**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\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (\_\_\_\_\_\_\_\_\_\_)

(фамилия, имя, отчество полностью)

Руководитель (\_\_\_\_\_\_\_\_\_\_)

(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: 41 страницы, 7 рисунков, 5 использованных источников, 1 приложения.

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**

РЕФЕРАТ 2

ВВЕДЕНИЕ 4

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 10
   3. Символ Лежандра 11
   4. Теорема Тоннели-Шенкса 11
   5. Генерирующий полином 12
   6. Этап просеивания 13
   7. Составление матрицы 14
   8. Решение матрицы в конечном поле 15

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.

**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

Определение 1.1 Говорят, что два целых числа a и b сравнимы по модулю p, записывается,

если

Отношение по модулю натурального числа обладает свойствами:

1. Симметричность
2. Рефлективность
3. Транзитивность

**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.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 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.

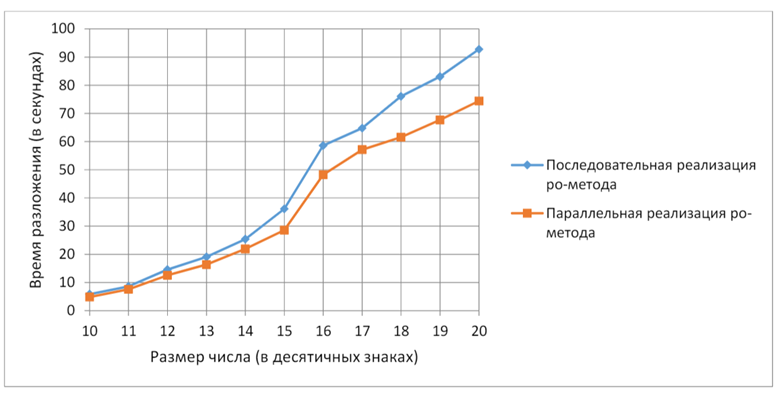


Рисунок 2 – Время работы алгоритма Полларда

**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.

**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).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 – Алгоритм решета Эратосфена

**Решето Аткина**

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()

Рисунок 4 – Время работы алгоритмов поиска факторной базы

**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.

**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.

**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 нам 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. Данное 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.

**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.

**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.

Рисунок 5 – Скорость работы алгоритмов просеивания

**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.

**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.

Рисунок 6 – 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 представлен график, на котором явно показано улучшение по времени работы алгоритма.

Рисунок 7 – 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. ИшмухаметовШ.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.

**ПРИЛОЖЕНИЕ**

Листинг файла index.py

if \_\_name\_\_ == "\_\_main\_\_":

from Factor import Factor

from time import time

import numpy as np

from data import n, B

from color import color

print("B "+color(B,"data"))

t = time()

res = Factor(n,B)

print(res)

Листинг файла Factor.py

from time import time

def Factor(n, B):

# color text!!!

from color import color

# some functions to support functionality (LOL)

from lib import GCD,Q

q = Q(n)

# class generating Factor base

from Primes import Primes

primes = Primes(n,B,q)

# func to suive until we have critical len of smooth numbers

from Suive import suive

t = time()

smooth\_numbers = suive(q,primes)

print("time:",color(round(time() - t,4),"time"))

print("Total number of smooth numberes:",color(len(smooth\_numbers),'data'))

print(color("All smooth numbers found",'strong')+'\n')

# matrix solves

print('Start making matrix\n')

t = time()

from Matrix\_solver import Matrix\_solver

matrix = Matrix\_solver(primes.p)

# form matrix with given smooth

for smooth in smooth\_numbers:

matrix.add(smooth[2])

print("Matrix builded in:",color(round(time() - t,4),"time"))

# possibe outcome [None,None] or if we LUCKY give ans as [gcd,n//gcd]

solve = matrix.solve(smooth\_numbers)

return solve

Листинг файла data.py

# N = number to be factorised

# n = 160769357899975610828199539114109518167531134514190990785144666932076614717841

# n = 1250171497372227982026555999675170108947918951378367343470923483104158597216632066586300921566811265776465427395026458151240042366061271512107752586681699923914902061886213022544496783070727061083763996630816279869169194623169255711135422521925444135939014878277515299870536875962948267973899545621728547726545192382593936985574978881305949487523233148677106330650818223443955800622774189936635106363035784698216185461573761714766211607812695281252356674432444279

