УДК 519.688

Сергиенко Елена Николаевна
к.ф.-м.н., доцент
Чурилов Антон Сергеевич
Давыденко Денис Олегович
Панарин Сергей Александрович
Пригорнев Иван Андреевич
Смакаев Анатолий Витальевич
студенты
Белгородский Государственный Технологический Университет им. В. Г. Шухова
Белгородская область, Россия

ВОПРОС РЕАЛИЗАЦИИ ТЕОРЕТИКО-ЧИСЛОВЫХ МЕТОДОВ В РАЗЛИЧНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ.

Во многих научных расчетах оперируют в основном числами, разрядность которых превышает размер машинного слова данной вычислительной машины. Производить действия с такими числами стандартными алгоритмами очень затруднительно, а порой и совсем невозможно. Чтобы сократить время вычислений и были придумана быстрая арифметика и быстрые алгоритмы для осуществления операций длинными числами.

Многие языки программирования имеют встроенную поддержку длинной арифметики, что в разы сокращает время написания программ.

Быстрые алгоритмы — область вычислительной математики, которая изучает алгоритмы вычисления заданной функции с заданной точностью с использованием как можно меньшего числа битовых операций.

Сложность умножения $M(n)$ определяется как количество битовых операций, достаточное для вычисления произведения двух n – значных чисел посредством данного алгоритма.

Алгоритм Карацубы-Оффмана

Рассмотрим идею быстрого алгоритма умножения, названного в честь советского и российского математика *Анатолия Алексеевича Карацубы*, известного как создателя первого быстрого метода умножения больших чисел. Сложность умножения данного алгоритма $O(n^{\log_2 n})$.

Основная идея алгоритма Карацубы заключается в формулах, которые позволяют вычислять произведения двух больших чисел X и Y, используя три умножения меньших чисел.

Представим, что есть два числа X и Y длиной n в какой-то системе счисления B:

$$X = x_{n-1}x_{n-2}\dots x_0$$
$$Y = y_{n-1}y_{n-2}\dots y_0$$

Здесь x_i, y_i — значение битов в соответствующем разряде числа. Каждое из этих чисел можно представить в виде суммы их двух частей, половинок длиной $m = \frac{n}{2}$. Если n нечетное, то одна часть короче другой на один разряд:

$$X_0 = x_{m-1}x_{m-2}\dots x_0\,,\quad X_1 = x_{n-1}x_{n-2}\dots x_m\,,\quad X = X_1 * B^m + X_0$$
$$Y_0 = y_{m-1}y_{m-2}\dots y_0\,,\quad Y_1 = y_{n-1}y_{n-2}\dots y_m\,,\quad Y = Y_1 * B^m + Y_0$$

$$(X_1 + X_0)(Y_1 + Y_0) = X_1 * Y_1 + \boldsymbol{X_1} * \boldsymbol{Y_0} + \boldsymbol{X_0} * \boldsymbol{Y_1} + X_0 * Y_0$$
$$\boldsymbol{X_1} * \boldsymbol{Y_0} + \boldsymbol{X_0} * \boldsymbol{Y_1} = (X_1 + X_0)(Y_1 + Y_1) - X_1 * Y_1 - X_0 * Y_1$$

Действительно, пусть

$$Z_2 = X_1 * Y_1,\qquad Z_0 = X_0 * Y_0,$$
$$Z_1 = (X_1 + X_0)(Y_1 + Y_0) - Z_2 - Z_0$$

Таким образом,

$$XY = X_1 * Y_1 * B^{2m} + \big((X_1 + X_0)(Y_1 + Y_0) - X_1 * Y_1 - X_0 * Y_0\big) * B^m + X_0 * Y_0$$

Описание алгоритма

Вход: целые числа A и B

Выход: целое число $C = A * B$

Шаг 1. Если числа A и B представляются с помощью чисел длинной меньше N цифр(обменная точка алгоритма), то ищется A и B произведение классическим способом.

Шаг 2. Разбить каждое из сомножителей на две части, старшую и младшую :

$$m = \frac{n}{2} \quad X = X_1 * B^m + X_0 \qquad Y = Y_1 * B^m + Y_0$$

Шаг 3. Вычислить $X_1 * Y_1$, $X_1 * Y_0 + X_0 * Y_1$, $X_0 * Y_0$, рекурсивно обращаясь к данному алгоритму.

