикл, начиная с некоторого номера n, что может быть проиллюстрировано, расположением чисел в виде греческой буквы ρ, что послужило названием семейству алгоритмов.

На рисунке 2.1 приводится минимальный код, из которого состоит основной блок факторизации методом Полларда

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2.1 – Алгоритм Полларда

Алгоритм, реализующий данный метод состоит из следующих шагов.

1. Выбирается натуральное число x0 < n и строится последовательность чисел {xn}, n = 0, 1, 2,…, где каждое следующее число определяется по формуле xm+1 = (xm2 + 1) mod n.
2. На каждом i-м шаге вычисляется d = НОД(n, |xi - xj|), где j < i. Можно ограничиваться парами вида (xi - xj), где j = 2k, где k = 1, 2, 3, …, i [2^k + 1, 2^(k+1)].
3. Когда будет найдено d не равное 1, вычисления заканчиваются и d – делитель числа n (не обязательно простой), если n / d – составное, то можно проверить алгоритм для числа n / d.

Таким образом ρ – алгоритм Полларда является вероятностным методом, для которого сложность вычисления, без учета логарифмической сложности нахождения НОД, оценивается как O(q^(1/2)) <= O(n^(1/4)). То есть можно сказать, что сложность зависит от размера делителя, а не от размеров самого числа, что может быть полезно для некоторых форматов чисел. Отсюда вытекает первый минус, алгоритм непредсказуем и для какого-то числа будет работать очень быстро, в силу своей вероятности, а для других чисел, потребует излишне много ресурсов и времени. Возникает идея оптимизации данного алгоритма, запуск нескольких потоков с различными стартовыми условиями, что, конечно, не дает нам линейного выигрыша, зато позволяет получить более большие шансы найти делитель числа. Вторая идея - использовать другие генерирующие многочлены.

Существует смысл использования алгоритма, зная, что факторизируемое число состоит из относительно маленьких множителей, скорость метода зависит от длинны такого числа, но что делать если число состоит из примерно равных по длине чисел, например, как числа RSA, в таком случае алгоритм может и не выдать ответ, сколько бы мы ни ждали. Нам нужна надежная оценка скорости, которая бы хоть и зависела бы от размеров числа, но для которой можно было бы найти примерное время факторизации.

На рисунке 2.2 представлен график сравнения скорости работы алгоритма Полларда на числах от 10 жо 20 десятичных знаков. График показывает, что прирост скорости составляет в среднем от 6 до 10 секунд на одну операцию факторизации.

Рисунок 2.2 – Время работы алгоритма Полларда

**3 Метод квадратичного решета**

Метод квадратичного решета - метод факторизации больших чисел, разработанный Померанцем в 1981 году. Долгое время превосходил другие методы факторизации целых чисел общего вида, не имеющих простых делителей, порядок которых значительно меньше корня из n. Этот метод считается вторым по быстроте (после общего метода решета числового поля). И до сих пор является самым быстрым для целых чисел до 100 десятичных цифр и устроен значительно проще чем общий метод решета числового поля. Это универсальный алгоритм факторизации, так как время его выполнения исключительно зависит от размера факторизуемого числа, а не от его особой структуры и свойств.

Алгоритм пытается найти такие квадраты чисел, которые равны по модулю n, что часто приводит к факторизации n. Алгоритм работает в два этапа: этап сбора данных, где он собирает информацию, которая может привести к равенству квадратов; и этап обработки данных, где он помещает всю собранную информацию в матрицу и обрабатывает её для получения равенства квадратов. Первый этап может быть легко распараллелен на много процессов, но второй этап требует большие объемы памяти и его трудно распараллелить.

В данной работе стоит разделить алгоритм не на два этапа, а на 3: формирование факторной базы, нахождение гладких чисел (просеивание), решение матрицы и формирование ответа. Связано это с особенностями гиперпараметра B, который определяет сколько простых чисел будет задействовано на втором этапе просеивания. В дальнейшем будет рассказано о данной оптимизации поподробнее.

**3.1 Нахождение простых чисел. Формирование факторной базы**

Первый этап алгоритма использует гиперпараметр B, который обозначает верхнюю границу факторной базы, которая будет состоять из простых чисел меньше заданного числа B. Данное множество простых чисел называется факторной базой для числа n с гиперпараметром B. Как известно, любое число можно представить в виде произведения простых чисел, числа, которые можно разложить в виде произведений простых чисел из факторной базы, называются гладкими числами по этому множеству.

