MAXimal

added: 10
Edited: 10

test BPSW the simplicity of numbers

Contents [hide]

Introduction

Algorithm BPSW - this is a test of the simplicity. This algorithm is named for its inventors: Robert Bailey (Ballie), Carl Pomerance (Pomerance), Selfridge (Selfridge), Samuel Wagstaff (Wagstaff). Algorithm was proposed in 1980. To date, the algorithm has not been found any counterexar as the proof has not been found.

BPSW algorithm was tested on all numbers 10 to ^{15} . Furthermore, counterexample to find using the PRIMO (cm. [6]), based on simple test usir curves. The program worked for three years, did not find any counterexample, based on which Martin has suggested that there is no single BPS pseudosimple smaller 10 ^{10000} (pseudosimple number - a composite number on which the algorithm gives the result "simple"). At the same time Pomerance in 1984 presented a heuristic proof that there are infinitely many BPSW-pseudosimple numbers.

Complexity of the algorithm BPSW is O (log ^{3} (N)) bit operations. If we compare the algorithm BPSW with other tests, such as the Miller-Rabin te algorithm BPSW is usually 3-7 times slower.

Algorithm is often used in practice. Apparently, many commercial mathematical packages, wholly or partly rely on an algorithm to check BPSW

Brief description of

The algorithm has several different implementations, differing only in details. In this case, the algorithm is:

1 Run Miller-Rabin test to the base 2.

2 Run a strong Lucas-Selfridge test using Lucas sequence with parameters Selfridge.

3 Return the "simple" only when both tests returned "simple".

+0. In addition, at the beginning of the algorithm can add a check for trivial divisors, say, 1000 This will increase the speed of operation on a cor number, however, has slowed somewhat in the simple algorithm.

Thus, the algorithm BPSW based on the following:

1 (true) test and the Miller-Rabin test Lucas-Selfridge and if wrong, it is only one way: some components of these algorithms are recognized as s Conversely, these algorithms do not make mistakes ever.

2 (assumption) Miller-Rabin test and the test of Lucas-Selfridge and if wrong, that are never wrong on one number at a time.

In fact, the second assumption seems to be as incorrect - heuristic proof-refutation Pomerance below. However, in practice, no one pseudosimp not found, so we can assume conditional second assumption correct.

Implementation of the algorithms in this article

All the algorithms in this paper will be implemented in C ++. All programs were tested only on the compiler Microsoft C ++ 8.0 SP1 (2005), shou compile on g ++.

The algorithms are implemented using templates (templates), which allows to use them as a built-in numeric types, as well as its own class that i the long arithmetic. [Long long arithmetic is not included in the article - TODO]

In the article itself will be given only the most essential functions, the texts of the same auxiliary functions can be downloaded in the appendix to tl Here we give only the headers of these functions, together with a commentary:

```
//! Module 64-bit number
Long Long abs (Long Long n);
```

```
unsigned long long abs (unsigned long long n);

//! Returns true, if n is even
template <class T>
bool even (const T & n);

//! Divides into 2
template <class T>
void bisect (T & n);

//! Multiplies the number by 2
template <class T>
void redouble (T & n);

//! Returns true, if n - the exact square of a prime number
template <class T>
bool perfect_square (const T & n);

//! Calculates the root of a number, rounding it down
template <class T>
T sq_root (const T & n);

//! Returns the number of bits including
template <class T>
unsigned bits_in_number (T n);

//! Returns the value of the k-th bit number (bits are numbered from zero)
template <class T>
bool test_bit (const T & n, unsigned K);

//! Multiplies a * = b (mod n)
template <class T>
void mulmod (T & A, T b, const T & n);

//! Computes a ^ k (mod n)
template <class T, class T2>
T powmod (T A, T2 K, const T & n);

//! Puts the number n in the form q * 2 ^ p
template <class T>
void transform_num (T n, T & P, T & q);

//! Euclid's algorithm
template <class T, class T2>
T gcd (const T & A, const T2 & b);

//! Calculates jacobi (a, b) - Jacobi symbol
template <class T>
T jacobi (T A, T b)

//! Calculates pi (b) of the first prime numbers. Returns a vector with simple and pi - pi (b)
template <class T, class T2>
const std :: vector& get_primes (const T & b, T2 & PI);

//! Trivial check n the simplicity, to get over all the divisors of m.
//! Results: 1 - if n is exactly prime, p - it found divisor 0 - if unknown
template <class T, class T2>
T2 prime_div_trivial (const T & n, m T2);
```

