Математика и ее приложения : ЖИМО. 2013. Вып. 1 (10). С. 77–84

УДК 512.543

С. И. Хашин[1]

C++-класс для работы с небольшими простыми числами

Ключевые слова: псевдопростые числа, факторизация.

При изучении дискретной математики, языков программирования, компьютерной графики часто требуется работать с небольшими простыми числами. Есть много хороших реализаций, но в основном они являются частью больших и сложных проектов. Включать весь проект в небольшую, учебную программу часто нецелесообразно. Поэтому и был разработана свой, небольшой и компактный класс на языке C++.

Keywords: pseudoprime numbers, integer factorization.

In the study of discrete mathematics, programming languages, computer graphics are often required to work with small primes. There are many good implementations, but mostly they are part of a large and complex projects. Include the entire project into a small, curriculum often impractical. Therefore, and was developed its small and compact C++ class.

1. Введение

На сегодняшний день C++ библиотеки, реализующие различные теоретико-числовые, криптографические алгоритмы стали очень эффективными и сложными. Например, библиотека GMP [6] содержит свыше 1700 файлов и почти 200 000 строк кода. Настройка библиотек на различные компиляторы тоже не всегда проста.

Во многих приложениях, в графике, видео, распознавании образов ([1, 3] и др.) не требуются все возможности таких библиотек, а нужны всего несколько функций, работающих с не очень большими числами. В такой ситуации хотелось бы иметь небольшую, простую в подключении и использовании библиотеку, пусть и работающую с числами небольшой длины, до 64 бит. Именно такая библиотека и описана в работе. Она доступна в интернете [2] без ограничений и регистрации. Этот модуль ока-

[1]Ивановский государственный университет; E-mail: khash2@mail.ru. Работа выполнена при поддержке гранта РФФИ 11-07-00653

зался полезным и в учебных целях, при изучении дискретной математики и математических основ криптографии.

Также оказалось удобно использовать модуль для проверки различных теоретико-числовых гипотез, например из [4, 5], исследовании алгоритмов проверки простоты чисел [7].

Весь модуль состоит из двух файлов: `z64.h` – заголовочный файл и `z64.cpp` – реализация класса. В нем предлагается набор функций для работы с небольшими (до 2^{64}) числами. Функции организованы в виде статических членов класса `z64`, динамических членов у класса нет. Для работы не требуются никакие дополнительные библиотеки. Динамическое выделение памяти также не используется.

Настройка пакета на различные процессоры и компиляторы сводится к корректировке двух строк в заголовочном файле:

```
// Visual C++
typedef unsigned __int64 UINT64;
typedef          __int64  INT64;
```

В этих строках должны быть заданы два типа данных `INT64` и `UINT64`, используемых во всех дальнейших вычислениях. Все остальные части модуля не зависят ни от компилятора, ни от операционной системы.

2. Основные методы

Основные методы класса (все методы – статические, слово `static` опущено):

```
bool   isFullSq(UINT64 n);               //n is a full square?
UINT64 gcd(UINT64 a, UINT64 b);          // GCD(a,b)
UINT64 mul_mod(UINT64 a, UINT64 b, UINT64 c);// (a*b)%c
UINT64 pow_mod(UINT64 a, UINT64 b, UINT64 c);// (a^b)%c
void   euclid (UINT64 a,UINT64 b,INT64 &x,INT64 &y,UINT64 &d);
                  // find x,y,d: a*x+b*y=d=gcd(a,b)
UINT64 inverse (UINT64 a, UINT64 n);          // 1/a mod n
bool   chineese(UINT64  a1, UINT64  b1,       // cheneese
                UINT64  a2, UINT64  b2,       // remainder
                UINT64 &a3, UINT64 &b3);      // theorem
```

Следующая группа методов относится к исследованию простых чисел. Они позволяют проверить число на простоту, найти i-е простое число,

количество простых чисел, меньших данного, следующее и предыдущие простые числа и наименьший простой делитель числа:

```
bool   isPrime( UINT64 n);  // is prime(n)?
 UINT64 p( UINT64 i);       // i-th prime,
     // in Maple notation: ithprime(i),
     // p(0)=0, p(1)=2, p(2)=3,...
UINT64 Pi( UINT64 n);        // the number of primes <= n,
     // Pi(0)=Pi(1)=0, Pi(2)=1, Pi(3)=Pi(4)=2,...
UINT64 NextPrime( UINT64 n);  // prime p>=n,
     // NextPrime(3)=5, NextPrieme(4)=5,
     // NextPrime(0)=NextPrime(1)=2, NextPrime(2)=3
UINT64 PrevPrime( UINT64 n);  // prime p<=n,
     // PrevPrime(0)=PrevPrime(1)=PrevPrime(2)=0,
     // PrevPrime(4)=3
UINT64 primeFactor( UINT64 n);// prime divisor or 1,
     // if not found
```