Шаг 4. Получить результат $C = A * B$ комбинируя для частичных результатов операции сложения и сдвига.

Конец алгоритма.

Анализ алгоритма

Для чисел, разрядность которых не превышает длину машинного слова данной машины, алгоритм Карацубы-Оффмана, согласно опытам, проигрывает во времени умножению "в столбик".

Но после этого порога наблюдается обратная тенденция. А с учетом того, что результаты разложения чисел, мы также можем перемножать методом Карацубы (до тех пор, пока их разрядность не опустится ниже разрядности архитектуры), то при переходе разрядности чисел в следующую степень двойки мы получаем скачок в разности скорости работы алгоритмов.

При реализации данного алгоритма на языках Java и Python была замечена их особенность при работе с большими числами. Она заключается в том, что по умолчанию в языке Java при работе с классом BigInteger используется алгоритм умножения в «столбик», а язык Python при работе с числами, разрядность которых превышает длину машинного слова, использует более быстрый алгоритм.

Существует понятие *обменной точки*, т.е. того значения длины числа (количества цифр в числе), при котором быстрый алгоритм начинает выигрывать у классического алгоритма. Эти величины обычно значительны и часто зависят от реализации алгоритма на компьютере.

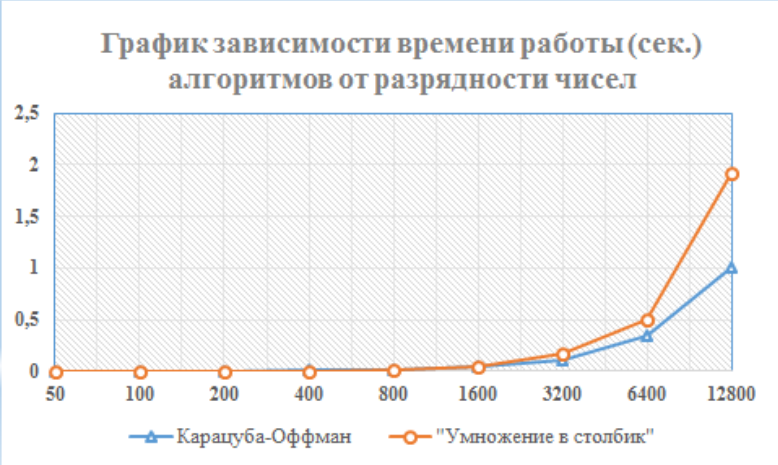

Рис 1. График зависимости времени работы (сек.) алгоритмов от количества десятичных разрядов.

На рис. 1 показан график зависимости времени работы алгоритмов Карацубы и алгоритма умножения «столбиком» от числа десятичных разрядов чисел, где обменная точка примерно равна 2500.

Алгоритм модульного умножения Монтгомери

Модульное умножение – наиболее часто встречающаяся операция в алгоритмах криптографии с открытым ключом. Обычно приходится выполнять умножение целых чисел по-простому или составному модулю или умножение полиномов по модулю неприводимого полинома.

Алгоритм Монтгомери — приём, позволяющий ускорить выполнение операций умножения и возведения в квадрат, необходимых при возведении числа в степень по модулю, когда модуль велик (порядка сотен бит), не требует операции деления. Был предложен в 1985 году *Питером Монтгомери*.

Для вычисления произведения Монтгомери $\mathrm{MP}(\mathrm{a}, b, N, R)$ достаточно перемножить операнды a и b как целые числа, а затем выполнить преобразование Монтгомери. Но можно действовать исключительно в терминах N-вычетов. Для вычисления произведения $ab\ (mod\ N)$ сначала переводят числа a и b в N-вычеты, вычисляя

$$MP(a, R^2(mod\ N), N, R) \equiv aR(mod\ N),$$
$$MP(b, R^2(mod\ N), N, R) \equiv bR(mod\ N).$$

Затем находят произведение N-вычетов

$$MP(aR, bR, N, R) \equiv (aR)(bR)R^{-1} \equiv (ab)R\ (mod\ N)$$

И наконец, выполняют обратное преобразование N-вычетов в число

$$MP(abR, 1, N, R) \equiv ((ab)R)R^{-1}(mod\ N).$$

Описание алгоритма

Вход: N-вычеты

$$aR(mod\ N) = (a_{n-1}, a_{n-2}, \dots, a_0)_B,$$
$$bR(mod\ N) = (b_{n-1}, b_{n-2}, \dots, b_0)_B,$$

где $B = 2^m, R = B^n, N' \equiv -N^{-1}(mod\ B)$, m – длина машинного слова.