Miller-Rabin test

I will not focus on the Miller-Rabin test, as it is described in many sources, including in Russian (for example., See. [5]).

My only comment is that its speed is O (log ^{3} (N)) bit operations and bring a ready implementation of this algorithm:

```
template <class T, class T2>
bool miller_rabin (T n, T2 b)
{

        // First check the trivial cases
        if (n == 2)
                return true;
        if (n <2 || even (n))
                return false;

        // Check that n and b are relatively prime (otherwise it will cause an error)
        // If they are not relatively prime, then n is not a simple, or it is necessary to increa
        if (b <2)
                b = 2;
        for (T g; (g = gcd (n, b))! = 1; ++ b)
                if (n> g)
                        return false;
        // Decompose n-1 = q * 2 ^ p
        T n_1 = n;
        --n_1;
        T p, q;
        transform_num (n_1, p, q);

        // Calculate b ^ q mod n, if it is equal to 1 or n-1, n is prime (or pseudosimple)
        T rem = powmod (T (b), q, n);
        if (rem == 1 || rem == n_1)
                return true;

        // Now compute b ^ 2q, b ^ 4q, ..., b ^ ((n-1) / 2)
        // If any of them is equal to n-1, n is prime (or pseudosimple)
        for (T i = 1; i <p; i ++)
        {
                mulmod (rem, rem, n);
                if (rem == n_1)
                        return true;
        }

        return false;

}
```

Strong test Lucas-Selfridge

Strong test Lucas-Selfridge consists of two parts: the algorithm to calculate the Selfridge some parameter, and strong algorithm Lucas performe parameter.

Algorithm Selfridge

Among the sequences 5, -7, 9, -11, 13, ... to find the first number D, for which J (D, N) = -1 and gcd (D, N) = 1, where J (x, y) - Jacobi symbol.

Selfridge parameters are P = 1 and Q = (1 - D) / 4.

It should be noted that the parameter does not exist for Selfridge properties that are precise squares. Indeed, if the number is a perfect square, t comes to sqrt (N), where it appears that gcd (D, N)> 1, ie, found that the number N is composite.

In addition, Selfridge parameters will be calculated incorrectly for even numbers and units; however, verification of these cases will not be difficul

Thus, **before the start of the algorithm** should check that the number N is odd, greater than 2, and is not a perfect square, otherwise (under pe least one condition), you should immediately exit the algorithm with the result of a "composite".

Finally, we note that if D for some number N is too large, then the algorithm from a computational point of view, would be inapplicable. Although i this has not been noticed (are sufficient 4-byte number), though the probability of this event should not be excluded. However, for example, in the 10^{6}] max (D) = 47, and in the interval [10^{19}; 10^{19} 10^{6}] max (D) = 67. Furthermore, Bailey and Wagstaff 1980 analytically proved that observa Ribenboim, 1995/96, p. 142).

Strong algorithm Lucas

Algorithm parameters are the number of Lucas **D, P and Q** such that D = P^{2} - 4 * Q? 0, and P> 0.

(Easy to see that the parameters calculated by the algorithm Selfridge satisfy these conditions)

Lucas sequence - a sequence U_K V and $_K$, defined as follows:

$$U_0 = 0$$
$$U_1 = 1,$$
$$\mathbf{U_K = PU_{K-1} - QU_{-K\ 2}}$$
$$V_0 = 2$$
$$V_1 = P$$
$$\mathbf{V_K = PV_{K-1} - QV_{K-2}}$$

Further, let M = N - J (D, N).

