УДК 004.056.55

РЕАЛИЗАЦИЯ ВЕРОЯТНОСТНОГО ТЕСТА НА ПРОСТОТУ СОЛОВЕЯ-ШТРАССЕНА НА ЯЗЫКЕ C#

Мыздриков Н. Е., Черкесова Л. В., Сафарьян О. А.
Донской государственный технический университет, Ростов-на-Дону, Российская Федерация
nmyzdrikov@mail.com
chia2002@inbox.ru
safari_2006@mail.ru

Представлен анализ вероятностного теста на простоту числа Соловея-Штрассеня и его возможные модификации. Приведены новые алгоритмы и код разработки на языке C#.

Ключевые слова: простое число, криптосистема, вероятностный тест, бинарное возведение в степень, символ Якоби, малая теорема Ферма.

UDC 004.056.55

REALIZATION OF THE SOLOVAY-STRASSEN PRIMALITY TEST IN THE PROGRAMMING LANGUAGE C#

Myzdrikov N. E., Cherkesova L. V., Safaryan O. A.
Don State Technical University, Rostov-on-Don, Russian Federation
nmyzdrikov@mail.com
chia2002@inbox.ru
safari_2006@mail.ru

The article presents the analysis of Solovay-Strassen primality test, as well as its possible modifications. New algorithms are presented and the development code in C# is given.

Key words: prime number, cryptosystem, probabilistic test, binary exponentiation, Jacobi symbol, Fermat's little theorem.

Введение. Для многих криптосистем с открытым ключом жизненно необходимым условием для их постоянной работы является стабильный поток простых чисел. Например, для использования протокола Диффи-Хеллмана и ему подобных алгоритмов необходимо сгенерировать простое число p, которое будет являться размером поля кольца вычетов F_p. Хорошим примером может послужить криптосистема Эль-Гамаля с открытым ключом. В этом алгоритме простые числа нужны для создания открытого ключа.

Следует заметить, что для качественной и бесперебойной работы многих криптосистем и протоколов нужна высокая скорость генерации этих самых простых чисел.

С другой стороны, может появиться вопрос о том, стоит ли их вообще генерировать, если их можно разом все посчитать и хранить где-либо до необходимости использовать. Для ответа на подобный вопрос можно провести испытания над алгоритмом «Решето Эратосфена», где время увеличивается параболически с увеличением объема выборки. Таким образом, подсчеты всех простых чисел в пределах не самого большого числа -2^{256} могут занять долгие месяцы. Также может появиться необходимость в нахождении простых чисел по модулю поля.

Все эти вычисления займут намного больше времени, чем определение числа методом простого перебора в необходимых границах с последующими тестами на его простоту [2].

Достаточно хорошо себя зарекомендовали вероятностные тесты простоты числа, которые не дают полной гарантии, что проверенное число явно простое, но вероятность увеличивается с каждой итерацией теста. Таким образом, если тест провести несколько раз, то вероятность простоты числа увеличится. Существует не так много вероятностных алгоритмов. Многие из них опираются на малую теорему Ферма.

В данной работе будет рассмотрен вероятностный тест на простоту числа Соловея-Штрассена. Его главным достоинством по отношению к простому тесту Ферма стала возможность распознавать числа Кармайкла как составные.

Целью работы является исследование теста Соловея-Штрассена, реализация его алгоритма на языке C# и анализ его производительности.

В процессе достижения поставленной цели авторами были сформулированы и успешно решены следующие задачи: исследование теста Соловея-Штрассена, выявление возможных модификаций теста; анализ времени работы теста по отношению к другим тестам на простоту [3].

Основная часть. Тест Соловея-Штрассена на выявление простоты у числа основывается на малой теореме Ферма, которая звучит следующим образом:

«Если p — простое число и a — целое число, взаимно простое с n, то:

$a^{n-1} \equiv 1(mod\ n)$. *И так для любого a в пределах* $0 < a < n - 1$»

Суть самого теста в том, чтобы проводить тест не над каждым числом из всей последовательности, а над случайным набором разных случайных чисел k раз.

При этом для выявления чисел Кармайкла используются свойства символа Якоби. Число, сгенерированное случайным образом во время теста и удовлетворяющее равенству $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}}(mod\ n)$, где $\left(\frac{a}{n}\right)$ — символ Якоби, называется свидетелем простоты n [1].

В зависимости от значений a и n, символ Якоби может быть равен 1 или -1.

