Оглавление

[Введение 3](#_Toc390034358)

[1. Постановка задачи 4](#_Toc390034359)

[2. Аналитическая часть 5](#_Toc390034360)

[2.1. Система шифрования RSA 6](#_Toc390034361)

[2.2. Сложность теоретико-числовых алгоритмов 10](#_Toc390034362)

[2.2.1. Алгоритм вычисления  11](#_Toc390034363)

[2.2.2. Алгоритм Евклида 11](#_Toc390034364)

[2.2.3. Алгоритм решения уравнения  12](#_Toc390034365)

[2.2.4. Алгоритм нахождения делителей многочлена  в кольце  14](#_Toc390034366)

[2.2. Качественная теория алгоритма RSA 17](#_Toc390034367)

[2.2.1. Алгоритм доказывающий не простоту числа 18](#_Toc390034368)

[2.2.2. Нахождение больших простых чисел 20](#_Toc390034369)

[2.2.3. Проверка большого числа на простоту 23](#_Toc390034370)

[3. Проектирование 30](#_Toc390034371)

[3.1. Проектирование функций 30](#_Toc390034372)

[3.2. Проектирование классов 32](#_Toc390034373)

[3.3. Проектирование классов исключений 34](#_Toc390034374)

[3.4. Модуль rsa.bigfile 35](#_Toc390034375)

[3.6. Модуль rsa.core 36](#_Toc390034376)

[4. Тестирование и документирование 37](#_Toc390034377)

[5. Экономическое обоснование разработки 38](#_Toc390034378)

[Заключение 39](#_Toc390034379)

[Список литературы 40](#_Toc390034380)

Введение

1. Постановка задачи
   1. Формулировка задачи

Безопасность передачи данных по каналам связи является актуальной. Современные компьютерные сети не исключение. К сожалению, в сетевых операционных системах (Windows NT/XP/Server, Novell и т.д.) иностранного производства, как следствие, из-за экспортных соображений уровень алгоритмов шифрования заметно снижен.

Исследовать современные методы шифрования и их применимость к шифрованию потоков данных. Разработать собственную библиотеку алгоритмов шифрования и программный продукт, демонстрирующий работу этих алгоритмов при передаче данных в сети.

* 1. Выводы

1. Аналитическая часть

Труды Евклида и Диофанта, Ферма и Эйлера, Гаусса, Чебышева и Эрмита содер­жат остроумные и весьма эффективные алгоритмы решения диофантовых уравнений, выяснения разрешимости сравнений, построения больших по тем временам простых чисел, нахождения наилучших приближений и т.д. В последние два десятилетия, благодаря в первую очередь запросам криптографии и широкому распространению ЭВМ, исследова­ния по алгоритмическим вопросам теории чисел переживают период бур­ного и весьма плодотворного развития.

Вычислительные машины и электронные средства связи проникли практически во все сферы человеческой деятельности. Немыслима без них и современная криптография. Шифрование и дешифрование текстов можно представлять себе как процессы переработки целых чисел при помощи ЭВМ, а способы, которыми выполняются эти операции, как неко­торые функции, определённые на множестве целых чисел. Всё это делает естественным появление в криптографии методов теории чисел. Кроме того, стойкость ряда современных криптосистем обосновывается только сложностью некоторых теоретико-числовых задач.

Но возможности ЭВМ имеют определённые границы. Приходится раз­бивать длинную цифровую последовательность на блоки ограниченной длины и шифровать каждый такой блок отдельно. Мы будем считать в дальнейшем, что все шифруемые целые числа неотрицательны и по вели­чине меньше некоторого заданного (скажем, техническими ограничени­ями) числа m. Таким же условиям будут удовлетворять и числа, получае­мые в процессе шифрования. Это позволяет считать и те, и другие числа элементами кольца вычетов . Шифрующая функция при этом может рассматриваться как взаимнооднозначное отображение колец вычетов



а число  представляет собой сообщение  в зашифрованном виде.

Простейший шифр такого рода - шифр замены, соответству­ет отображению  при некотором фиксированном целом k. Подобный шифр использовал еще Юлий Цезарь. Конечно, не каждое отображение  подходит для целей надежного сокрытия инфор­мации.

В 1978 г. американцы Р. Ривест, А. Шамир и Л. Адлеман (R.L.Rivest. A.Shamir. L.Adleman) предложили пример функции , обла­дающей рядом замечательных достоинств. На её основе была построена реально используемая система шифрования, получившая название по пер­вым буквам имен авторов -система RSA. Эта функция такова, что

1) существует достаточно быстрый алгоритм вычисления значений ;

2) существует достаточно быстрый алгоритм вычисления значений об­ратной функции ;

3) функция  обладает некоторым «секретом», знание которого позво­ляет быстро вычислять значения ; в противном же случае вычисле­ние  становится трудно разрешимой в вычислительном отношении задачей, требующей для своего решения столь много времени, что по его  
прошествии зашифрованная информация перестает представлять инте­рес для лиц, использующих отображение  в качестве шифра.

Еще до выхода из печати статьи копия доклада в Массачусетском Технологическом институте, посвящённого системе RSA. была послана известному популяризатору математики М. Гарднеру, который в 1977 г. в журнале Scientific American опубликовал статью посвящённую этой системе шифрования. В русском переводе заглавие статьи Гарднера зву­чит так: Новый вид шифра, на расшифровку которого потребуются мил­лионы лет. Именно эта статья сыграла важнейшую роль в распростране­нии информации об RSA, привлекла к криптографии внимание широких кругов неспециалистов и фактически способствовала бурному прогрессу этой области, произошедшему в последовавшие 20 лет.

* 1. Система шифрования RSA

Пусть  и  натуральные числа. Функция  реализующая схему RSA, устроена следующим образом

. (1)

Для расшифровки сообщения  достаточно решить сравнение

. (2)

При некоторых условиях на  и  это сравнение имеет единственное решение .

Для того, чтобы описать эти условия и объяснить, как можно найти решение, нам потребуется одна теоретико-числовая функция, так назы­ваемая функция Эйлера. Эта функция натурального аргумента  обозна­чается  и равняется количеству целых чисел на отрезке от 1 до , взаимно простых с . Так  и  для любого простого числа  и натурального . Кроме того,  для лю­бых натуральных взаимно простых  и . Эти свойства позволяют легко вычислить значение , если известно разложение числа  на простые сомножители.