Имеется группа теоретико-числовых методов, позволяющих найти порядок числа в мультипликативной группе вычетов, символ Якоби и извлечь квадратный корень по простому модулю:

```
 UINT64 order( UINT64 a, UINT64 n);
             // order of a by modulo prime n
 int jacobi(UINT64 a, UINT64 n);// Jacobi symbol
 UINT64 msqrt( UINT64 a, UINT64 p);
             // sqrt(a) mod p, p must be prime!
```

Преобразование в строку:

```
 char * factorString(UINT64 n); //string with factorisation:
             // 180200 -> "2^3*5^2*17*53"
 void    toString(char *s, UINT64 n, int w=0);
             // s <- to String of width w
 char   *toString(UINT64 n, int w=0);
             // return to String of width w
```

Рассмотрим некоторые из методов более подробно.

2.1. Решето Эратосфена. Механизм решета Эратосфена решает, фактически, две задачи: быстрая, за одно обращение к таблице провер-

ка простоты числа и нахождение функции `pi(k)` – количества простых чисел $\leq k$.

Решето представляет собой битовый массив длины N, где бит, соответствующий натуральному числу равен 1 для простых чисел и 0 для составных. Платой за быструю проверку простоты является большой расход памяти. При $N = 2^{32}$ ее требуется 500 Мбайт. Расход памяти можно уменьшить, если хранить биты только для нечетных чисел. Объем памяти уменьшается вдвое, скорость работы остается прежней.

Еще более удобным оказывается следующий метод. Рассмотрим 30 целых чисел, идущих подряд: $30k,\ 30k+1, \ldots, 30k+29$. Среди них не делящимися ни на 2, ни на 3, ни на 5 будут только 8 чисел:

$$30k+1,\ 30k+7,\ 30k+11,\ 30k+13,\ 30k+17,\ 30k+19,\ 30k+23,\ 30k+29.$$

Поэтому в k-м байте решета будем хранить признаки простоты этих восьми чисел. В этом случае решето при $N = 2^{32}$ будет занимать чуть меньше 137 Мбайт, а в одном гигабайте памяти поместится список простых чисел до 30 миллиардов.

Решето заполняется сравнительно быстро. Список простых меньших 2^{24} строится на процессоре AMD Athlon II X2 240, 2.8 GHz примерно за 0.1 сек., меньших 2^{32} – примерно за 1 минуту, до 2^{34} – около 5 минут. Таким образом, если вашей программе достаточно списка простых чисел до 2^{24}, то решето можно вычислять каждый раз заново (метод `Efill`). Но если нужны простые числа до 2^{32} и более, то решето лучше читать из заранее созданного файла (метод `Eload`).

С помощью решета (в пределах до `N=maxP()`) легко реализуются функции `isPrime, NextPrime, PrevPrime`. Однако для фукнций `p()` `(ithprime)` и `Pi()` требуются дополнительные данные. Одновременно с построением решета, запоминаются в отдельном массиве значения `pi(k)` (количество простых, не больших `k`) для каждого `k`, кратного 1024. Таким образом, для хранения самого решета до числа N требуется $N/16$ байт, и ешч $N/256$ байт для этой дополнительной таблицы. С ее помощью быстро работают функции

```
UINT64 p ( UINT64 i);
UINT64 Pi( UINT64 n);
```

Эти функции взаимно-обратны: `z64::Pi(z64::p(i)) == i`.

Для совсем небольших простых чисел (до 8000) все проверки проводятся по заранее заготовленным таблицам.

2.2. isPrime. Проверка числа на простоту.

```
bool isPrime( UINT64 n); // is prime(n)?
```

В пределах имеющегося решета Эратосфена простота числа проверяется с его помощью. Если число n меньше 2^{32} и не попадает в решето, то проводится проверка с помощью метода Миллера-Рабина по основаниям 2 и 3 и затем проверяются по таблице все 103 составных числа (`unsigned BCompos[103]`), меньшие 2^{32}, на которых ошибается метод сразу по обоим основаниям. Таким образом, для чисел меньших 2^{32} ответ всегда точный (а не вероятностный). Для больших чисел пока просто проводится 12 тестов Миллера-Рабина по основаниям $2, 3, \ldots, 37$.