If N is prime, and gcd (N, Q) = 1, we have:

$$\mathbf{U_M = 0 \pmod N}$$

In particular, when the parameters D, P, Q calculated Selfridge algorithm, we have:

$$U_{N+1} = 0 \pmod N$$

The converse is not true in general. Nevertheless, pseudosimple numbers when the algorithm is not very much on what, in fact, is based algorith

Thus, the **algorithm is to calculate the Lucas U_M and compare it with zero** .

Next, you need to find some way to speed up computation U_K, otherwise, of course, no practical sense in this algorithm would not be.

We have:

$$U_K = (A^K - b^K) / (A - b),$$
$$V_K\ A = {}^K\ b + {}^K\ ,$$

where a and b - different roots of the quadratic equation x^2 - P x + Q = 0.

Now we can prove the following equation is simple:

```
U 2K U = K V K (mod N)
V 2K V = K 2 - 2 Q K (mod N)
```

Now, imagine if M = E 2^T, where E - an odd number, it is easy to obtain:

$U_M = U_E V_E V_{2E} V_{4E} \dots V_{2^{T-2}E} V_{2^{T-1}E} = 0$ (mod N) ,

and at least one of the factors is zero modulo N.

It is understood that **it suffices to calculate U $_E$ V and $_E$**, and all subsequent Multipliers $V_{2E} V_{4E} \dots V_{2^{T-2}E} V_{2^{T-1}E}$ can **already receive fro**

Thus, it is necessary to learn how to quickly calculate U_E and V_E for odd E.

First, consider the following formulas for the addition of members of Lucas sequences:

```
U + J I = (U I V J + U J V I ) / 2 (mod N)
V + J I = (V I V J + DU I U J ) / 2 (mod N)
```

Note that the division is performed in the field (mod N).

These formulas are proved in a very simple, and here their proof is omitted.

Now, having the formulas for addition and doubling the terms of the sequences of Lucas, concepts and ways of calculating the acceleration U_E a

Indeed, consider the binary representation of the number of E. Suppose first result - U_E V and $_E$ - to be, respectively, U_1 and V_1. Walk into all b from younger to older, skipping only the first bit (the initial term of the sequence). For each i-th bit will calculate U_{2^i} and V_{2^i} of the preceding ter doubling formulas. Furthermore, if the current i-th bit equal to one, then we will add to the response current U_{2^i} and $V_{2,^i}$ by using the sum formul end of the algorithm that runs in O (log (E)), we **obtain the desired U_E and V_E**.

If U_E or V_E were zero (mod N), then N is prime number (or pseudosimple). If they are both different from zero, then calculate V_{2E}, V_{4E}, ... V_2 $_E$. If at least one of them is comparable to zero modulo N, the number N is prime (or pseudosimple). Otherwise, the number N is composite.

Discussion of the algorithm Selfridge

Now that we have looked at Lucas algorithm can elaborate on its parameters D, P, Q, one of the ways to obtain and which is the algorithm of Se

Recall the basic requirements for the parameters:

```
P> 0 ,
 D = P 2 - 4 * Q? 0 .
```

Now continue the study of these parameters.

D should not be a perfect square (mod N) .

Indeed, otherwise we get:

D = b^2, hence J (D, N) = 1, P = b + 2, Q = b + 1, here U_{n-1} = (Q^{n-1} - 1) / (Q - 1).

Ie if D - perfect square, then the algorithm Lucas becomes almost ordinary probabilistic test.

One of the best ways to avoid this - **to require that J (D, N) = -1** .

For example, it is possible to select the first sequence number D of 5, -7, 9, -11, 13, ... for which J (D, N) = -1. Also, let P = 1. Then Q = (1 - D) / 4 method was proposed Selfridge.

However, there are other methods of selecting D. possible to select from a sequence of 5, 9, 13, 17, 21, ... Also, let P - smallest odd, privoskhoc sqrt (D). Then Q = (P^2 - D) / 4.