Если показатель степени  в сравнении (2) взаимно прост с , то сравнение (2) имеет единственное решение. Для того, чтобы найти его, определим целое число , удовлетворяющее условиям

. (3)

Такое число существует, поскольку , и притом единствен­но. Здесь и далее символом  будет обозначаться наибольший общий делитель чисел  и . Классическая теорема Эйлера, утверждает, что для каждого числа , взаимно простого с , выполняется сравнение  и, следовательно.

. (4)

Таким образом, в предположении , единственное решение срав­нения (2) может быть найдено в виде

. (5)

Если дополнительно предположить, что число  состоит из различных простых сомножителей, то сравнение (5) будет выполняться и без предпо­ложения . Действительно, обозначим  и . Тогда  делится на , а из (2) следует, что . Подобно (4), теперь легко находим . А кроме того, имеем . Получившиеся сравнения в силу  дают нам (5).

Функция (1), принятая в системе RSA, может быть вычислена доста­точно быстро. Обратная к  функция  вычисляется по тем же правилам, что и , лишь с заменой показателя степени  на . Таким образом, для функции (1) будут выполнены указанные выше свойства 1) и 2).

Для вычисления функции (1) достаточно знать лишь числа  и . Именно они составляют открытый ключ для шифрования. А вот для вы­числения обратной функции требуется знать число . оно и является «се­кретом», о котором речь идёт в пункте в). Казалось бы. ничего не стоит. зная число . разложить его на простые сомножители, вычислить затем с помощью известных правил значение  и, наконец, с помощью (3) определить нужное число . Все шаги этого вычисления могут быть реа­лизованы достаточно быстро, за исключением первого. Именно разложе­ние числа  на простые множители и составляет наиболее трудоемкую часть вычислений. В теории чисел несмотря на многолетнюю её историю и на очень интенсивные поиски в течение последних 20 лет, эффективный алгоритм разложения натуральных чисел на множители так и не найден. Конечно, можно, перебирая все простые числа до , и. деля на них , найти требуемое разложение. Но, учитывая, что количество простых в этом промежутке, асимптотически равно , на­ходим, что при , записываемом 100 десятичными цифрами, найдётся не менее  простых чисел, на которые придётся делить  при разложе­нии его на множители. Очень грубые прикидки показывают, что компью­теру, выполняющему миллион делений в секунду, для разложения числа  таким способом на простые сомножители потребуется не менее, чем  лет. Известны и более эффективные способы разложения целых чисел на множители, чем простой перебор простых делителей, но и они работают очень медленно.

Авторы схемы RSA предложили выбирать число  в виде произведе­ния двух простых множителей  и , примерно одинаковых по величине. Так как

, (6)

то единственное условие на выбор показателя степени  в отображении (1) есть

. (7)

Итак, лицо, заинтересованное в организации шифрованной переписки с помощью схемы RSA, выбирает два достаточно больших простых числа  и . Перемножая их, оно находит число . Затем выбирается число , удовлетворяющее условиям (7), вычисляется с помощью (6) число  и с помощью (3) - число . Числа  и  публикуются, число  остается секретным. Теперь любой может отправлять зашифрованные с помощью (1) сообщения организатору этой системы, а организатор легко сможет расшифровывать их с помощью (5).

Для иллюстрации своего метода Ривест, Шамир и Адлеман зашифро­вали таким способом некоторую английскую фразу. Сначала она стан­дартным образом (а=01, b=02, .... z=26, пробел=00) была записана в виде целого числа , а затем зашифрована с помощью отображения (1) при

m=11438162575788886766932577997614661201021829672124236256256184293570 6935245733897830597123563958705058989075147599290026879543541

и . Эти два числа были опубликованы, причем дополнительно сообщалось, что . где  и  - простые числа, записываемые со­ответственно 64 и 65 десятичными знаками. Первому, кто расшифрует соответствующее сообщение

, была обещана награда в 100$.

Эта история завершилась спустя 17 лет в 1994 г., когда D. Atkins, M. Graff, А. К. Lenstra и Р. С. Leyland сообщили о расшифровке фразы. Числа  и  оказались равными

,

.

Этот замечательный результат (разложение на мно­жители 129-значного десятичного числа) был достигнут благодаря ис­пользованию алгоритма разложения чисел на множители, называемого методом квадратичного решета. Выполнение вычислений потребовало колоссальных ресурсов. В работе, возглавленной четырьмя авторами проекта, и продолжавшейся после предварительной теоретической под­готовки примерно 220 дней, на добровольных началах участвовало около 600 человек и примерно 1600 компьютеров, объединённых сетью Inter­net. Наконец, отметим, что премия в 100$ была передана в Free Software Foundation.

* 1. Сложность теоретико-числовых алгоритмов

Сложность алгоритмов теории чисел обычно принято измерять коли­чеством арифметических операций (сложений, вычитаний, умножений и делений с остатком), необходимых для выполнения всех действий, пред­писанных алгоритмом. Впрочем, это определение не учитывает величины чисел, участвующих в вычислениях. Ясно, что перемножить два стозначных числа значительно сложнее, чем два однозначных, хотя при этом и в том, и в другом случае выполняется лишь одна арифметическая опе­рация. Поэтому иногда учитывают ещё и величину чисел, сводя дело к так называемым битовым операциям, т. е. оценивая количество необхо­димых операций с цифрами 0 и 1, в двоичной записи чисел.

Говоря о сложности алгоритмов, мы будем иметь в ви­ду количество арифметических операций. При построении эффективных алгоритмов и обсуждении верхних оценок сложности обычно хватает ин­туитивных понятий той области математики, которой принадлежит алго­ритм. Формализация же этих понятий требуется лишь тогда, когда речь идёт об отсутствии алгоритма или доказательстве нижних опенок слож­ности.

Приведем теперь примеры достаточно быстрых алгоритмов с опен­ками их сложности. Здесь и в дальнейшем мы не будем придерживаться формального описания алгоритмов, стараясь в первую очередь объяснить смысл выполняемых действий.