# n = 97 \* 13

# n = 104729 \* 103591 # ~10^11

n = 104729 \* 21732382677641

n = 999998727899999 \* 21732382677641 # ~10^29

n = 21732382677641 \* 523347633027360537213687137 # ~10^40

# n = 3331113965338635107 \* 523347633027360537213687137 # ~10^45

# n = 618970019642690137449562111 \* 523347633027360537213687137 # ~10^53

# n = 10848981839

# RSA 100

# n = 1522605027922533360535618378132637429718068114961380688657908494580122963258952897654000350692006139

# RSA 110

# n = 35794234179725868774991807832568455403003778024228226193532908190484670252364677411513516111204504060317568667

# step to make in sieve process

step = 15000

# B = upper bound for Factor Base

B = 100000

# do we need to search for factor base? default = True

B\_search = True

# do we need to save our factor base? default = False

B\_save = False

# if B\_search == False, name file where we find factor base? default = primes.crypt

# file structure:

# data = [B, n, self.p, self.r]

B\_file = "primes.crypt"

# do we need to find smooth numbers for given B? default = True

Smooth\_search = True

# use parallel search instead of serial

Smooth\_search\_parallel = not Smooth\_search

# do we need to save our smooth base? default = False

Smooth\_save = False

# if Smooth\_search == False, name file where we find smooth numbers? default = smooth.crypt

# file structure:

# data = [B,n,smooth\_numbers[x,q(x),vector]]

Smooth\_file = "smooth.crypt"

Parallel\_max\_processes = 4

Verbose\_level = 1

"""level of output detail"""

# do we need to form matrix? default = True

Matrix\_solve = True

# do we need to save our matrix? default = False

matrix\_save = False

# if Matrix\_solve == False, name file where we find matrix? default = matrix.crypt

# file structure:

# B

# matrix for smooth numbers

Matrix\_file = "matrix.crypt"

Листинг файла gaus\_solve\_py.py

import ctypes

import pathlib

import numpy as np

from cfiles.numpyctypes import c\_ndarray

# Load the shared library into ctypes

libname = pathlib.Path().absolute() / "gaus\_solve.so"

c\_lib = ctypes.CDLL(libname)

def gaus\_solve(input\_matrix):

ret = np.empty(input\_matrix.shape, dtype='uint8')

par1 = c\_ndarray(input\_matrix, dtype = 'uint8', ndim = 2)

par2 = c\_ndarray(ret, dtype = 'uint8', ndim = 2)

N = c\_lib.test(par1, par2)

return ret[:,:N]

Листинг файла gaus\_solve.cpp

#include <iostream>

#include <vector>

#include <unordered\_set>

#include "cfiles/ndarray.h"

class cBiteMatrix {

class column;

public:

cBiteMatrix(int i, bool diag = false): n(i), m(i) {

data.resize(n, column(m));

if(diag) {

for(int i = 0; i < n; i++) {

data[i].put(1, i);

}

}

}

cBiteMatrix(Ndarray<unsigned char,2>& from): n(from.getShape(1)), m(from.getShape(0)) {

data.resize(n, column(m));

for(int i = 0; i < n; i++) {

for(int j = 0; j < m; j++) {

data[i].put(from[j][i], j);

}

}

}

void addCollumnToAnother(int i, int k) {

data[k] ^= data[i];

}

column& operator[](int i) {

return data[i];

}

private:

class column {

public:

std::vector<unsigned long long> data;

column(int i){

if(i % 64) {

data.resize(i / 64 + 1, 0);

} else {

data.resize(i / 64, 0);

}

}

int operator[](int i) {

return (data[i / 64] & (1ull << (i % 64))) ? 1 : 0;

}

void operator^=(const column& from) {

for(int i = 0; i < data.size(); i++ ) {

data[i] ^= from.data[i];

}

}

void put(int a, int i) {

if (a % 2) {

data[i / 64] |= (1ull << (i % 64));

} else {

unsigned long long mask = 1ull << (i % 64);

mask = ~mask;

data[i / 64] &= mask;

}

}

bool isZeroRow() {

for(int i = 0; i < data.size(); i++) {

if (data[i] != 0) {

return false;

}

}

return true;