It is clear that the choice of a particular method of calculating the parameters depends Lucas and its result - pseudosimple may vary for different parameter selection. As shown, the algorithm proposed by Selfridge, was very successful all pseudosimple Lucas-Selfridge are not pseudosim Rabin, at least, no counterexample has been found.

The implementation of the algorithm strong Lucas-Selfridge

Now you only have to implement the algorithm:

```
template <class T, class T2>
bool lucas_selfridge (const T & n, T2 unused)
{

        // First check the trivial cases
        if (n == 2)
                return true;
        if (n <2 || even (n))
                return false;

        // Check that n is not a perfect square, otherwise the algorithm will give an error
        if (perfect_square (n))
                return false;
```

```
// Selfridge algorithm: find the first number d such that:
// Jacobi (d, n) = - 1 and it belongs to the series {5 -7.9, -11.13 ...}
T2 dd;
for (T2 d_abs = 5, d_sign = 1;; d_sign = -d_sign, ++++ d_abs)
{
        dd = d_abs * d_sign;
        T g = gcd (n, d_abs);
        if (1 <g && g <n)
                // Found divider - d_abs
                return false;
        if (jacobi (T (dd), n) == -1)
                break;
}

// Parameters Selfridge
T2
        p = 1,
        q = (p * p - dd) / 4;

// Expand the n + 1 = d * 2 ^ s
T n_1 = n;
++ N_1;
T s, d;
transform_num (n_1, s, d);

// Algorithm Lucas
T
        u = 1,
        v = p,
        u2m = 1,
        v2m = p,
        qm = q,
        qm2 = q * 2,
        qkd = q;
for (unsigned bit = 1, bits = bits_in_number (d); bit <bits; bit ++)
{
        mulmod (u2m, v2m, n);
        mulmod (v2m, v2m, n);
        while (v2m <qm2)
                v2m + = n;
        v2m - = qm2;
        mulmod (qm, qm, n);
        qm2 = qm;
        redouble (qm2);
        if (test_bit (d, bit))
        {
                T t1, t2;
                t1 = u2m;
                mulmod (t1, v, n);
                t2 = v2m;
                mulmod (t2, u, n);

                T t3, t4;
                t3 = v2m;
                mulmod (t3, v, n);
                t4 = u2m;
                mulmod (t4, u, n);
                mulmod (t4, (T) dd, n);

                u = t1 + t2;
                if (! even (u))
                        u + = n;
                bisect (u);
                u% = n;

                v = t3 + t4;
                if (! even (v))
                        v + = n;
                bisect (v);
                v% = n;
                mulmod (qkd, qm, n);
        }
}

// Exactly easy (or pseudo-prime)
if (u == 0 || v == 0)
        return true;

// Dovychislyaem remaining members
T qkd2 = qkd;
redouble (qkd2);
for (T2 r = 1; r <s; ++ r)
{
        mulmod (v, v, n);
        v - = qkd2;
        if (v <0) v + = n;
        if (v <0) v + = n;
```

```
                    if (v> = n) v - = n;
                    if (v> = n) v - = n;
                    if (v == 0)
                            return true;
                    if (r <s-1)
                    {
                            mulmod (qkd, qkd, n);
                            qkd2 = qkd;
                            redouble (qkd2);
                    }
            }

            return false;

    }
```

Code BPSW

It now remains to simply combine the results of all three tests: checking for small trivial divisors, Miller-Rabin test, test strong Lucas-Selfridge.

```
template <class T>
bool baillie_pomerance_selfridge_wagstaff (T n)
{

        // First check for trivial divisors - for example, up to 29
        int div = prime_div_trivial (n, 29);
        if (div == 1)
                return true;
        if (div> 1)
                return false;

        // Miller-Rabin test to the base 2
        if (! miller_rabin (n, 2))
                return false;

        // Strong Lucas-Selfridge test
        return lucas_selfridge (n, 0);

}
```