Следующий алгоритм вычисляет  за  арифмети­ческих операций. При этом, конечно, предполагается, что натуральные числа  и  не превосходят по величине .

* + 1. Алгоритм вычисления 

1. Представим  в двоичной системе счисления , где , цифры в двоичном представлении, равны 0 или 1, .
2. Положим  и затем для  вычислим

.

3)  есть искомый вычет .

Справедливость этого алгоритма вытекает из сравнения

,

легко доказываемого индукцией по .

Так как каждое вычисление на шаге 2 требует не более трёх умноже­ний по модулю  и этот шаг выполняется  раз, то сложность алгоритма может быть оценена величиной .

Второй алгоритм - это классический алгоритм Евклида вычисления наибольшего общего делителя целых чисел. Мы предполагаем заданными два натуральных числа  и  и вычисляем их наибольший общий дели­тель .

* + 1. Алгоритм Евклида

1. Вычислим  - остаток от деления числа  на , , .
2. Если , то  есть искомое число.
3. Если , то заменим пару чисел  парой  и перейдем к  
   шагу 1.

Теорема 1. При вычислении наибольшего общего делителя  с помощью алгоритма Евклида будет выполнено не более  операций де­ления с остатком, где  есть количество цифр в десятичной записи меньшего из чисел  и .

Доказательство. Положим  и определим  - последовательность делителей, появляющихся в процессе выполнения ша­га 1 алгоритма Евклида. Тогда

.

Пусть также , , , , - последователь­ность Фибоначчи. Индукцией по  от  до  легко доказывается неравенство . А так как , то имеем неравенства  и .

Немного подправив алгоритм Евклида, можно достаточно быстро ре­шать сравнения  при условии, что . Эта задача равносильна поиску целых решений уравнения .

* + 1. Алгоритм решения уравнения 

1) Определим матрицу .

2) Вычислим  - остаток от деления числа  на , , .

3) Если , то второй столбец матрицы  даёт вектор   
решений уравнения.

4) Если , то заменим матрицу  матрицей .

5) Заменим пару чисел  парой  и перейдем к шагу 1.

Если обозначить  через матрицу , возникающую в процессе работы алгоритма перед шагом 3 после  делений с остатком (шаг 2), то в обозначениях из доказательства теоремы 1 в этот момент выполняется векторное равенство . Поскольку числа  и  взаимно просты, имеем , и это доказы­вает, что алгоритм действительно даёт решение уравнения . Буквой  мы обозначили количество делений с остатком, которое в точ­ности такое же, как и в алгоритме Евклида.

Три приведённых выше алгоритма относятся к разряду так называе­мых полиномиальных алгоритмов. Это название носят алгоритмы, слож­ность которых оценивается сверху степенным образом в зависимости от длины записи входящих чисел. Если наибольшее из чисел, подаваемых на вход алгоритма, не превосходит , то сложность алгоритмов этого типа оценивается величиной , где  - некото­рая абсолютная постоянная. Во всех приведённых выше примерах .

Полиномиальные алгоритмы в теории чисел - большая редкость. Да и опенки сложности алгоритмов чаше всего опираются на какие-либо не доказанные, но правдоподобные гипотезы, обычно относящиеся к анали­тической теории чисел.

Для некоторых задач эффективные алгоритмы вообще не известны. Иногда в таких случаях все же можно предложить последовательность действий, которая, «если повезет», быстро приводит к требуемому ре­зультату. Существует класс так называемых вероятностных алгоритмов, которые дают правильный результат, но имеют вероятностную опен­ку времени работы. Обычно работа этих алгоритмов зависит от одного или нескольких параметров. В худшем случае они работают достаточно долго. Но удачный выбор параметра определяет быстрое завершение ра­боты. Такие алгоритмы, если множество «хороших» значений параметров велико, на практике работают достаточно эффективно, хотя и не имеют хороших опенок сложности.

Мы будем иногда использовать слова детерминированный алгоритм, чтобы отличать алгоритмы в обычном смысле от вероятностных алго­ритмов.

Как пример, рассмотрим вероятностный алгоритм, позволяющий эф­фективно находить решения полиномиальных сравнений по простому мо­дулю. Пусть  — простое число, которое предполагается большим, и  - многочлен, степень которого предполагается ограничен­ной. Задача состоит в отыскании решений сравнения

. (8)

Например, речь может идти о решении квадратичных сравнений, если степень многочлена  равна 2. Другими словами, мы должны отыскать в поле  все элементы, удовлетворяющие уравнению .

Согласно малой теореме Ферма, все элементы поля  являются од­нократными корнями многочлена . Поэтому, вычислив наибольший общий делитель , мы найдем многочлен , множест­во корней которого в поле  совпадает с множеством корней многочлена , причем все эти корни однократны. Если окажется, что многочлен  имеет нулевую степень, т. е. лежит в поле , это будет означать, что сравнение (8) не имеет решений.

Для вычисления многочлена  удобно сначала вычислить многочлен , пользуясь алгоритмом, подобным описанному выше алгоритму возведения в степень (напомним, что число  предполагается большим). А затем с помощью аналога алгоритма Евклида вычислить . Всё это выполняется за полиномиальное количество арифметических операций.

Таким образом, обсуждая далее задачу нахождения решений сравне­ния (8), мы можем предполагать, что в кольце многочленов  спра­ведливо равенство:



* + 1. Алгоритм нахождения делителей многочлена  в кольце 

1) Выберем каким-либо способом элемент .

2) Вычислим наибольший общий делитель .

3) Если многочлен  окажется собственным делителем , то многочлен  распадётся на два множителя и с каждым из них незави­симо нужно будет проделать все операции, предписываемые настоящим алгоритмом для многочлена .

4) Если окажется, что  или , следует перейти к шагу 1 и. выбрав новое значение , продолжить выполнение алгоритма.

Количество операций на шаге 2 оценивается величиной , ес­ли вычисления проводить так, как это указывалось выше при нахожде­нии . Выясним теперь, сколь долго придётся выбирать числа , пока на шаге 2 не будет найден собственный делитель .