Выход: N-вычет $c = (ab)R\ (mod\ N)$

Шаг 1. Положить $c\ \leftarrow 0, c = (c_{n-1}, c_{n-2}, \dots, c_0)_B$.

Шаг 2. Для $i\ = 0, 1, \dots, n-1$ выполнить следующие действия

Шаг 2.1. Положить $u_i\ \leftarrow (c_0 + a_i b_0)N'(mod\ B)$.

Шаг 2.2. Вычислить $c = \frac{c + a_i(bR\ (mod\ N)) + u_i N}{B}$

Шаг 3. При $c \geq N$ положить $c\ \leftarrow c - N$.

Шаг 4. Результат: c.

Операции умножения и деления на B выполняются очень быстро, так как при $B = 2^m$ представляют собой просто сдвиги бит на m. Таким образом алгоритм Монтгомери быстрее обычного вычисления $ab\ (mod\ N)$, которое содержит деление на N. Однако вычисление N^{-1} и перевод чисел в N- вычеты и обратно – трудоемкие операции, вследствие чего применять алгоритм Монтгомери при однократном вычислении произведения двух чисел представляется неразумным.

Анализ алгоритма

На рис. 2 представлен график, на котором видно, что для чисел, разрядность которых превышает 2500 десятичных знаков, алгоритм Монтгомери является лучшим, среди представленных в данной статье.

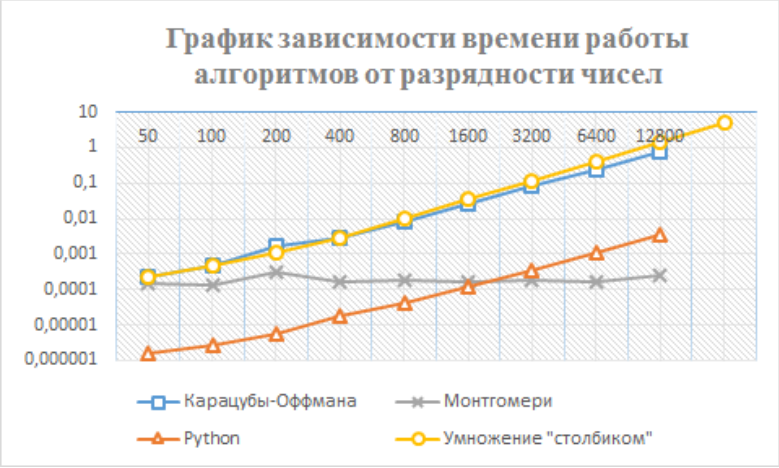

Рис 2. График зависимости времени работы (сек.) алгоритмов от количества десятичных разрядов (логарифмический масштаб)

Алгоритм умножения в классах вычетов (с использованием КТО)

Алгоритм умножения в классах вычетов, предложенный *А. Шенхаге*, использует китайскую теорему об остатках. Если целые числа $m_0, m_1, \dots, m_{k-1}$ попарно взаимно просты, то каждому целому числу $0 \le a < M$, где $M = m_0 m_1 \dots m_{k-1}$ единственным образом можно сопоставить набор элементов $a_0, a_1, \dots, a_{k-1}$ классов вычетов по модулям m_i, где $a_i \equiv a \ (mod\ m_i), 0 \le i < k$.

Сумме, разности произведению чисел по модулю M соответствуют суммы, разности и произведения по модулям m_i.

Наиболее трудоемкой операцией при вычислениях в классах вычетов является представление числа в классах вычетов и обратное восстановление по китайской теореме об остатках.

Алгоритм Гаусса для восстановления числа по его представлению в кольцах вычетов

Вход: число k - модули $m[k]$; Представление числа в классах вычетов $r[k]$

Выход: Целое число x:

Шаг 1. Вычислить $M = m_1 *..* m_t$, положить $x = 0$.