Итак, у нас есть задача найти все простые числа до некой верхней планки, сразу на ум приходит алгоритм решета Эратосфена. Является базовым, но от этого не менее эффективным способом нахождения всех простых чисел до некой верхней границы, решето Эратосфена прекрасно справляется с нашей задачей. Так как в дальнейшем, нам возможно придется возвращаться к первому этапу, нам важно, чтобы алгоритм работал крайне быстро. Сложность алгоритма составляет O(n log log n).

На рисунке 3.1 приводится примерный псевдокод алгоритма решета Эратосфена.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3.1 – Алгоритм решета Эратосфена

Небольшое улучшение алгоритма поиска простых чисел, придуманное А. О. Л. Аткином и Д. Ю. Бернштайном, называется решетом Аткина. Скорость работы алгоритма соответствует скорости ранее известных алгоритмов просеивания, но требует меньше памяти и может работать на интервале. Хитрость алгоритма сводится к решению уравнений вида .

Все числа, равные (по модулю 60) 1, 13, 17, 29, 37, 41, 49 или 53, имеют остаток от деления на 4, равный 1. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 4x2 + y2 = n нечётно и само число не кратно никакому квадрату простого числа.

Числа, равные (по модулю 60) 7, 19, 31, или 43, имеют остаток от деления на 6, равный 1. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 3x2 + y2 = n нечётно и само число не кратно никакому квадрату простого.

Числа, равные (по модулю 60) 11, 23, 47, или 59, имеют остаток от деления на 12, равный 11. Эти числа являются простыми тогда и только тогда, когда количество решений уравнения 3x2 − y2 = n (для x > y) нечётно и само число n не кратно никакому квадрату простого.

Отдельный шаг алгоритма вычёркивает числа, кратные квадратам простых чисел. Так как ни одно из рассматриваемых чисел не делится на 2, 3, или 5, то, соответственно, они не делятся и на их квадраты. Поэтому проверка, что число не кратно квадрату простого числа, не включает 22, 32, и 52.

Для уменьшения требования к памяти, просеивание строится на некотором участке, равным примерно корню из верхней границы.

По оценке авторов алгоритм имеет асимптотическую сложность O(N / (log log N)) и требует O().

На рисунке 3.2 приводится график скорости работ блока нахождения простых чисел для факторной базы. Сравниваются алгоритм решета Эратосфена и решета Аткина.

Рисунок 3.2 – Время работы алгоритмов поиска факторной базы

**3.2 Отбор факторной базы**

Конечно, можно найти все простые числа до некоторой границы и использовать их для разложения всех гладких чисел, которые мы будем встречать в дальнейшем, но это не так эффективно. Мною был разработан алгоритм нахождения специальных простых чисел, которые будут давать нам прирост скорости при просеивании значений, полученных генерирующим полиномом.

**3.3 Символ Лежандра**

Символ Лежандра – функция, используемая в теории чисел. Является частным случаем символа Якоби. Определение его звучит примерно так: пусть а – целое число и p – простое число, отличное от 2. Символ Лежандра определяется следующим образом:

, если а делится на p.

, если а является квадратичным вычетом по модулю p (то есть существует такое целое x, что , но при этом а не делится на p.

, если а является квадратичным невычетом по модулю p.

Существуют различные способы нахождения символа Лежандра, я использую самый простой их них, так как встроенные функции работы с длинными числами в языке Python позволяют брать модуль и целить на очень большие числа. Однако существуют и другие более быстрые способы вычисления данного символа, рекурсивные, или с помощью разложения символа Якоби.

**3.4 Теорема Тоннели-Шенкса**

Теперь перейдем к реализации моего ускорения, используя символ Лежандра получаем только те простые числа, для которых существует решение уравнения, то есть они являются квадратичным вычетом нашему факторизируемому числу.

Входные данные: p – нечетное простое число, n – целое число, являющееся квадратичным вычетом по модулю p.

Результат работы алгоритма: вычет R, удовлетворяющий сравнению .

1. Выделим степени двойки из p – 1, то есть пусть p – 1 = 2SQ, где Q нечетно, S больше или равняется 1.
2. Выберем произвольный квадратичный невычет z, то есть символ Лежандра для него равен -1, положим .
3. Пусть также , , M = S.
4. Выполняем цикл:
   1. Если , то алгоритм возвращает R.
   2. В противном случае в цикле находим наименьшее i, 0 < i < M, такое что , с помощью итерирования возведем в квадрат.
   3. Пусть и положим , , , , возвращаемся к началу цикла.