Количество решений уравнения  в поле  не превосходит . Это означает, что подмножество  элементов , удовлетворяющих условиям

,

состоит не менее, чем из  элементов. Учитывая теперь, что каждый ненулевой элемент  удовлетворяет одному из равенств , либо , заключаем, что для  одно из чисел  будет корнем многочлена , а другое - нет. Для таких элементов  многочлен , определённый на шаге 2 алгоритма, будет собственным делителем многочлена .

Итак, существует не менее  «удачных» выборов элемента , при которых на шаге 2 алгоритма многочлен  распадётся на два соб­ственных множителя. Следовательно, при «случайном» выборе элемента , вероятность того, что многочлен не разложится на множители после  повторений шагов алгоритма 1-4. не превосходит . Вероят­ность с ростом  убывает очень быстро. И действительно, на практике этот алгоритм работает достаточно эффективно.

Заметим, что при опенке вероятности мы использовали только два корня многочлена . При  эта вероятность, конечно, еще меньше. Более тонкий анализ с использованием опенок А. Вейля для сумм харак­теров показывает, что вероятность для многочлена  не распасться на множители при однократном проходе шагов алгоритма 1-4. не пре­восходит . Здесь постоянная в  зависит от .

Если в сравнении (8) заменить простой модуль  составным моду­лем , то задача нахождения решений соответствующего сравнения ста­новится намного более сложной. Известные алгоритмы её решения осно­ваны на сведении сравнения к совокупности сравнений (8) по простым модулям — делителям , и. следовательно, они требуют разложения чи­сла то на простые сомножители, что, как уже указывалось, является до­статочно трудоемкой задачей.

* 1. Качественная теория алгоритма RSA

Существует довольно эффективный способ убедиться, что заданное число является составным, не разлагая это число на множители. Согласно малой теореме Ферма, если число  простое, то для любого целого , не делящегося на , выполняется сравнение

. (9)

Если же при каком-то  это сравнение нарушается, можно утверждать, что  - составное. Проверка (9) не требует больших вычислений, это следует из алгоритма 1. Вопрос только в том, как найти для составного  целое число , не удовлетворяющее (9). Можно, например, пытаться найти необходимое число , испытывая все целые числа подряд, начиная с 2. Или попробовать выбирать эти числа случайным образом на отрезке .

К сожалению, такой подход не всегда даёт то, что хотелось бы. Име­ются составные числа , обладающие свойством (9) для любого целого  с условием . Такие числа называются числами Кармайкла. Рассмотрим, например, число . Так как 560 делится на каждое из чисел 2, 10, 16, то с помощью малой теоремы Ферма легко проверить, что 561 есть число Кармайкла. Можно доказать, что любое из чисел Кармайкла имеет вид , где все простые  различны, причем  делится на каждую разность . Лишь недавно, была решена проблема о бесконечности множества таких чисел.

В 1976 г. Миллер предложил заменить проверку (9) проверкой несколь­ко иного условия. Если  - простое число, , где  нечётно, то согласно ма­лой теореме Ферма для каждого  с условием  хотя бы одна из скобок в произведении



делится на . Обращение этого свойства можно использовать, чтобы от­личать составные числа от простых.

Пусть  - нечётное составное число, , где  нечётно. Назовем целое число , , «хорошим» для , если нарушается одно из двух условий:

1)  не делится на ;

2)  или существует целое , , такое, что

.

Из сказанного ранее следует, что для простого числа  не существует хороших чисел . Если же  составное число, то, как доказал Рабин, их существует не менее .

Теперь можно построить вероятностный алгоритм, отличающий со­ставные числа от простых.

* + 1. Алгоритм доказывающий не простоту числа

1. Выберем случайным образом число , , и проверим для  
   этого числа указанные выше свойства 1) и 2) п.2.
2. Если хотя бы одно из них нарушается, то число  составное.
3. Если выполнены оба условия 1) и 2) п.2, возвращаемся к шагу 1.

Из сказанного выше следует, что составное число не будет определено как составное после однократного выполнения шагов 1-3 с вероятностью не большей . А вероятность не определить его после  повторений не превосходит . т. е. убывает очень быстро.

Миллер предложил детерминированный алгоритм определения состав­ных чисел, имеющий сложность , однако справедливость его ре­зультата зависит от недоказанной в настоящее время так называемой расширенной гипотезы Римана. Согласно этому алгоритму достаточно проверить условия 1) и 2) п.2 для всех целых чисел , . Если при каком-нибудь  из указанного промежутка нарушается одно из условий а) или б), число  составное. В противном случае оно будет простым или степенью простого числа. Последняя возможность, конечно, легко проверяется.

Напомним некоторые понятия, необходимые для формулиров­ки расширенной гипотезы Римана. Они понадобятся нам и в дальнейшем. Пусть  - целое число. Функция  называется характе­ром Дирихле по модулю , или просто характером, если эта функция периодична с периодом , отлична от нуля только на числах, взаимно простых с , и мультипликативна, т. е. для любых целых  выполня­ется равенство . Для каждого  существует ровно  характеров Дирихле. Они образуют группу по умножению. Единичным элементом этой группы является так называемый главный характер , равный 1 на всех числах, взаимно простых с , и 0 на остальных це­лых числах. Порядком характера называется его порядок как элемента мультипликативной группы характеров.

С каждым характером может быть связана так называемая  - функция Дирихле - функция комплексного переменного , определённая рядом. Сумма этого ряда аналитична в области  и может быть аналитически продолжена на всю комплексную плос­кость. Следующее соотношение  связывает L - функцию, отвечающую главному характеру, с дзета-функцией Римана . Расширенная гипотеза Римана утверждает, что комплексные нули всех L -функций Дирихле, расположенные в полосе , лежат на прямой . В настоящее время не доказана даже простей­шая форма этой гипотезы - классическая гипотеза Римана, утвержда­ющая такой же факт о нулях дзета-функции.