Шаг 2. Для $i = 0,..,k$ do

Шаг 2.1 $y_i := \frac{M}{m_i}$

Шаг 2.2 вычислить расширенным алгоритмом Евклида $si = y_i^{-1} \ mod\ m_i$

Шаг 2.3 $c_i = r_i * s_i \ mod\ m_i$

Шаг 2.4 $x = x + c_i * y_i \ (mod\ M)$

3. Результат: x

Алгоритм умножения:

Вход: $k, m[k]$ – массив модулей

$A[k], B[k]$ – числа A и B, представленные в классах вычетов

Выход: $C[k]$ – результат умножения

Шаг 1. Вычислить $M = m_1 *..* m_t$,

Шаг 2. Для $i = 0$ до $k - 1$

$C[i] = a[i] * b[i] \ (mod\ M)$

Шаг 3. Возврат C

Анализ алгоритма

Асимптотическая сложность алгоритма умножения в классах вычетов равна $O(k \ log\ k)$. Практически данный метод умножения эффективнее, чем умножение «в

столбик» и алгоритм Карацубы-Оффмана, лишь тогда, когда длина сомножителей превышает разрядность процессора в десятки раз. Если выполняется подряд несколько умножений небольших сомножителей без восстановлений по китайской теореме об остатках, то необходимо принять меры, исключающие «переполнение», при котором произведение превышает модуль М.

Параллельные вычисления

Большинство алгоритмов несимметричного шифрования используют операцию умножения для длинных чисел. Проблема увеличения производительности выполнения операций над такими числами остается актуальной уже длительное время. Но появлением многопроцессорных систем, стало возможным дальнейшее повышение эффективности таких операций, которое заключается в использовании параллельных вычислений для приведенных нами алгоритмов.

Сравнительный анализ алгоритмов факторизации чисел

Длинная арифметика так же нашла своё применение в алгоритмах шифрование. Например, при реализации метода шифрования RSA, требуется обеспечить точность результатов умножения и возведения в степень порядка 10^{309}. Но ещё более вычислительно сложной задачей является факторизация чисел.

В настоящее время не существует эффективного не квантового алгоритма факторизации целых чисел. Однако доказательства того, что не существует решения этой задачи за полиномиальное время так же нет.

Предположение о том, что для больших чисел задача факторизации является вычислительно сложной лежит в основе широко используемых алгоритмов (например, RSA). Множество областей математики и информатики находят применение в решении этой задачи. Среди них: эллиптические кривые, алгебраическая теория чисел и квантовые вычисления.

Факторизацией натурального числа называется его разложение в произведение простых множителей. Существование и единственность (с точностью до порядка следования множителей) такого разложения следует из основной теоремы арифметики.

Существует множество алгоритмов факторизации и их модификации, вот основные из них:

Экспоненциальные алгоритмы:

- перебор возможных делителей
- метод факторизации Ферма
- ρ - алгоритм Полларда;
- $\rho - 1$ - алгоритм Полларда;
- $\rho + 1$ алгоритм Вильямса;
- метод квадратичных форм Шенкса;
- метод Лемана.

Субэкспоненциальные алгоритмы:

- алгоритм Диксона;
- метод непрерывных дробей;
- метод квадратичного решета;
- метод эллиптических кривых.

В данной работе рассмотрены метод факторизации Ферма и две его модификации: алгоритм Диксона и метод квадратичного решета.

Метод факторизации Ферма — алгоритм факторизации (разложения на множители) нечётного целого числа n, предложенный Пьером Ферма (1601-1665) в 1643 году.

Метод факторизации Ферма основан на поиске такой пары целых чисел (x, y), которая удовлетворяет соотношению $x^2 - y^2 = n$. Тогда задача факторизации числа n решается так: $n = (x - y) \cdot (x + y)$. При этом не исключено, что $(x - y)$ и $(x + y)$

являются тривиальными множителями n (т.е. одно из них равно 1). Если оно является тривиальным, то n - простое.