Если по итогам всего тестирования свидетелей простоты числа n было обнаружено столько же, сколько и итераций k, то число n называется вероятно простым, с вероятностью в $1 - 2^{-k}$.

Применение теста Соловея-Штрассена в криптографии

Вероятностные тесты применяются в системах, основанных на проблеме факторизации, например, RSA или схема Рабина. Тест Соловея-Штрассена может быть использован везде, где есть необходимость в быстрой проверке некоторого числа на простоту.

Однако на практике степень достоверности теста Соловея-Штрассена не является достаточной. Вместо него используется тест Миллера-Рабина, так как его точность значительно выше. Более того, используются объединенные алгоритмы. Например, пробное деление и тест Миллера-Рабина или последовательные тестирования. При правильном выборе параметров можно получить результаты лучше, чем при применении каждого теста по отдельности [1].

Алгоритм теста для программной реализации

Вход: n>2, тестируемое нечётное натуральное число, k , параметр, определяющий точность теста.

Выход: составное, означает, что n — точно составное, вероятно простое, означает, что n с вероятностью в $1 - 2^{-k}$ является простым.

Цикл: $i = 1,2, \dots, k$;

$a =$ случайное целое от 2 до $n - 1$, включительно;

Если наибольший общий делить — НОД(a, n)>1, тогда:

Вывести, что n — составное, и **Остановиться**.

Если $a^{\frac{n-1}{2}} \neq \left(\frac{a}{n}\right)(mod\ n)$,тогда:

Вывести что n — составное и остановиться.

Иначе: Перейти к следующей итерации в цикле.

Вывести: $\boldsymbol{n}$ — вероятно простое, с шансом в $\mathbf{1 - 2^{-k}}$ [2].

Существующие модификации теста Соловея-Штрассена

В 2005 году на Международной конференции «Informational Technologies» А. А. Балабанов, А. Ф. Агафонов, В. А. Рыку предложили модернизированный тест Соловея-Штрассена. Тест Соловея-Штрассена основан на вычислении символа Якоби, что занимает время. Идея улучшения состоит в том, чтобы, в соответствии с теоремой квадратичной взаимности Гаусса, перейти к вы-

числению величины, являющейся обратной символу Якоби. Это является более простой процедурой [2].

Возможные модернизации с использованием существующих решений

Для ускоренного поиска НОД числа воспользуемся бинарным алгоритмом. Это позволит уменьшить затраты времени на поиск общего делителя чисел в алгоритме теста. Главное отличие бинарного алгоритма от классического заключается в том, что в нем используются побитовые сдвиги числа, для ускоренного деления на 2.

В качестве быстрого алгоритма возведения в степень будет выступать его улучшенная, рекурсивная бинарная версия, которая позволит сократить время выполнения операции.

Символ Якоби (алгоритм)

1. **Если** НОД (a, b) ≠1, выход из алгоритма с ответом 0.
2. r:=1.
3. **Если** a<0 то a:=-a
   Если b (mod 4) = 3 то r:= -r
4. t:=0

Цикл ПОКА a — чётное
t:=t+1
a:=a/2
Конец цикла
Если t — нечётное, то
Если b (mod 8) = 3 или 5, то r:=-r.

5. **Если** a (mod 4) = b (mod 4) = 3, то r:=-r.
   c:=a; a:=b (mod c); b:=c.
6. **Если** a≠0, то идти на шаг 4, иначе выйти из алгоритма с ответом r.

Код алгоритма на C#

```
public static long Yacobi(long a, long b)
  {
      int r = 1;
      while (a != 0)
      {
          int t = 0;

          while ((a & 1) == 0)
          {
              t++;
              a >>= 1;
          }

          if ((t & 1) !=0)
          {
              long temp = b % 8;
              if (temp == 3 || temp == 5)
              {
                  r = -r;
```

```
            }
        }
        long a4 = a % 4, b4 = b % 4;
        if (a4 == 3 && b4 == 3)
        {
            r = -r;
        }
        long c = a;
        a = b % c;
        b = c;
    }
    return r;
}
```

Алгоритм быстрого возведения в степень по модулю.

Как было указано ранее, для улучшения производительности стоит улучшить быстродействие возведения числа в степень по модулю. Это необходимо для проверки основного утверждения $a^{\frac{n-1}{2}} \neq \left(\frac{a}{n}\right) (mod\ n)$.

Для упрощения записи, метод, который реализует быстрое возведение в степень, будет называться POW.