В 1952 г. Анкени с помощью расширенной гипотезы Римана доказал, что для каждого простого числа  существует квадратичный невычет , удовлетворяющий неравенствам . Константа 70 была со­считана позднее. Именно это утверждение и лежит в основе алгоритма Миллера. В 1957 г. Берджесс доказал существование такого невычета без использования расширенной гипотезы Римана, но с худшей оценкой , справедливой при любом положительном  и , большем некоторой границы, зависящей от.

Алгоритм Миллера принципиально отличается от алгоритма 2.1., так как полученное с его помощью утверждение о том, что число  - со­ставное, опирается на недоказанную расширенную гипотезу Римана и по­тому может быть неверным. В то время как вероятностный алгоритм 2.1. даёт совершенно правильный ответ для составных чисел. Несмотря на отсутствие оценок сложности, на практике он работает вполне удовле­творительно.

* + 1. Нахождение больших простых чисел

Конечно же, большие простые числа можно строить сравнительно быстро. При этом можно обеспечить их случайное распределение в заданном диапазоне величин. В противном случае теряла бы всякий практический смысл система шифрования RSA. Наиболее эффективным средством построения простых чисел является несколько модифицированная малая теорема Ферма.

Теорема 2*.* Пусть  - нечётные натуральные числа, , причем для каждого простого делителя  числа  существует целое число  такое, что

. (10)

Тогда каждый простой делитель  числа  удовлетворяет сравнению

.

Доказательство. Пусть  - простой делитель числа , a  - не­который делитель . Из условий (10) следует, что в поле вычетов  спра­ведливы соотношения

. (11)

Обозначим буквой  порядок элемента  в мультипликативной группе поля . Первые два из соотношений (11) означают, что  входит в раз­ложение на простые множители числа  в степени такой же, как и в раз­ложение , а последнее - что  делится на . Таким образом, каждый простой делитель числа  входит в разложение  в степени не меньшей, чем в , так что  делится на . Кроме того,  четно. Теорема 2 доказана.

Следствие. Если выполнены условия теоремы 2 и , то  - простое число.

Действительно, пусть  равняется произведению не менее двух про­стых чисел. Каждое из них, согласно утверждению теоремы 2, не меньше, чем . Но тогда . Противоречие и доказывает следствие.

Покажем теперь, как с помощью последнего утверждения, имея боль­шое простое число , можно построить существенно большее простое число . Выберем для этого случайным образом чётное число  на про­межутке  и положим . Затем проверим число  на отсутствие малых простых делителей, разделив его на малые простые числа; испытаем  некоторое количество раз с помощью алгоритма 5. Если при этом выяснится, что  - составное число, следует выбрать новое значение  и опять повторить вычисления. Так следует делать до тех пор, пока не будет найдено число , выдержавшее испытания алго­ритмом 5 достаточно много раз. В этом случае появляется надежда на то, что  - простое число, и следует попытаться доказать простоту с помощью тестов теоремы 2.

Для этого можно случайным образом выбирать число , и проверять для него выполнимость соотношений

. (12)

Если при выбранном  эти соотношения выполняются, то, согласно след­ствию из теоремы 2, можно утверждать, что число  простое. Если же эти условия нарушаются, нужно выбрать другое значение  и повторять эти операции до тех пор, пока такое число не будет обнаружено.

Предположим, что построенное число  действительно является про­стым. Зададимся вопросом, сколь долго придётся перебирать числа , по­ка не будет найдено такое, для которого будут выполнены условия (12). Заметим, что для простого числа  первое условие (12), согласно малой теореме Ферма, будет выполняться всегда. Те же числа , для которых на­рушается второе условие (12), удовлетворяют сравнению . Как известно, уравнение  в поле вычетов  имеет не более  решений. Одно из них . Поэтому на промежутке  имеется не более  чисел, для которых не выполняются условия (12). Это означа­ет, что, выбирая случайным образом числа  на промежутке , при простом  можно с вероятностью большей, чем , найти чи­сло , для которого будут выполнены условия теоремы 2, и тем доказать, что  действительно является простым числом.

Заметим, что построенное таким способом простое число  будет удовлетворять неравенству , т. е. будет записываться вдвое боль­шим количеством цифр, чем исходное простое число . Заменив теперь число  на найденное простое число  и повторив с этим новым  все указанные выше действия, можно построить еще большее простое число. Начав с какого-нибудь простого числа, скажем, записанного 10 десятич­ными цифрами (простоту его можно проверить, например, делением на маленькие табличные простые числа), и повторив указанную процедуру достаточное число раз. можно построить простые числа нужной величи­ны.

Обсудим теперь некоторые теоретические вопросы, возникающие в связи с нахождением числа , удовлетворяющего неравенствам , и такого, что  - простое число. Прежде всего, со­гласно теореме Дирихле, доказанной еще в 1839 г., прогрессия ,  содержит бесконечное количество простых чисел. Нас интересуют простые числа, лежащие недалеко от начала прогрессии. Опенка наименьшего простого числа в арифметической прогрессии была полу­чена в 1944 г. Ю. В. Линником. Соответствующая теорема утверждает, что наименьшее простое число в арифметической прогрессии  не превосходит , где  - некоторая достаточно большая абсолютная постоянная.

Таким образом, в настоящее время никаких теоретических гарантий для существования простого числа  не сущес­твует. Тем не менее опыт вычислений на ЭВМ показывает, что простые числа в арифметической прогрессии встречаются достаточно близко к её началу. Упомянем в этой связи гипотезу о существовании бесконечного количества простых чисел  с условием, что число  также простое, т. е. простым является уже первый член прогрессии.

Очень важен в связи с описываемым методом построения простых чисел также вопрос о расстоянии между соседними простыми числами в арифметической прогрессии. Ведь убедившись, что при некотором  число  составное, можно следующее значение  взять равным  и действовать так далее, пока не будет найдено простое число . И если расстояние между соседними простыми числами в прогрессии ве­лико, нет надежды быстро построить нужное число . Перебор чисел  до того момента, как мы наткнемся на простое число  окажется слишком долгим. В более простом вопросе о расстоянии между соседни­ми простыми числами  и  в натуральном ряде доказано лишь, что , что, конечно, не очень хорошо для наших целей. Вместе с тем существует так называемая гипотеза Крамера (1936 г.), что , дающая вполне приемлемую опенку. Примерно такой же результат следует и из расширенной гипотезы Римана. Вычисления на ЭВМ показывают, что простые числа в арифметических прогрессиях расположены достаточно плотно.