}

int findFirstOne(std::unordered\_set<int>& banned\_columns) {

int i = 0;

while(i < data.size()) {

while(i < data.size()) {

if (data[i] == 0) {

i++;

}

else {

break;

}

}

if (i == data.size()) {

return -1;

} else {

for(int k = 0; k < 64; k++) {

if(data[i] & (1ull << k)) {

if (banned\_columns.find(i \* 64 + k) == banned\_columns.end())

return i \* 64 + k;

}

}

}

i++;

}

return -1;

}

};

int n;

int m;

std::vector<column> data;

};

extern "C" int test(numpyArray<unsigned char> A, numpyArray<unsigned char> B) {

Ndarray<unsigned char,2> matrix\_from(A);

Ndarray<unsigned char,2> ret\_matrix(B);

std::cout << "deliting zero rows\n";

int N\_old = matrix\_from.getShape(0);

cBiteMatrix nonCutMatrix(N\_old);

int N = 0;

bool del;

for(int i = 0; i < N\_old; i++) {

del = true;

for(int j = 0; j < N\_old; j++) {

if(matrix\_from[i][j] % 2 > 0){

del = false;

break;

}

}

if(!del) {

for(int j = 0; j < N\_old; j++) {

nonCutMatrix[j].put(matrix\_from[i][j], N);

}

N++;

}

}

cBiteMatrix matrix(N);

for(int k = 0; k < N; k++) {

for(int i = 0; i < matrix[k].data.size(); i++) {

matrix[k].data[i] = nonCutMatrix[k].data[i];

// std::cout << matrix[k][i] << "rows were deleted\n";

}

}

std::cout << N\_old - N << "rows were deleted\n";

// std::cout << "making bit matrix\n";

// cBiteMatrix matrix(matrix\_from);

std::cout << "making lineal columns support matrix\n";

cBiteMatrix lineal\_cols(N, true);

std::unordered\_set<int> banned\_columns;

std::cout << "start solving\n";

int x;

for(int i = 0; i < N - 1; i++)

{

std::cout << "progress: " << (((float)i / N) \* 100) << "\t % \r";

x = matrix[i].findFirstOne(banned\_columns);

banned\_columns.insert(x);

if(x >= 0) {

for(int j = i + 1; j < N; j++) {

if(matrix[j][x] == 1 && i != j) {

matrix.addCollumnToAnother(i, j);

lineal\_cols.addCollumnToAnother(i, j);

}

}

}

}

std::cout << "\n\nPreparing ansver\n";

int z = 0;

for(int i = 0; i < N; i++) {

std::cout << "progress: " << (((float)i / N) \* 100) << "\t % \r";

if(matrix[i].isZeroRow()) {

for(int k = 0; k < N; k++) {

ret\_matrix[k][z] = lineal\_cols[i][k];

}

z++;

}

}

std::cout << "\n\nPreparing ansver\n";

return z;

// return 1;

}

Листинг файла lib.py

# -\*- coding: utf-8 -\*-

import decimal,copy

from color import color

from math import log10

from time import time

import numpy as np

from debug\_info import smooth\_region\_output

def GCD(m,n):

mult = 1

if m > n:

m = m % n

elif n > m:

n = n % m

while True:

if m == 0 or n == 0 or m == n:

return mult\*max(n,m)

if m == 1 or n == 1:

return mult

mm2 = m % 2

nm2 = n % 2

if mm2 == 0 and nm2 == 0:

mult \*= 2

m = m//2

n = n//2

elif mm2 == 0 and nm2 != 0:

m = m//2

elif mm2 != 0 and nm2 == 0:

n = n//2

elif mm2 != 0 and nm2 != 0:

if n > m:

piv = (n-m)//2

n = m

m = piv

elif n < m:

m = (m-n)//2

def eratosthenes(n):

numbers = list(range(2, n + 1))

for number in numbers:

if number != 0:

for candidate in range(2 \* number, n+1, number):

numbers[candidate-2] = 0

return list(filter(lambda x: x != 0, numbers))

class Q:

def \_\_init\_\_(self,n):

self.n = n

self.m = int(decimal.Decimal(n).sqrt() + 1)

def \_\_call\_\_(self,x):

return (x + self.m)\*\*2 - self.n

def legendre(a, p):

return pow(a, (p - 1) // 2, p)

def tonelli(n, p):