From here you can download the program (source + exe), containing the full realization of the test BPSW. [77 KB]

Quick implementation

Code length can be significantly reduced at the expense of flexibility, giving up templates and various support functions.

```
const int trivial_limit = 50;
int p [1000];

int gcd (int a, int b) {
        return a? gcd (b% a, a): b;
}

int powmod (int a, int b, int m) {
        int res = 1;
        while (b)
                if (b & 1)
                        res = (res * 1ll * a)% m, --b;
                else
                        a = (a * 1ll * a)% m, b >> = 1;
        return res;
}

bool miller_rabin (int n) {
        int b = 2;
        for (int g; (g = gcd (n, b))! = 1; ++ b)
                if (n> g)
                        return false;
        int p = 0, q = n-1;
        while ((q & 1) == 0)
                ++ P, q >> = 1;
        int rem = powmod (b, q, n);
        if (rem == 1 || rem == n-1)
                return true;
        for (int i = 1; i <p; ++ i) {
                rem = (rem * 1ll * rem)% n;
                if (rem == n-1) return true;
        }
        return false;
}
```

```
int jacobi (int a, int b)
{
        if (a == 0) return 0;
        if (a == 1) return 1;
        if (a <0)
                if ((b & 2) == 0)
                        return jacobi (-a, b);
                else
                        return - jacobi (-a, b);
        int a1 = a, e = 0;
        while ((a1 & 1) == 0)
                a1 >> = 1, ++ e;
        int s;
        if ((e & 1) == 0 || (b & 7) == 1 || (b & 7) == 7)
                s = 1;
        else
                s = -1;
        if ((b & 3) == 3 && (a1 & 3) == 3)
                s = -s;
        if (a1 == 1)
                return s;
        return s * jacobi (b% a1, a1);
}

bool bpsw (int n) {
        if ((int) sqrt (n + 0.0) * (int) sqrt (n + 0.0) == n) return false;
        int dd = 5;
        for (;;) {
                int g = gcd (n, abs (dd));
                if (1 <g && g <n) return false;
                if (jacobi (dd, n) == -1) break;
                dd = dd <0? -dd + 2: -dd-2;
        }
        int p = 1, q = (p * p-dd) / 4;
        int d = n + 1, s = 0;
        while ((d & 1) == 0)
                ++ S, d >> = 1;
        long long u = 1, v = p, u2m = 1, v2m = p, qm = q, qm2 = q * 2, qkd = q;
        for (int mask = 2; mask <= d; mask << = 1) {
                u2m = (u2m * v2m)% n;
                v2m = (v2m * v2m)% n;
                while (v2m <qm2) v2m + = n;
                v2m - = qm2;
                qm = (qm * qm)% n;
                qm2 = qm * 2;
                if (d & mask) {
                        long long t1 = (u2m * v)% n, t2 = (v2m * u)% n,
                                t3 = (v2m * v)% n, t4 = (((u2m * u)% n) * dd)% n;
                        u = t1 + t2;
                        if (u & 1) u + = n;
                        u = (u >> 1)% n;
                        v = t3 + t4;
                        if (v & 1) v + = n;
                        v = (v >> 1)% n;
                        qkd = (qkd * qm)% n;
                }
        }
        if (u == 0 || v == 0) return true;
        long long qkd2 = qkd * 2;
        for (int r = 1; r <s; ++ r) {
                v = (v * v)% n - qkd2;
                if (v <0) v + = n;
                if (v <0) v + = n;
                if (v> = n) v - = n;
                if (v> = n) v - = n;
                if (v == 0) return true;
                if (r <s-1) {
                        qkd = (qkd * 1ll * qkd)% n;
                        qkd2 = qkd * 2;
                }
        }
        return false;
}

bool prime (int n) {// this function should be called to check for ease of
        for (int i = 0; i <trivial_limit && p [i] <n; ++ i)
                if (n% p [i] == 0)
                        return false;
        if (p [trivial_limit-1] * p [trivial_limit-1]> = n)
                return true;
        if (! miller_rabin (n))
                return false;
        return bpsw (n);
}

void prime_init () {// call before the first call prime ()!
        for (int i = 2, j = 0; j <trivial_limit; ++ i) {
                bool pr = true;
                for (int k = 2; k * k <= i; ++ k)
                        if (i% k == 0)
                                pr = false;
                if (pr)
                        p [j ++] = i;
        }
}
```