В качестве итога обсуждения в этом пункте подчеркнём следующее: если принять на веру, что наименьшее простое число, а также расстояние между соседними простыми числами в прогрессии  при  оцениваются величиной , то описанная схема построения больших простых чисел имеет полиномиальную опенку сложности. Кро­ме того, несмотря на отсутствие теоретических опенок времени работы алгоритмов, отыскивающих простые числа в арифметических прогрес­сиях со сравнительно большой разностью, на практике эти алгоритмы работают вполне удовлетворительно. На обычном персональном компью­тере без особых затрат времени строятся таким способом простые числа порядка .

Конечно, способ конструирования простых чисел для использования в схеме RSA должен быть массовым, а сами простые числа должны быть в каком-то смысле хорошо распределёнными. Это вносит ряд дополнитель­ных осложнений в работу алгоритмов.

Наконец, отметим, что существуют методы построения больших про­стых чисел, использующие не только простые делители , но и делите­ли чисел . В основе их лежит использование после­довательностей целых чисел, удовлетворяющих линейным рекуррентным уравнениям различных порядков. Отметим, что последовательность , члены которой присутствуют в формулировке малой теоремы Ферма, со­ставляет решение рекуррентного уравнения первого порядка .

* + 1. Проверка большого числа на простоту

Есть некоторое отличие в постановках задач предыдущего и настоя­щего пунктов. Когда мы строим простое число , мы обладаем некото­рой дополнительной информацией о нем, возникающей в процессе постро­ения. Например, такой информацией является знание простых делителей числа . Эта информация иногда облегчает доказательство просто­ты .

В этом пункте мы предполагаем лишь, что нам задано некоторое чи­сло , например, выбранное случайным образом на каком-то промежут­ке, и требуется установить его простоту, или доказать, что оно является составным. Эту задачу за полиномиальное количество операций решает указанный в п. 3 алгоритм Миллера. Однако, справедливость полученного с его помощью утверждения зависит от недоказанной расширенной гипо­тезы Римана. Если число  выдержало испытания алгоритмом 5 для 100 различных значений параметра , то, по-видимому, можно утверждать, что оно является простым с вероятностью большей, чем . Эта вероятность очень близка к единице, однако всё же оставляет некоторую тень сомнения на простоте числа . В дальнейшем в этом пункте мы будем считать, что заданное число  является простым, а нам требуется лишь доказать это.

В настоящее время известны детерминированные алгоритмы различ­ной сложности для доказательства простоты чисел. Мы остановимся по­дробнее на одном из них, предложенном в 1983 г. в совместной работе Адлемана. Померанца и Рамели. Для доказательства простоты или непростоты числа  этот алгоритм требует  арифметиче­ских операций. Здесь  - некоторая положительная абсолютная посто­янная. Функция  хоть и медленно, но всё же возрастает с ростом , поэтому алгоритм не является полиномиальным. Но всё же его прак­тические реализации позволяют достаточно быстро тестировать числа на простоту. Существенные усовершенствования и упрощения в перво­начальный вариант алгоритма были внесены в работах X. Ленстры и А. Коена. Мы будем называть описываемый ниже алгоритм алго­ритмом Адлемана - Ленстры.

В основе алгоритма лежит использование сравнений типа малой те­оремы Ферма, но в кольцах целых чисел круговых полей, т. е. полей. порождённых над полем  числами  - корнями из 1. Пусть  - простое нечётное число и  — первообразный корень по модулю , т. е. образующий элемент мультипликативной группы поля , которая пиклична. Для каждого целого числа , не делящегося на , можно опре­делить его индекс, , называемый также *дискретным логарифмом*, с помощью сравнения . Рассмотрим далее два простых числа ,  с условием, что  делится на , но не делится на .

Следующая функция, определённая на множестве целых чисел.



является характером по модулю  и порядок этого характера равен .

Сумма:



называется суммой Гаусса. Формулируемая ниже теорема 3 представляет собой аналог малой теоремы Ферма, используемый в алгоритме Адлемана - Ленстры.

Теорема3. Пусть  - нечетное простое число, . Тогда в кольце  выполняется сравнение

.

Если при каких-либо числах  сравнение из теоремы 3 нарушается. можно утверждать, что  составное число. В противном случае, если сравнение выполняется, оно даёт некоторую информацию о возможных простых делителях числа . Собрав такую информацию для различных , в конце концов удаётся установить, что  имеет лишь один простой делитель и является простым.

В случае  легко проверить, что сравнение из теоремы 3 равно­сильно хорошо известному в элементарной теории чисел сравнению

, (13)

где  - так называемый символ Якоби. Хорошо известно также, что последнее сравнение выполняется не только для простых , но и для любых целых , взаимно простых с . Заметим также, что для вычисления символа Якоби существует быстрый алгоритм, основанный на законе вза­имности Гаусса и. в некотором смысле, подобный алгоритму Евклида вы­числения наибольшего общего делителя. Следующий пример показывает. каким образом выполнимость нескольких сравнений типа (13) даёт неко­торую информацию о возможных простых делителях числа .

Пример (X. Ленстра). Пусть  — натуральное число, . для которого выполнены сравнения

, (14)

а кроме того с некоторым целым числом  имеем

. (15)

Как уже указывалось, при простом  сравнения (14) выполняются для любого , взаимно простого с , а сравнение (15) означает, что  есть первообразный корень по модулю . Количество первообразных корней равно , т. е. достаточно велико. Таким образом, число  с усло­вием (15) при простом  может быть найдено достаточно быстро с помощью случайного выбора и последующей проверки (15).

Докажем, что из выполнимости (14-15) следует, что каждый делитель  числа  удовлетворяет одному из сравнений

 или . (16)

Не уменьшая общности, можно считать, что  - простое число. Введем теперь обозначения , где  и  - нечётные числа. Из (15) и сравнения  следует, что . Далее, согласно (14). выполняются следующие сравнения

,

означающие (в силу того, что символ Якоби может равняться лишь -1 или +1), что .