assert legendre(n, p) == 1, "not a square (mod p)"

q = p - 1

s = 0

while q % 2 == 0:

q //= 2

s += 1

if s == 1:

return pow(n, (p + 1) // 4, p)

z = 2

for z in range(2, p):

if p - 1 == legendre(z, p):

break

c = pow(z, q, p)

r = pow(n, (q + 1) // 2, p)

t = pow(n, q, p)

m = s

t2 = 0

while (t - 1) % p != 0:

t2 = (t \* t) % p

for i in range(1, m):

if (t2 - 1) % p == 0:

break

t2 = (t2 \* t2) % p

b = pow(c, 1 << (m - i - 1), p)

r = (r \* b) % p

c = (b \* b) % p

t = (t \* c) % p

m = i

return r

def smooth\_region(L1, L2, q, primes):

t = time()

# все значения между L1 и L2

res0 = list(range(L1, L2))

# единица означает что число под этим индексом гладкое

res1 = np.array([q(x) for x in range(L1, L2)])

# массив из разложений чисел по простым

res2 = np.zeros((len(res0), len(primes)), dtype="int8")

table\_creation\_time = time() - t

t = time()

primes\_skipped = 0

# подгоняем r до [L1, L2] => получаем s

# инициализируем массив сдвинутых r (делаем для того, чтобы не искать с

# нуля, а искать уже в [L1, L2])

s = [[] for \_ in range(len(primes.r))]

for smooth\_idx, prime in enumerate(primes):

# print("\rshift "+'\033[92m'+str(round(float(i)/float(len(primes))\*100,2))+'\033[0m'+" %",end="")

for r in primes.r[smooth\_idx]:

# получаем примерную оценку, сколько праймов надо пропустить

k = L1 // prime

# уточняем оценку

while r + k\*prime >= L1:

k -= 1

k+=1

# если полученное число выходит за рамки области, то в дальнейшем,

# мы его вообще проверять не будем (if просто для статистики)

if r + k\*prime >= L2:

primes\_skipped += 1

# при просеивании, начинаем сразу с этой позиции

s[smooth\_idx].append(r + k\*prime)

s\_search\_time = time() - t

t = time()

for prime\_idx, prime in enumerate(primes):

# print("\rsuive "+'\033[92m'+str(round(float(p)/float(len(primes))\*100,2))+'\033[0m'+" %",end="")

for s\_i in s[prime\_idx]:

if s\_i < L2:

# гарантируем что начиная с s\_1 - L1, каждое число через prime

# делится на этот prime хотя бы 1 раз

res2[s\_i - L1::prime, prime\_idx] += 1

res1[s\_i - L1::prime] //= prime

# дальнешее деление проверяется вручную

for smooth\_idx in range(s\_i, L2, prime):

x = smooth\_idx - L1

while res1[x] % prime == 0:

res1[x] //= prime

res2[x, prime\_idx] += 1

prime\_div\_time = time() - t

t = time()

ans = []

for smooth\_idx, qx in enumerate(res1):

if abs(qx) == 1:

ans.append([res0[smooth\_idx],q(res0[smooth\_idx]),np.copy(res2[smooth\_idx])])

# if abs(res1[smooth\_idx]) == 1:

answer\_fill\_time = time() - t

smooth\_region\_output(table\_creation\_time, s\_search\_time, prime\_div\_time,

answer\_fill\_time, ans, L1, L2, primes\_skipped, primes)

return ans

# TODO:

# + implement slice handling

def get\_region(idx, step):

"""

return unique region for specified index

if index is even: return positive

odd: return negative

"""

# четные: положительный знак, нечетные: отрицательный

sign = (idx % 2) \* (-1)

if sign == 0:

return [idx \* step, (idx + 1) \* step]

else:

return [- (idx) \* step, - (idx - 1) \* step]

Листинг файла Matrix\_solver.py

import copy

from time import time

import numpy as np

from color import color

from data import n

from lib import GCD

from gaus\_solve\_py import gaus\_solve

import decimal

class Matrix\_solver:

def \_\_init\_\_(self, primes):

self.matrix = []

self.primes = primes

self.gaus = []

self.num\_smooth\_numbers = 0

self.matrix = np.zeros((len(primes), len(primes)), dtype='uint8')