Heuristic proof-refutation Pomerance

Pomerance in 1984 proposed the following heuristic proof.

Adoption: **Number BPSW-pseudosimple from 1 to X is greater than X $^{1-a}$ for any a> 0** .

Proof.

Let k> 4 - an arbitrary but fixed number. Let T - a large number.

Let P $_K$ (T) - the set of primes p in the interval [T; T ^{K}], for which:

(1) p = 3 (mod 8), J (5, p) = -1

(2) the number (p-1) / 2 is not a perfect square

(3) The number (p-1) / 2 is composed solely of ordinary q <T

(4) the number (p-1) / 2 is composed solely of prime q, such that q = 1 (mod 4)

(5) the number of (p + 1) / 4 is not a perfect square

(6) The number (p + 1) / 4 composed exclusively of common d <T

(7) The number (p + 1) / 4 composed solely of ordinary d, that q = 3 (mod 4)

It is understood that about 8.1 all simple in the interval [T; T ^{K}] satisfies the condition (1). You can also show that the conditions (2) and (5) retain numbers. Heuristically, the conditions (3) and (6) also allows us to leave some of the numbers from the interval (T; T ^{K}). Finally, the event (4) has probability (C (Log T) $^{-1/2}$), as well as an event (7). Thus, the cardinality of the set P $_K$ (T) is prblizitelno at T -> oo

$$\frac{cT^k}{\log^2 T}$$

where c - is a positive constant depending on the choice of k.

Now we **can build a number n** , which is not a perfect square, composed of simple l of P $_K$ (T), where l is odd and less than T ^{2} / Log (T ^{K}). Nu ways to choose a number n is approximately

$$\binom{[cT^k/\log^2 T]}{\ell} > e^{T^2(1-3/k)}$$

for large T and fixed k. Furthermore, each n is a number less than e $^{T^2}$.

Let Q denote the $_1$ product of prime q <T, for which q = 1 (mod 4), and by Q $_3$ - a product of prime q <T, for which q = 3 (mod 4). Then gcd (Q $_1$, Q $_1$ Q $_3$? e ^{T} . Thus, the number of ways to choose n **with the additional conditions**

```
n = 1 (mod Q 1 ), n = -1 (mod Q 3 )
```

must heuristically at least

$$e^{T^2 (1 - 3 / K)} / e^{2T} > \mathbf{e^{T^2 (1 - 4 / k)}}$$

for large T.

But **every such n - is a counterexample to the test BPSW** . Indeed, n is the number of Carmichael (ie, the number on which the Miller-Rabin for any reason), so it will automatically pseudosimple base 2 Since n = 3 (mod 8) and each p | n = 3 (mod 8), it is obvious that n is also a strong pseudosimple Since J (5, n) = -1, then every prime p | n satisfies J (5, p) = -1, and since the p + 1 | n + 1 for any prime p | n, it follows that n - pseu Lucas Lucas for any test with discriminant 5.

Thus, we have shown that for any fixed k and all large T, there will at least e $^{T^2 (1 - 4 / k)}$ counterexamples to test BPSW of numbers less than e ^{T} we put x = e $^{T^2}$, x is at least $^{1 - 4 / k}$ counterexamples smaller x. Since k - a random number, then our evidence indicates that **the number of counterexamples, smaller x, is a number greater than x $^{1-a}$ for any a> 0** .

Practical tests test BPSW

In this section we will consider the results obtained as a result of me testing my test implementation BPSW. All tests were carried out on the inte including 64-bit long long. Long arithmetic has not been tested.