При  это равенство означает, что  при , и, следовательно, . Если же , то имеем  и . Этим (16) доказано.

Информация такого рода получается и в случае произвольных про­стых чисел  и  с указанными выше свойствами.

Опишем схему алгоритма Адлемана - Ленстры для про­верки простоты :

1. выбираются различные простые числа  и различные про­стые нечётные  такие, что
   1. для каждого  все простые делители числа  содержатся  
      среди  и  не делятся на квадрат простого числа;

1.2) .

1. для каждой пары выбранных чисел ,  проводятся тесты, подобные сравнению из теоремы 3. Если  не удовлетворяет какому-либо из  
   этих тестов - оно составное. В противном случае
2. определяется не очень большое множество чисел, с которыми толь­ко и могут быть сравнимы простые делители . А именно, каждый простой делитель  числа  должен удовлетворять сравнению вида

, .

1. проверяется, содержит ли найденное множество делители . Ес­ли при этом делители не обнаружены, утверждается, что  - простое  
   число.

Если число  составное, оно обязательно имеет простой делитель , меньший , который сам содержится среди возможных остатков. Именно на этом свойстве основано применение пункта 4) алгоритма.

Сумма Якоби:



определяется для двух характеров  модулю . Если характеры имеют порядок , то соответствующая сумма Якоби принадлежит кольцу . Поскольку числа , участвующие в алгоритме, сравнительно невели­ки, то вычисления с суммами Якоби производятся в полях существенно меньшей степени, чем вычисления с суммами Гаусса. Это главная при­чина, по которой суммы Якоби предпочтительнее для вычислений. При  выполняется классическое соотношение



связывающее суммы Гаусса с суммами Якоби и позволяющее переписать сравнение теоремы 3 в терминах сумм Якоби. Так. при  и  соответствующее сравнение, справедливое для простых , отлич­ных от 2,3,7, принимает вид

,

где  и  - некоторый корень кубический из 1.

В 1984 г. было внесено существенное усовершенствование в алгоритм, позволившее освободиться от требования неделимости чисел  на квадраты простых чисел. В результате, например, выбрав число  и взяв  равным произведению простых чисел  с условием, что  делится на , получим , что позволяет доказывать простоту чисел , записываемых сотней десятичных знаков. При этом вычисления будут проводиться в полях, порождённых корнями из 1 степеней 16, 9, 5 и 7.

Персональный компьютер с процессором Pentium-150. пользуясь реализацией этого алгоритма на языке UBASIC, доказал простоту записываемого 65 десятичными знаками, большего из простых чисел в при­мере Ривеста, Шамира и Адлемана за 8 секунд. Сравнение этих 8 секунд и 17 лет, потребовавшихся для разложения на множители предложенного в примере числа, конечно, впечатляет.

Отметим, что опенка сложности этого алгоритма представляет со­бой трудную задачу аналитической теории чисел. Как уже указывалось, количество операций оценивается величиной . Однако соот­ветствующие числа  и , возникающие в процессе доказательства, не могут быть явно указаны в зависимости от . Доказано лишь существо­вание чисел  и , для которых достигается оценка. Впрочем, есть веро­ятностный вариант алгоритма, доказывающий простоту простого числа  с вероятностью большей  за  арифметических операций. А в предположении расширенной гипотезы Римана эта опенка сложности может быть получена при эффективно указанных  и.

1. Проектирование

Разрабатываемую информационную систему можно отнести к клиент-серверной архитектуре. На стороне клиента используется приложение, написанное под операционную систему GNU/Linux Debian Wheezy на языке Python.

* 1. Проектирование функций
* rsa.**encrypt**(message, pub\_key)

Шифрует данное сообщение используя PKCS#1 v1.5

**Parameters:**

1. **message** – сообщение для шифрования. Должен быть строкой не более k-11, где k – число байтов, для кодирования n компонент открытого ключа.
2. **pub\_key** –открытый ключ rsa.PublicKey для шифрования.

**Выдает OverflowError:**

Сообщение слишком велико, чтобы уместить блок кодирования.

* rsa.**decrypt**(*crypto*, *priv\_key*)

Расшифровывает данное сообщение с помощью PKCS # 1 v1.5

Расшифровка считается "не удалась", когда в результате открытый текст не начинается с байтов 00 02, или когда 00 байт между внутренним отступом и сообщением, не могут быть найдено.

**Parameters:**

1. **crypto** – крипто текст как результат выполнения rsa.PublicKey()
2. **priv\_key** – закрытый ключ rsa.PrivateKey для дешифрования.

**Выдает DecryptionError:**

Когда не поддается расшифровке. Никаких подробностей не приводится, почему не получилось расшифровать, т.к. это будет наводящей информацией о закрытом ключе.

* rsa.**sign**(message, priv\_key, hash)

Подписывает сообщение с секретным ключом.

Хэширует сообщение, затем подписывает хэш с данным ключом. Это известно как «Отдельная сигнатура», потому что само сообщение не изменяется.

**Parameters:**

1. **message** – подписываемое сообщение
2. **priv\_key** – закрытый ключ rsa.PrivateKey для подписывания сообщения.
3. **hash** – способ хеширования сообщения. Возможно использовать ‘MD5’, ‘SHA-1’, ‘SHA-256’, ‘SHA-384’ или ‘SHA-512’.

**Возвращает:**

Блок подписанного сообщения.

**Выдает OverflowError:**

Если закрытый ключ слишком мал, чтобы содержать выбранный хэш.

* rsa.**verify**(message, signature, pub\_key)

Проверяет подпись сообщения на совпадение.

Хэш метод определяется автоматически с момента подписания.

**Parameters:**

1. **message** – подписанное сообщение. Может быть 8-битной строкой или файл-подобный объект.
2. **signature**  – блок подписи, созданный с rsa.sign()
3. **pub\_key** – публичный ключ rsa.PublicKey подписавшего сообщение.

**Выдает VerificationError:**

Когда подпись не совпадает с сообщением.

* rsa.**newkeys**(keysize)

Создает открытый и закрытый ключи и возвращает их в виде (открытый, закрытый).