Из равенства следует, что $x^2 - n = y^2$, откуда $x^2 - n$ является квадратом, следовательно искомое x находится в промежутке от $\sqrt{n}$ до n, т.е. значение $x^2 - n$ не должно быть отрицательным.

Для каждого значения x в заданном промежутке вычисляют $x^2 - n$ и проверяют не является ли это число квадратом. Если число не является квадратом, то x увеличивают на 1, иначе проверяют полученное разложение: $n = (x - \sqrt{x^2 - n}) \cdot (x + \sqrt{x^2 - n})$.

Большинство современных методов факторизации основано именно на этой идее.

Алгоритм факторизации Диксона.

В 20-х г. XX столетия Морис Крайчик (1882-1957), обобщая теорему Ферма предложил вместо пар чисел, удовлетворяющих уравнению $x^2 - y^2 = n$, искать пары чисел, удовлетворяющих более общему уравнению $x^2 = y^2 \, (mod\ n)$. Крайчик заметил несколько полезных для решения фактов. В 1981 г. Джон Диксон опубликовал разработанный им метод факторизации, использующий идеи Крайчика, и рассчитал его вычислительную сложность.

Алгоритм Диксона.

1. Составить факторную базу $B = \{p_1, p_2, \dots, p_h\}$, состоящую из всех простых чисел $p < M = L(n)^{\frac{1}{2}}$, где L(n) = exp($\sqrt{\ln n * \ln \ln n}$).

Факторной базой называют некоторое множество B небольших простых чисел.

2. Выбрать случайное b, где $\sqrt{n} < b < n$.
3. Вычислить $a = b^2 (mod\ n)$.
4. Проверить число a на гладкость пробными делениями.

В теории чисел гладким числом называется целое число, все простые делители которого малы. В нашем случае B-гладким будет являться число простые делители которого меньше B.

5. Если a является B – гладким числом, то есть $a = \prod_{p \in B} p^{a_p(b)}$, следует запомнить вектора $\bar{a}(b)$ и $\bar{\varepsilon}(b)$:

$\bar{a}(b) = (\mathrm{a}_{p1}(\mathrm{b}), \dots, \mathrm{a}_{ph}(\mathrm{b}))$

$\bar{\varepsilon}(b) = (\mathrm{a}_{p1}(\mathrm{b}) \bmod 2, \dots, \mathrm{a}_{ph}(\mathrm{b}) \bmod 2)$

6. Повторять процедуру генерации чисел b до тех пор, пока не будет найдено $h + 1$ B-гладких чисел $b_1, \dots, b_{h+1}$

7. Методом Гаусса найти линейную зависимость среди векторов $\bar{\varepsilon}(b_1), \dots, \bar{\varepsilon}(b_{h+1})$:

$\bar{\varepsilon}(b_{i1})\ xor\ \dots\ xor\ \bar{\varepsilon}(b_{it}) = \bar{0}$, где $1 \le t \le m$, и положить:

$x = b_{i1} \ \dots \ b_{it} \ (mod\ n)$

$$y = \prod_{p \in B} p^{\frac{\left(a_p(b_{i1}) + \dots + a_p(b_{it})\right)}{2}} (mod\ n)$$

8. Проверить $x = \pm y \, (mod\ n)$. Если это так, то повторить процедуру генерации. Если нет, то найдено нетривиальное разложение:

$n = u \cdot v$, где $u = \text{НОД}(x + y, n), v = \text{НОД}(x - y, n)$.

Метод квадратичного решета – метод факторизации больших чисел, разработанный Померанцем в 1981 году. Этот метод занимает вторую строчку в списке самых быстрых алгоритмов, уступая только методу 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. В основу этого метода вошли идеи Мориса Крайчика, обобщающего идею Фермы, и алгоритм Диксона.

Алгоритм квадратичного решета.