Testing was conducted on a computer with a processor Celeron 1.3 GHz.

All times are given in **microseconds** (10 $^{-6}$ s).

The average operating time on the segment number, depending on the limit of the trivial enumeration

This refers to the parameter passed to the function prime_div_trivial (), which in the above code is 29.

Download a test program (source code and exe-file). [83 KB]

If you run a test **on all the odd numbers** in the interval, the results turn out to be:

the beginning of the segment	End segments	limit> iterate>	0	10^2	10^3	10^4	
1	10^5		8.1	4.5	0.7	0.7	
10^6	10^6 10^5		12.8	6.8	7.0	1.6	
10^9	10^9 10^5		28.4	12.6	12.1	17.0	
10^{12}	10^{12} 10^5		41.5	16.5	15.3	19.4	
10^{15}	10^{15} 10^5		66.7	24.4	21.1	24.8	

If the test is run **only on the primes** in the interval, the rate of work is as follows:

the beginning of the segment	End segments	limit> iterate>	0	10^2	10^3	10^4	
1	10^5		42.9	40.8	3.1	4.2	
10^6	10^6 10^5		75.0	76.4	88.8	13.9	
10^9	10^9 10^5		186.5	188.5	201.0	294.3	2
10^{12}	10^{12} 10^5		288.3	288.3	302.2	387.9	1
10^{15}	10^{15} 10^5		485.6	489.1	496.3	585.4	1

Thus, optimally choose **the trivial limit busting at 100 or 1000** .

For all these tests, I chose the limit in 1000.

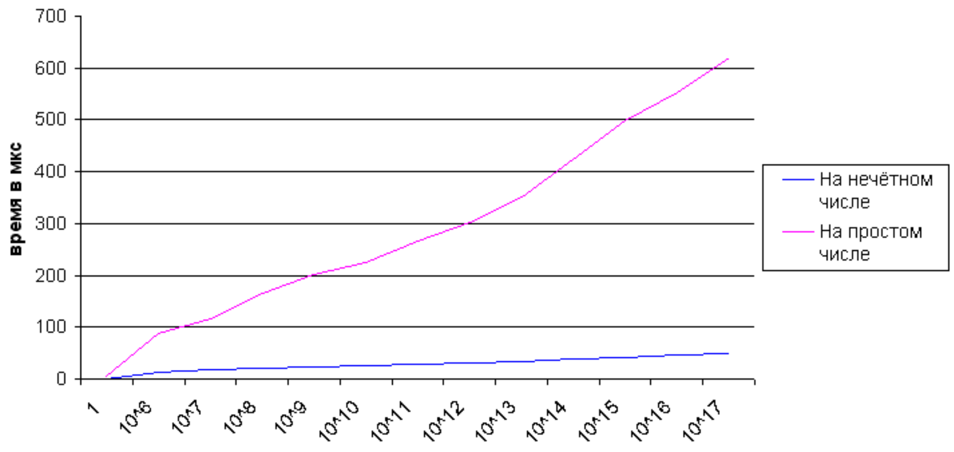
The average operating time on the segment number

Now, when we chose the limit of trivial enumeration, you can more accurately test the speed at various intervals.

Download a test program (source code and exe-file). [83 KB]

the beginning of the segment	End segments	while working on the odd numbers	time work on prime numbers
1	10^5	1.2	4.2
10^6	10^6 10^5	13.8	88.8
10^7	10^7 10^5	16.8	115.5
10^8	10^8 10^5	21.2	164.8
10^9	10^9 10^5	24.0	201.0
10^{10}	10^{10} 10^5	25.2	225.5
10^{11}	10^{11} 10^5	28.4	266.5
10^{12}	10^{12} 10^5	30.4	302.2
10^{13}	10^{13} 10^5	33.0	352.2
10^{14}	10^{14} 10^5	37.5	424.3
10^{15}	10^{15} 10^5	42.3	499.8
10^{16}	10^{15} 10^5	46.5	553.6
10^{17}	10^{15} 10^5	48.9	621.1

Or, in the form of a graph, the approximate time of the test on one BPSW including:



That is, we have found that in practice, a small number (10 ^{17}), **the algorithm runs in O (Log N)** . This is due to the fact that the embedded typ division operation is performed in O (1), i.e. dividing complexity zavisisit not the number of bits in number.