Открытый ключ нужен как «ключ шифрования» и является объектом rsa.PublicKey. Закрытый ключ нужен как «ключ дешифрования» и является объектом rsa.PrivateKey.

**Parameters:**

1. **nbits** – количество бит необходимых для хранения n = p \* q
2. **accurate**  – когда истина, n будет иметь запрошенное количество битов.
3. **poolsize** – количество используемых процессов для генерации простых чисел. Если установлено число большее 1, параллельный алгоритм будет использоваться. Это требует Python 2.6 и новее.

**Возвращает:**

Запись (rsa.PublicKey, rsa.PrivateKey)

* 1. Проектирование классов
* class rsa.**PublicKey**(n, e)

Представляет открытый ключ RSA

Этот ключ используется как «ключ шифрования». Он содержит ‘N’ и ‘E’ значения.

Поддерживает атрибуты, а также словарь доступа. Атрибуты доступа все же быстрее.

**load\_pkcs1**(keyfile, format='PEM')

Загружает ключ PKCS#1 в DER или PEM форматах.

**Parameters:**

1. **keyfile** – содержимое DER или PEM закодированного файла, содержащего открытый ключ.
2. **format**  – загружаемый формат файла DER или PEM.

**Возвращает:**

Объект публичного ключа PublicKey.

classmethod **load\_pkcs1\_openssl\_pem**(keyfile)

Загружает файл открытого ключа PKCS#1 в DER – кодировке из OpenSSL

**Parameters:**

Параметр **keyfile** – содержимое файла в DER кодировке, что содержит открытый ключ OpenSSL

**Возвращает:**

Объект публичного ключа PublicKey.

classmethod **load\_pkcs1\_openssl\_der**(n, e, d, p, q, exp1=None, exp2=None, coef=None)

Загружает файл открытого ключа PKCS#1 в PEM – кодировке из OpenSSL

Этот файл признается в том случае, если начинается с BEGIN PUBLIC KEY, а не с BEGIN RSA PUBLIC KEY.

Содержимое файла перед “—–BEGIN PUBLIC KEY—–” и после “—–END PUBLIC KEY—–” линий, игнорируются.

**Parameters:**

Параметр **keyfile** – содержимое файла в PEM кодировке, что содержит открытый ключ OpenSSL

**Возвращает:**

Объект публичного ключа PublicKey.

**save\_pkcs1**(format='PEM')

Сохраняет публичный ключ PKCS#1 в DER или PEM формате

**Parameters:**

**format** – формат сохранения (DER или PEM формат).

**Возвращает:**

Публичный ключ шифрования в DER или PEM формате.

* class rsa.PrivateKey(n, e, d, p, q, exp1=None, exp2=None, coef=None)

Представляет секретный ключ RSA.

Этот ключ используется как «ключ дешифрования». Он содержит ‘n’, ‘e’, ‘d’, ‘p’, ‘q’ и другие значения.

Поддерживает атрибуты, а также словарь доступа. Атрибут доступа все же быстрее.

**load\_pkcs1**(keyfile, format='PEM')

Загружает ключ PKCS#1 в DER или PEM формате

**Parameters:**

1. **keyfile** – содержимое в DER или PEM формате закодированного файла, содержащего открытый ключ.
2. **format -** формат файла для загрузки (DER или PEM формат).

**Возвращает:**

Объект публичного ключа.

**save\_pkcs1**(format='PEM')

Сохраняет публичный ключ PKCS#1 в DER или PEM формате

**Parameters:**

1. **format -** формат файла для сохранения (DER или PEM формат).

**Возвращает:**

Публичный ключ в DER или PEM формате.

* 1. Проектирование классов исключений
* *class* rsa.pkcs1.**CryptoError**(Exception)

Базовый класс для всех исключений в этом модуле.

* *class* rsa.pkcs1.**DecryptionError**(CryptoError)

Возникает, когда расшифровать не удается.

* *class* rsa.pkcs1.**VerificationError**(CryptoError)

Возникает, когда верифицировать не удается.

* 1. Модуль rsa.bigfile

Модуль rsa.bigfile содержит функции для шифрования и дешифрования файлов, размер которых превышает размер ключа RSA.

rsa.bigfile.**encrypt\_bigfile**(infile, outfile, pub\_key)

Шифрует файл, записывая его в выходной файл «outfile» в формате VARBLOCK

**Parameters:**

1. **infile –** файл-подобный объект, чтобы читать открытым ткстом.
2. **outfile** – файл-подобный объект, чтобы зашифровать в формате VARBLOCK.
3. **pub\_key** – публичный ключ шифрования rsa.PublicKey

rsa.bigfile.**decrypt\_bigfile**(infile, outfile, priv\_key)

Дешифрует зашифрованный VARBLOCK файл, записывая в входной файл «outfile».

**Parameters:**

1. **infile –** файл-подобный объект, чтобы прочитать шифрованный VARBLOCK файл.
2. **outfile** – файл-подобный объект, чтобы читать открытым ткстом.
3. **priv\_key** – закрытый ключ дешифрования rsa.PrivateKey
   1. Формат VARBLOCK файла

Формат файла VARBLOCK позволяет шифровать файлы, размер которых превышает размер ключа RSA. Формат должен быть следующим; || обозначает конкатенацию байт строк:

VARBLOCK := VERSION || BLOCK || BLOCK || ...

VERSION := 1

BLOCK := LENGTH || DATA

LENGTH := varint-кодировка длины в байтах

DATA := данные для хранения в блоке

Varint-формат был взят из Protobuf Google, и позволяет эффективно кодировать сколь угодно длинное целое.

* 1. Модуль rsa.core

В основе метода шифрования RSA лежат эти две функции. Они обе работают только на (сколько угодно длинных) целых числах. Эти функции являются ядром всей библиотеки.

rsa.core.**encrypt\_int**(message, ekey, n)

Шифрует сообщения с использованием ключа шифрования «Ekey» по модулю n.

rsa.core.**decrypt\_int**(cyphertext, dkey, n)

Дешифрует зашифрованный текст с помощью дешифрующего ключа «dkey» по модулю n.

1. Тестирование и документирование
2. Экономическое обоснование разработки

Заключение

Список литературы