1. Составить факторную базу $B = \{p_1, p_2, \dots, p_h\}$, состоящую из всех простых чисел p, где $p < M = L(n)^{\frac{1}{2}}$, где L(n) = exp($\sqrt{\ln n * \ln \ln n}$).
2. Фильтруем факторную базу, оставив в B только такие элементы p, для которых n является квадратичным вычетом по модулю p. Используем для этого символ Лежандра.
3. Чтобы числа $f(x) = \left(x + \left[\sqrt{x}\right]\right)^2 - n$ были с большей вероятностью B-гладкими, в методе квадратичного решета необходимо осуществить просеивание: решая сравнение $\left(x + \left[\sqrt{x}\right]\right)^2 = n \, (mod\ p)$ для каждого $p \in B$, мы получим $x_1(p_i)$ и $x_2(p_i)$ по модулю p_i. Таким образом, в интервале [-c, c] можно оставить только те числа x, которые удовлетворяют решению сравнений $x_1(p_i)$ и $x_2(p_i)$ для достаточно большого числа элементов $p_i \in B$.
4. Подберем оптимальное значение c. Выполним последний этап просеивания: из целых чисел $|x| \leq c$ выберем те, которые удовлетворяют трем и более сравнениям.
5. Для каждого из чисел x проверяем значение $f(x)$ на B-гладкость. Таких чисел должно получиться $h + 1$.
6. Получить единичные вектора e для каждого разложения: если степень вхождения числа четная – 0, нечетная – 1.
7. Методом Гаусса найти линейную зависимость ($e_1 xor\ e_2 = 0$) среди векторов $e_1, .., e_p$.
8. $A = \prod_{x \in X}(x + m),\ B^2 = \prod_{x \epsilon X} f(x)$
9. $n = u \cdot v$, где $u = $ НОД$(x + y, n), v = $ НОД$(x - y, n)$.

В процессе выполнения данной работы, был выявлен рядом проблем:

В алгоритме Диксона при генерации h+1 случайных B-гладких чисел, на больших значениях входного числа n, может возникнуть зацикливание из-за невозможности найти B-гладкое число. Эту проблему можно решить увеличением факторной базы B, что может привести к дальнейшему усложнению алгоритма в результате увеличения количества и размера единичных векторов $\bar{\varepsilon}(b_1), \dots, \bar{\varepsilon}(b_{h+1})$. Была заменена случайную генерацию числе из диапазона $(\sqrt{n}, n)$, на перебор чисел начиная с $\sqrt{n}$. Эта дало небольшой прирост в скорости генерации B- гладких чисел. Так же можно улучшить производительность алгоритма Диксона заменой метода Гаусса на один из методов, более подходящих для разреженных матриц. Именно разреженные матрицы и будут получаться в результате нахождения векторов $\bar{\varepsilon}(b)$. Вот некоторые из них:

1. Алгоритм Ланцоша - почти не использует дополнительной памяти, кроме матрицы;
2. Алгоритм Видеманна;
3. Копперсмит - блочный алгоритм Видеманна;

Их сложность O(n^2).

Алгоритм квадратичного решета построен на основе идей Ферма и Крайчика, поэтому, как и алгоритм Диксона, подвержен тем же проблемам. Основные трудности возникают в процессе построения факторной базы и выбора интервала поиска $|x| \leq c$, где размер факторной базы и интервал $[-c, c]$ выбираются из условия оптимальности. Это очень сильно влияет на работу алгоритма. Одним из способов решения данной проблемы для процедуры просеивания (так же и для Диксона) является распараллеливание задачи на более мелкие. Этап нахождения линейной зависимости векторов можно ускорить, заменив метод Гаусса одним из предложенных ранее алгоритмом.

Алгоритм Диксона и метод квадратичного решета имеют субэкспоненциальную функцию сложности. Стандартная субэкспоненциальная функция сложности имеет вид

L_N(a, c) =exp((c+o(1))$(\ln n)^a (\ln \ln n)^{1-a}$).

Алгоритм Диксона имеет сложность $L_N(1/2,\ 2\sqrt{2})$, а метод квадратичного решета $L_N(1/2,\ 1)$. В результате мы можем построить график зависимости вычислительной сложности от длины числа.