После нахождения решения сравнения R второе решение сравнения находится как p – R.

Сам код алгоритма будет приведен в приложении. Нам необходимо просто получить корни уравнения, они пригодятся в дальнейшем при просеивании. Дело в том, что данные корни дают возможность не проверять лишний раз числа, которые не делятся на какое-либо число из факторной базы, работает это так. При просеивании генерирующий многочлен дает значения из ограниченной сетки, корни, полученные для каждого из простых чисел, позволяют перескакивать с числа, которое делится, к следующему такому числу, минуя все внутренние, которые не делятся на данное простое число. Улучшение дает существенный прирост скорости, так как плотность гладких чисел может быть довольно низка. Небольшая предобработка на всех простых числах позволяет пропускать сразу множество регионов, в которых просто не существует гладких чисел.

**3.5 Генерирующий полином**

Этап просеивания начинается с генерирующего полинома. В этом алгоритме он представляет собой полином второй степени, подставляя в него число, мы получаем возможно гладкое число, которое в дальнейшем требуется проверить.

Как правило генерирующий полином выглядит как , где m – корень из числа, которое требуется факторизовать, округленное вниз, а n – число, которое требуется факторизовать. Сразу видно недостаток данного полинома, чем дальше уходит число x в положительную или отрицательную сторону, тем более большими становятся числа и, следовательно, тяжелее найти в них те, которые являются гладкими на нашем множестве простых чисел. Как правило, результат генерирующего полинома ищется по модулю числа n, чтобы избежать очень больших чисел. Но нам может не хватить чисел, требуемых в дальнейшем для решения матрицы. В таком случае выбирается другой гиперпараметр B, ограничивающий факторную базу. Это влечет за собой последствия, гладкие числа будет найти гораздо проще, но их потребуется в разы больше. При маленькой плотности гладких чисел мы никогда не сможем найти необходимый минимум.

**3.6 Этап просеивания**

Второй этап алгоритма квадратичного решета – этап просеивания. Заключается в формировании интервала из чисел, сформированный с помощью генерирующего многочлена. Далее используя ускорение из теоремы Тоннели – Шенкса, мы находим смещения для каждого простого числа из факторной базы, и начинаем делить. Если какое-либо из чисел после процесса деления на все степени каждого числа из факторной базы окажется равным единице, тогда данное число оказалось гладким, и мы вносим его в отдельный массив гладких чисел. Моя реализация после просеивания участка сразу формирует матрицу гладких чисел с их разложениями. Так как каждый отдельный фрагмент интервала считается независимо от других, данный этап прекрасно можно распараллелить, в моем случае используя модуль мультипроцессинга, я выделяю рабочих по размеру доступных ядер, в моем случае 4 ядра. Фактически, данное улучшение дает существенный прирост в скорости просеивания.

Выше я упоминал, что нам потребуется иногда возвращаться к первому этапу, связано это с тем, что имея данные о плотности и скорости нахождения гладких чисел, мы сможем примерно оценить время, которое нам потребуется на нахождение минимального объема гладких чисел. В случае если данное время очень велико, нам следует вернутся к прошлому этапу формирования факторной базы и поменять гиперпараметр, в надежде получить лучшую оценку по времени.

В коде нет реализации возврата к первому пункту, однако все данные выводятся в консоль, и пользователь может сам оценить время и скорость работы. В случае, если время не устраивает пользователя, он в праве начать менять гиперпараметр.

На рисунке 3.3 приводится график скорости работы алгоритмов просеивания. Представлено как обычное линейное просеивание на 1 ядре процессора, а также просеивание с использованием всех доступных ядер.

Рисунок 3.3 – Скорость работы алгоритмов просеивания

**3.7 Составление матрицы**

Финальный третий этап работы алгоритма осуществляет формирование матрицы по модулю два разложения всех найденных гладких чисел. Каждая строка является вектором разложения найденного числа.