# for i in range(len(primes)):

# self.matrix.append([])

def add(self, smooth\_number):

if(self.num\_smooth\_numbers == len(self.primes)):

return False

self.matrix[:,self.num\_smooth\_numbers] = smooth\_number

self.num\_smooth\_numbers += 1

# for i in range(len(self.primes)):

# self.matrix[i].append(smooth\_number[i])

def solve(self,smooth\_numbers):

# c++ matrix solve

t = time()

ans = [None,None]

# lineal\_rows = gaus\_solve(self.gaus)

lineal\_rows = gaus\_solve(self.matrix)

print("\ndone solving matrix in",color(round(time() - t,4),'time'))

t = time()

for zero\_column in range(len(lineal\_rows[0])):

t1 = time()

print("\nfind",color("posible",'data'),"ans")

b = [] #indexes of smooth numbers i guess

for i in range(len(lineal\_rows)):

if lineal\_rows[i][zero\_column]:

b.append(i)

# got vector b as indexes of posible ans

print("\nseg1",color(round(time() - t1,4),'time'))

t1 = time()

left = 1

right = []

for i in b:

left \*= int(decimal.Decimal(n).sqrt() + 1) + smooth\_numbers[i][0]

right.append(smooth\_numbers[i][2])

true\_right = int(1)

right\_piv = np.zeros(len(self.primes), int)

# right\_piv = [0] \* len(self.primes)

print("\nseg2",color(round(time() - t1,4),'time'))

t1 = time()

print(len(right), right[0][0])

for r in right:

right\_piv += r

print("\nseg3",color(round(time() - t1,4),'time'))

t1 = time()

for j in range(len(right\_piv)):

right\_piv[j] //= 2

print("\nseg4",color(round(time() - t1,4),'time'))

t1 = time()

for j in range(len(right\_piv)):

true\_right \*= pow(self.primes[j], int(right\_piv[j]))

print("\nseg5",color(round(time() - t1,4),'time'))

t1 = time()

gcd = min(GCD(abs(int(left+true\_right)), n), GCD(abs(int(left-true\_right)), n))

if gcd > 1 and n // gcd \* gcd == n:

print("\nseg6",color(round(time() - t1,4),'time'))

print(color("Solve Done",'strong'))

ans = [gcd, n//gcd]

break

else:

print("\nseg6",color(round(time() - t1,4),'time'))

print("guess was",color('wrong', 'strong'),"keep search!\n")

print("\ndone searching ans's",color(round(time() - t,4),'time'))

return ans

Листинг файла Primes.py

from lib import eratosthenes,Q,tonelli,legendre

from time import time

import numpy as np

import pickle

from data import B\_search, B\_file, B\_save

from color import color

class Primes:

def \_\_init\_\_(self, n, B, q):

if B\_search:

# B\_search == True so we need to find out factor base

primes = eratosthenes(B)

self.p = []

self.r = []

q = Q(n)

t = time()

if legendre(n%primes[0], primes[0]) == 1:

tr = tonelli(n,primes[0])

r = [(tr - q.m) % primes[0]]

self.p.append(primes[0])

self.r.append(r)

# Для каждого прайма проверяем Лежандра и находим корни Тонелли Шенксом

for i in range(1,len(primes)):

print("\r"+color(round(float(i)/float(len(primes))\*100,2),"%"),end="")

if legendre(n%primes[i], primes[i]) == 1:

tr = tonelli(n,primes[i])

r = [(tr - q.m) % primes[i],(primes[i] - tr - q.m) % primes[i]]

self.p.append(primes[i])

self.r.append(r)

print("\nprimes done in time: "+color(round(time() - t,4),"time"))

print("primes len",color(len(self.p),"data"))

if B\_save:

# we need to save our factor base in file B\_file

t = time()

data = [B, n, self.p, self.r]

with open(B\_file, 'wb') as f:

pickle.dump(data, f)

print("\nprimes saved in time: "+color(round(time() - t,4),"time"),"in file: "+str(B\_file))

else:

# B\_search == False so we need to upload factor base from B\_file

t = time()

data = []

with open(B\_file, 'rb') as f:

data = pickle.load(f)

if data[0] != B:

print(color("Primes::Error: wrong B!","strong"))

exit()

if data[1] != n:

print(color("Primes::Error: wrong n!","strong"))

exit()

self.p = data[2]

self.r = data[3]

print(color("primes upload in time:","strong")+color(round(time() - t,4),"time"),"from file: "+str(B\_file))

print("primes len",color(len(self.p),"data"))

def \_\_getitem\_\_(self,i):

return self.p[i]

def \_\_len\_\_(self):

return len(self.p)