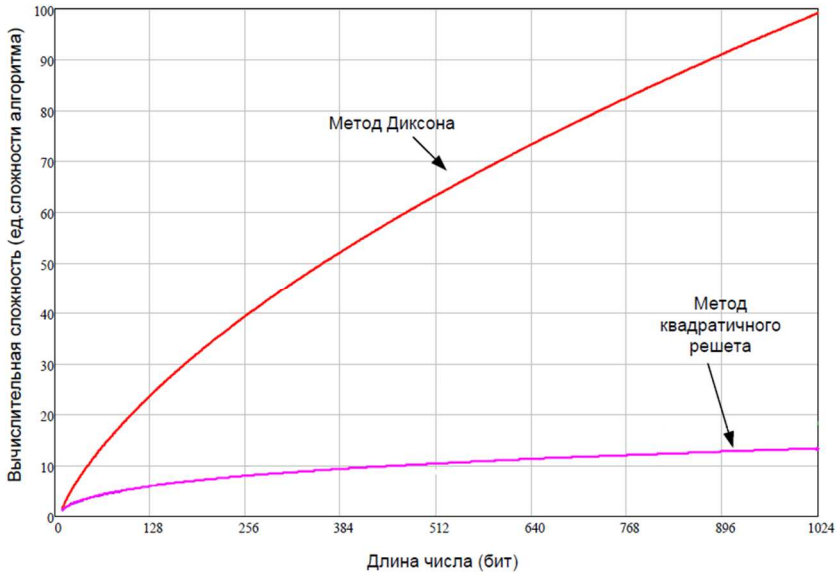

Рис 3.График сравнения, зависимости времени факторизации от длинны факторизованного числа, методами Диксона и квадратного решета.

По графику видно превосходство метода квадратичного решета практически для любой длины числа. Следовательно, будет целесообразно в криптографических системах использовать именно этот алгоритм факторизации. Благодаря этому методу, в 1994 г. Аткинс, Граф, Лейланд и Ленстра сумели разложить 129-значное число, предложенное создателями RSA, не имея технических возможностей доступных в наши дни.

Также, не следует забывать о выборе языка программирования, так как некоторые языки имеют превосходство в скорости над другими (например, на некоторых тестах python уступает c++ до 100 раз в вычислительной способности), а как раз скорость и является основным критериям в криптосистемах.

Несмотря на большое количество существующих методов, при определенной длине ключа криптосистема имеет высокую стойкость к факторизации. Поэтому задача модификации существующих методов факторизации очень актуальна. Наряду с модификацией уже существующих методов, очень важна разработка принципиально новых алгоритмов. Но, тем не менее, модификацией уже существующих методов можно добиться хороших результатов.

Основными направлениями модификации являются распараллеливание вычислительных задач [1], использование быстрых вычислений, применение новых алгоритмов, оптимизация процесса вычислений и другие.

Список литературы:

1. *Белан В.И., Белоус Л.Ф., Поляков В.М., Торгонин Е.Ю., Хутайфа А*. Применение гетерогенных вычислительных систем для описания процессов в системах виртуальной реальности //Материалы международной научно-технической конференции «Parallel and Distributed Computing Systems (PDCS 2014)». –Харьков, 2014. –С.41–44.
2. *Сергей Николенко*. Разложение чисел на множители: лекции / Криптография «CS Club», 2009. – с. 52
3. Википедия. Факторизация целых чисел: статья / http://ru.wikipedia.org/wiki/Факторизация_целых_чисел.
4. *Ишмухаметов Ш.Т*. Методы факторизации натуральных чисел: учебное пособие / Казань: Казан. ун. 2011. – с. 190

5. *Черемушкин А. В.* Лекции по арифметическим алгоритмам в криптографии / М.: МЦНМО, 2001. – с. 104
6. Википедия. Метод факторизации Ферма: статья / http://ru.wikipedia.org/wiki/ Метод_факторизации_Ферма.
7. Карацуба А., Оффман Ю. Умножение многозначных чисел на автоматах / Доклады Академии Наук СССР. — 1962. — Т. 145. — № 2.
8. Карацуба А. А. Сложность вычислений / Тр. МИАН. — 1995. — Т. 211. — 202 с..
9. Колмогоров А. Н. Теория информации и теория алгоритмов / Москва: Наука, 1987.
10. Мао, Венбо Современная криптография: теория и практика. / М.: Издательский дом «Вильямс», 2005. — 768 с.
11. Менезис А., Пол ван Оорсхот, Ванстоун С. Прикладная криптография / CRC-Press, 1996. — 816 с.
12. Фергюсон, Нильс, Шнаер, Брюс Практическая криптография / М.: Вильямс, 2004. — 432 с.