If we apply the test to a long BPSW arithmetic, it is expected that it will work just for the O (log ^{3} (N)). [TODO]

Appendix. All programs

Download all the programs in this article. [242 KB]

Literature

Usable me literature, is available online:

1. Robert Baillie; Samuel S. Wagstaff **Lucas pseudoprimes** Math. Comp. 35 (1980) 1391-1417 mpqs.free.fr/LucasPseudoprimes.pdf

2. Daniel J. Bernstein **Distinguishing Prime numbers from Composite numbers: the State of the art in 2004** Math. Comp. (2004) cr.yp.to/primetests/prime2004-20041223.pdf

3. Richard P. Brent **Primality Testing and Integer factorisation** The Role of Mathematics in Science (1990) wwwmaths.anu.edu.au/~brent/pd/rpb120.pdf

4. H. Cohen; HW Lenstra **Primality Testing and Jacobi Sums** Amsterdam (1984) www.openaccess.leidenuniv.nl/bitstream/1887/2136/1/3

5. Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest **Introduction to Algorithms** [without reference] The MIT Press (2001)

6. M. Martin **PRIMO - Primality Proving** www.ellipsa.net

7. F. Morain **Elliptic curves and Primality proving** Math. Comp. 61 (203)

8. Carl Pomerance **Are there Counter-examples to the Baillie-PSW Primality test?** Math. Comp. (1984) www.pseudoprime.com/dopo.p

9. Eric W. Weisstein **Baillie-PSW Primality test** MathWorld (2005) mathworld.wolfram.com/Baillie-PSWPrimalityTest.html

10. Eric W. Weisstein **Strong Lucas pseudoprime** MathWorld (2005) mathworld.wolfram.com/StrongLucasPseudoprime.html

11. Paulo Ribenboim **The Book of Prime Number Records** Springer-Verlag (1989) [no link]

List of recommended books, which I could not find on the Internet:

12. Zhaiyu Mo; James P. Jones **A New Primality test using Lucas sequences** Preprint (1997)

13. Hans Riesel **Prime numbers and computer Methods for Factorization** Boston: Birkhauser (1994)

2 Комментариев e-maxx

Лучшее вначале ▾ Поделиться

Присоединиться к обсуждению...

BotanIQ • 2 года назад
Последняя таблица
Столбец "конец отрезка"
Две нижних строки
Указаны отрезки [10^16; 10^15+10^5] и [10^17; 10^15+10^5]
Следует исправить 10^15+10^5 в этих строках на 10^16+10^5 и 10^17+10^5 соответственно
3 • Ответить • Поделиться ›

Denis Sherstennikov • год назад
Внимание! У вас в методе get_primes возвращается ссылка на вектор, выделенный локально на стеке! Это ОЧЕНЬ плохо! (почитайте
если ещё не читали)

Суть в том, что при однопоточном исполнении вектор в стеке доживает (как - точно не знаю) до места считывания. При распараллели
потока - уже нет, видимо из-за Context Switch. Получается, что в prime_div_trivial вы итерируетесь по пустому вектору, что роняет прог

До распараллеливания всё работало как надо, после - потратил 3 часа на отладку.

А помимо этого - спасибо большое за код! Правда, пришлось юзать C++/CLI чтобы дотянуть его до C#.

Надеюсь, что вы исправите код. Я пока просто убрал ссылку и передаю по значения, это дикий отладочный оверхед, так что можно п
таблицу хранить с первыми X простыми числами - хоть хешмапом.
1 • Ответить • Поделиться ›