2.3. mul_mod. Это – ключевая функция пакета. Реализация ее на ассемблере ускоряет работу более, чем на порядок. Ассемблерная реализация проверена лишь для `gcc-64` под `Linux`. Отметим, что в 32-разрядных системах ассемблерная реализация невозможна.

```
UINT64  z64::mul_mod(UINT64 a, UINT64 b, UINT64 c)
                                          //(a*b)%c
{ #if defined(_MSC_VER) || defined( _WIN32 )
    UINT64 res=0;
    if(a>c) a %= c;
    while(b){
        if(b&1) {res+=a; if(res>c) res-=c; b--; }
        else    { a<<=1; if(a  >c) a  -=c; b>>=1;}
    }
    return res;
#else
  UINT64 d;        // to hold the result of a*b mod c
                   // calculates a*b mod c, stores result in d
  asm ("mov %1, %%rax;"       // put a into rax
       "mul %2;"              // mul a*b->rdx:rax
       "div %3;"              //(a*b)/c->quot in rax,
                              //    remainder in rdx
       "mov %%rdx, %0;"       // store result in d
       :"=r"(d)               // output
       :"r"(a), "r"(b), "r"(c)// input
       :"%rax", "%rdx"        // clobbered registers
```

```
      );
  return d;
#endif
} // z64::mul_mod
```

2.4. primeFactor. Нахождение простого множителя.

```
static bool  primeFactor( UINT64 n);
// prime divisor or 1, if not found
```

По результатам некоторых исследований и большого числа экспериментов в наших пределах (до 2^{64}) из элементарных методов наиболее эффективен оказался ρ-метод Полларда, все остальные не стоит и реализовывать. Если все простые множители у числа велики (несколько миллиардов), то методу может потребоваться до 2-3 сотен тысяч шагов. Но остальные методы – все равно еще медленнее. По крайней мере, например, разложение числа $p-1$ на множители для простого p (это используется в функции `order`) находится без труда.

Кроме того, метод не работает на квадратах простых чисел, поэтому их рассматриваем отдельно:

```
UINT64 sn = (UINT64) sqrt ((double)n);
if( sn*sn==n ) return sn;
```

Эта проверка на числах, меньших 2^{64} всегда является точной (проверено!).

2.5. factorString. Строка с разложением на простые множители.

```
char *factorString(UINT64 n);
 // string with factorisation: 180200 -> "2^3*5^2*17*53"
```

Эта функция использует всегда одну и ту же статическую область памяти для формирования строки. То есть возвращаемый указатель – всегда один и тот же. Память не надо ни выделять, ни освобождать. Надо только не забывать, что при повторном вызове этой функции, она снова использует ту же самую память. Если требуется преобразовать в строки сразу несколько чисел, то можно использовать функцию

```
void toString(char *s, UINT64 n, int w=0);
 // s <- to String of width w
```

Но о выделении достаточной памяти тогда уж придется заботится самому.

3. Пример использования

```
#include "z64.h" ...

   z64::Efill(1<<24);              // fill the sieve till 2^24
   z64::Esave("16M.sieve");        // save the sieve
   UINT64 p = z64::p(1000000);     // p = 15485863
   UINT64 i = z64::Pi(p);          // i =  1000000
   UINT64 p2= z64::p( 700000);     // p = 10570841
   UINT64 n = p*p2;                // n = 163 698 595 520 783
   bool   r = z64::isPrime(n);     // r = false
   UINT64 f = z64::primeFactor(n); // f = 10570841
```

4. Скорость работы

В основном, модуль разработан для учебных и исследовательских целей, скорость работы не являлась главной задачей. Однако при размере чисел в 64 бита, все равно вычисления получаются более быстрыми,чем при использовании библиотеки `GMP`. Например операция `mul_mod` на процессоре AMD Athlon II X2 240, 2.8 GHz занимает для 63-битовых чисел:

разрядность	язык	время, нс
32	C++	75
64	C++	31
64	asm	2.7
64	GMP	7

Результат вполне ожидаемый, надо только помнить, что библиотека `GMP` будет почти с той же скоростью обрабатывать и более длинные числа, а наш модуль (`z64`) для этого не предназначен.

Список литературы

1. *Хашин С. И.* Динамическая сегментация последовательности кадров // Машинное обучение и анализ данных. 2013. Т. 1. № 6. С. 787–795.
2. *Хашин С. И.* Небольшие простые числа на C++. 2013. URL: http://math.ivanovo.ac.ru/dalgebra/Khashin/Eratosthenes/z64_r.html (дата обращения: 12.12.2013).
3. *Хашин С. И., Хашина Ю. А.* Свойства b-ранга системы булевых полиномов. // Математика и ее приложения : журн. Иван. мат. о-ва. 2013. Вып. 1(10). С. 65–70.

4. *Crandall R. E., Pomerance C.* Prime Numbers: A Computational Perspective (Second Edition). Springer-Verlag, 2005. 597 p.

5. *Damgard I. B., Frandsen G. S.* An Extended Quadratic Frobenius Primality Test with Average- and Worst-Case Error Estimate // Journal of Cryptology. 2006. Vol. 19, № 4. P. 489–520.

6. *GMP* , The GNU Multiple Precision Arithmetic Library. 2013. URL: https://gmplib.org/ (дата обращения: 12.12.2013).

7. *Khashin S. I.* Counterexamples for Frobenius primality test // http://arxiv.org/abs/1307.7920.

Поступила в редакцию 12.12.2013.