**3.8 Решение матрицы в конечном поле**

Решение матрицы сводится к нахождению линейно зависимых строк, путем прямого прохода алгоритмом Гаусса. Как только будет найдена совокупность чисел, которая дает полный квадрат (значения вектора будет состоять из нулей и только), алгоритм переходит к проверке полученного решения. Используя первичное разложение, формируется правая часть из критерия Ферма. Полученный вектор дает квадрат числа, чтобы получить его корень необходимо просто поделить все степени в разложении на два. Таким образом получено одно из чисел, дальнейшие операции над которым могут дать ответ факторизации.

Так как сложность алгоритма Гаусса стремится к кубу, каких-то возможных улучшений для данного этапа найти очень сложно. Изначально, решение матрицы выполнялось в среде Python, но было принято решение использовать отдельный написанный модуль на языке C++, который дал существенный прирост в скорости. Основным улучшением можно назвать использование битовых срезов. Идея такого улучшения выводится из того, что при решении матрицы, числа не превосходят единицу, таким образом, строки матрицы можно представить в виде битого представления других чисел. Нам больше не требуется суммировать каждое число отдельно, применяя битовые операции над элементами матрицы, которые мы представили как битовые маски, мы сокращаем количество операций в разы. Решение спорное, так как тратится время на перевод изначальной матрицы в другой формат, а после формирования ответа нам необходимо привести матрицу к изначальному формату.

На рисунке 3.4 приводится график скорости подходов к решению матрицы. Использование битовых срезов и организация кода на языке C++ дает потенциальный выигрыш в скорости работы.

Рисунок 3.4 – Время работы алгоритмов решения матрицы

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе бакалавра были рассмотрены возможные улучшения алгоритма квадратичного решета. Сравнивая его с алгоритмом Полларда, мы получили субэкспотенциальный, не вероятностный алгоритм, который имеет простор для воображения и дальнейшего улучшения. На рисунке 4.1 представлен график, на котором явно показано улучшение по времени работы алгоритма.

Рисунок 4.1 – Итоговое время

Данная работа может являться отправной точкой для дальнейших исследований в области факторизации чисел. Работы в данной области ведутся очень активно по всему миру, неизвестно, получится ли найти пути ускорения существующих алгоритмов факторизации. В представленной реализации много, что можно сделать по-другому, написать весь код на более быстром языке программирования, например на языке C++, использовать более быстрые алгоритмы нахождения простых чисел, ускорить этап просеивания, путем распределенных вычислений или иным методом работы. Использовать Cuda ядра и перенести основные вычисления на видеокарты. Для ускорения решения матрицы в конечных полях существует алгоритм Видемана и прочие возможности для улучшения скорости работы алгоритма над разряженными матрицами.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Шнайер Б. Прикладная криптография. —Москва: Триумф, 2013.
2. Коблиц Н. Курс теории чисел и криптографии. —Москва: Научное издательство «ТВП», 2001.
3. Василенко О.Н. Теоретико – числовые алгоритмы в криптографии. —М.: МЦНМО, 2003.
4. Ишмухаметов Ш.Т. Методы факторизации натуральных чисел: учебное пособие —Казань: Казан. Ун., 2011.
5. Brent R. P.Some parallel algorithms for integer factorization—Proc. Fifth International Euro-Par Conference (Toulouse, France, 1-3 Sept 1999), Lecture Notes in Computer Science 1685, Springer, 1999.
6. Koblitz N.A. Course in number theory and cryptography — Springer-Verlag, 1987. — С. 180—182.
7. Gerver J.L. Factoring large numbers with a quadratic sieve. — Math. Comp., 1983. — Vol. 41. — P. 287—294.
8. Коэн А. A Course in Computational Algebraic Number Theory — 4th Print Edition — Берлин, Гейдельберг, Нью-Йорк: Springer, 2000. — 550 с. — (Graduate Texts in Mathematics) — ISBN 978-3-540-55640-4 — ISSN 0072-5285; 2197-5612
9. Montgomery P., Silverman R. D. An FFT extension to the P-1 factoring algorithm Math. Comp.— AMS, 1990. — Vol. 54, Iss. 190. — P. 839—854. — ISSN 0025-5718; 1088-6842 — doi:10.1090/S0025-5718-1990-1011444-3
10. Pollard J. M. Theorems on factorization and primality testing Mathematical Proceedings of the Cambridge Philosophical Society B. J. Green — Cambridge University Press, 1974. — Vol. 76, Iss. 3. — P. 521—528.

ПРИЛОЖЕНИЕ А

Листинг кода

https://github.com/pagamov/VKR