1. **На вход** алгоритма $\boldsymbol{POW}$ поступают значения: $\boldsymbol{a}$— целое число, $\boldsymbol{n}$ – искомая степень числа a, $\boldsymbol{Gf}$ – модуль над a^n.
2. **Если** $n = 0$ **вернуть** 1.
3. **Если** n — нечетное, **вернуть** результат (**вызвать** $POW(a, n-1, Gf) * a$) по модулю Gf.
4. **Иначе** добавляется переменная b.

4.1 **Если** n делится на 4 без остатка, то b присваивается остаток от деления $POW\left(a, \frac{n}{4}, Gf\right)$ на Gf.

4.1.1 b возводится в квадрат и результат берется по модулю Gf. Затем полученный результат снова возводится в квадрат.

4.1.2 **Вернуть** остаток от деления b на Gf.

4.2 **Иначе** b присвоить значение остатка деления $POW(a, \frac{n}{2}, Gf)$ на Gf.

Вернуть остаток от деления b^2 на Gf.

Реализация приведенного алгоритма на языке C#.

```
public static long binpow(long a, long n, long Gf)
{

    if (n == 0) return 1;
    if (n % 2 == 1) return (binpow(a, n - 1, Gf) * a) % Gf;
    else
    {
```

```
            long b;
            if (n % 4 == 0)
            {
                b = binpow(a, n / 4, Gf) % Gf;
                b *= b; b *= (b % Gf);
                return b % Gf;
            }
            else
            {
                b = binpow(a, n / 2, Gf) % Gf;
                return (b * b) % Gf;
            }
        }
    }
```

Код на C# для поиска наибольшего общего делителя:

```
    public static int BinaryGCD(int A, int B)
    {
        int k = 1;
        while ((A != 0) && (B != 0))
        {
            while (((A & 1) == 0) && ((B & 1) == 0))
            {
                A >>= 1;
                B >>= 1;
                k <<= 1;
            }
            while ((A & 1) == 0) A >>= 1;
            while ((B & 1) == 0) B >>= 1;
            if (A >= B) A -= B; else B -= A;
        }
        return B * k;
    }
```

Результаты тестирования

После написания программы на языке C# были произведены тесты. На их основании можно сделать вывод об эффективности этой реализации. В таблице 1 приведены результаты работы программы с числами разной длинны при цикле в 1000 повторений.

Таблица 1

Затраты времени на работу

Число X	X≈250	X≈ 10^3	X≈ 10^5	X≈ 10^7	X≈ 10^9	X≈ 10^{11}	X≈ 10^{13}
Класич.	0,144	0,243	1,522	2,320	3,854	5,216	7,562
Разработ.	0,257	0,351	0,629	0,921	1,237	1,695	2,350

Основываясь на полученных результатах можно заметить, что эффективность реализованного алгоритма хуже на числах меньшего размера, но значительно лучше при работе с числами от 10^5и выше. На основании проделанных экспериментов можно сделать вывод, что разработанный алгоритм будет более эффективным на практике, так как во многих случаях основная потребность в проверке на простоту будет для больших чисел.

Заключение. Быстрый поиск простых чисел остается важной задачей на сегодняшний день, так как многие криптографические протоколы используют простые числа для работы. Реализованный в ходе работы программы алгоритм эффективно справляется с этой задачей.

Основной инновацией в проделанной работе является алгоритм быстрого возведения в степень, позволявший сократить время расчетов. Все алгоритмы были реализованы в их бинарном представлении, что также позволило сократить затраты ресурсов.

По итогам проделанной работы можно заключить, что модифицированный тест Соловея-Штрассена в более крупных криптографических реализациях очень эффективен в использовании.

Библиографический список

1. W. R. Alford, A. Granville, C. Pomerance (1994). «There are Infinitely Many Carmichael Numbers». Annals of Mathematics 139: 703-722. DOI:10.2307/2118576

2. Тест Соловея-Штрассена [электронный ресурс] / Википедия. — Режим доступа : www/URL: https://ru.wikipedia.org/wiki/Тест_Соловея_—_Штрассена (дата обращения : 14.04.2018).

3. Коблиц, Н. Курс теории чисел и криптографии / Н. Коблиц ; перев. с англ. М. А. Михайловой, В. Е. Тараканова; под ред. А. М. Зубкова. — 2-е. изд. перераб. и доп. — Москва : Научное издательство ТВП, 2001. — 254 с.