Листинг файла Suive.py

# -\*- coding: utf-8 -\*-

from data import Smooth\_search, Smooth\_file, Smooth\_save

from data import n, B, step

from data import Smooth\_search\_parallel

from color import color

from lib import smooth\_region

from debug\_info import suive\_output

# for multiprocessing

import multiprocessing as mp

from lib import get\_region

# indicate workers to stop

from ctypes import c\_bool

# settings for multiprocessing

from data import Parallel\_max\_processes as MAX\_PROCESSES

def worker\_task(region\_idx, stop\_working, answer\_queue, primes, q):

while True:

# process safe read and write operation

with region\_idx.get\_lock():

local\_region\_idx = region\_idx.value

region\_idx.value += 1

# exit if we reached maximum

if stop\_working.value:

break

region = get\_region(local\_region\_idx, step)

# calculate answer

answer = smooth\_region(\*region, q, primes)

# return it

answer\_queue.put(answer)

def suive(q, primes):

if Smooth\_search\_parallel:

mp.set\_start\_method('spawn')

# mp.set\_start\_method("spawn", force=True)

MAX\_SMOOTH\_QTY = len(primes)

print("MAX\_SMOOTH\_QTY:", MAX\_SMOOTH\_QTY)

# инициализируем lock-safe очередь для ответов

answer\_queue = mp.Queue()

# создаем работающие процессы

workers = []

# индекс региона

region\_idx = mp.Value('i', 0)

# нужно ли останавливаться воркерам

stop\_workers = mp.Value(c\_bool, False)

# инициализируем процессы

for worker\_id in range(MAX\_PROCESSES):

workers.append(mp.Process(

target=worker\_task,

args=(region\_idx, stop\_workers, answer\_queue, primes, q)

))

for worker in workers:

worker.start()

# читаем из очереди ответов в нашу локальную переменную

smooth\_numbers = []

while len(smooth\_numbers) < MAX\_SMOOTH\_QTY:

smooth\_numbers.extend(answer\_queue.get())

suive\_output(smooth\_numbers, primes)

else:

# чтобы следующие линии не продолжались вместе с прогресс баром

# надо поставить новую линию

print()

# говорим воркерам закончить

stop\_workers.value = True

# ждем пока они закончат

for idx, worker in enumerate(workers):

worker.kill()

return smooth\_numbers

elif Smooth\_search:

# Smooth\_search == True so we need to find our smooth numbers

print("step:",color(step,"data"))

k = 1

# found\_smooth = 0

smooth\_numbers = []

while q((k-1)\*step) < n:

ans = smooth\_region((k-1)\*step,k\*step,q,primes)

for i in range(len(ans)):

smooth\_numbers.append(ans[i])

ans = smooth\_region(-k\*step,-(k-1)\*step,q,primes)

for i in range(len(ans)):

smooth\_numbers.append(ans[i])

k+=1

print("\rTotal number of smooth numbers:",color(len(smooth\_numbers),'data'))

if len(smooth\_numbers) > len(primes):

# Выброс из функции

if Smooth\_save:

# we need to save our smooth numbers in file Smooth\_file

t = time()

data = [B, n, smooth\_numbers]

with open(Smooth\_file, 'wb') as f:

pickle.dump(data, f)

print("\nsmooth numbers saved in time: "+color(round(time() - t,4),"time"),"in file: "+str(Smooth\_file))

return smooth\_numbers

else:

# Smooth\_search == False so we need to upload factor base from Smooth\_file

t = time()

data = []

smooth\_numbers = []

with open(Smooth\_file, 'rb') as f:

data = pickle.load(f)

if data[0] != B:

print(color("Suive::Error: wrong B!","strong"))

exit()

if data[1] != n:

print(color("Suive::Error: wrong n!","strong"))

exit()

smooth\_numbers = data[2]

print(color("smooth upload in time:","strong")+color(round(time() - t,4),"time"),"from file: "+str(Smooth\_file))

print("smooth len",color(len(smooth\_numbers),"data"))

return smooth\_numbers