BPSW test for primality

Introduction

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

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

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

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

Brief description

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

1. Run the Miller-Rabin test with base 2.

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

3. Redo "prime" only when both tests have returned "prime".

+0. Furthermore, the algorithm can start to add a check for trivial divisors, say, 1000. This will increase the speed of operation on a composite number, however, has slowed somewhat simple algorithm.

Thus BPSW algorithm based on the following:

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

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

In fact, the second assumption seems to be as wrong - heuristic proof-refutation Pomerance below. Nevertheless, in practice, no one pseudosimple still have not found, so we can assume conditional second assumption is true.

Implementation of 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), should also compile on g+ +.

Algorithms are implemented using templates (templates), which allows their use as a built-in numeric types, and own class that implements the long arithmetic.[Until long arithmetic is not included in the article - TODO]

In the article itself will be given only the most essential functions, texts as auxiliary functions can be downloaded in the appendix to this article. Here, we present only the headers of these functions along with the comments:

```
/ /! 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 the number by 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 k-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);

/ /! Converts 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);

/ /! N trivial check on simplicity, all the divisors are moving to m.
/ /! Results: 1 - if exactly n prime, p - found its 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 (eg, see [5]).

My only comment is that the speed of his work has O (log ^{3} (N)) bit operations and bring the finished implementation of this algorithm:

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

	/ / First check for 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 fail)
	/ / If they are not relatively prime, then either n is not easy, you need to increase
or b
	if (b <2)
		b = 2;
	for (T g; (g = gcd (n, b))! = 1; + + b)
		if (n> g)
			return false;

	/ / Expand the 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 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 Lucas-Selfridge test consists of two parts: Selfridge algorithm for calculating a parameter, and a strong algorithm Lucas performed with this parameter.

Algorithm Selfridge

Among the sequences 5, -7, 9, -11, 13, ... a first number of D, such that 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 Selfridge properties that are precise squares. Indeed, if the number is a perfect square, then bust D comes to sqrt (N), where it appears that gcd (D, N)> 1, ie found that the number N is composite.

In addition, the parameters will be calculated incorrectly Selfridge for even numbers and units; however, inspection of these cases is not difficult.

Thus, **before the algorithm** should check that the number N is an odd number greater than 2, and is not a perfect square, otherwise (under penalty of at least one condition) should immediately withdraw from the algorithm with the result of "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 in practice this was not observed (exerted enough 4-byte number), however the likelihood of this event should not be excluded. However, for example, in the interval [1, 10^6] max (D) = 47, and in the interval [10^{19}, $10^{19,}$ 10^6] max (D) = 67. In addition, Bailey and Vagstaf in 1980 analytically proved this observation (see Ribenboim, 1995/96, page 142).

Strong algorithm Lucas

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

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

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

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

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

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

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

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

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

The converse is not generally true. Nevertheless, pseudosimple numbers in this algorithm is not very much on what, in fact, is based algorithm Lucas.

Thus, **the algorithm is calculating Lucas U_M and comparing it to zero** .

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

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:

$U_{2K} = U_K V_K \pmod N$
$V_{2K} V = {}_K{}^2 - 2 Q^K \pmod N$

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

$U_M = U_E V_E V_{2E} V_{4E} \ldots V_2{}^{T-2}{}_E V_2{}^{T-1}{}_E = 0 \pmod N$,

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

It is clear that **it suffices to compute U_E and V_E** , and all subsequent factors $V_{2E} V_{4E} \ldots V_2{}^{T-2}{}_E V_2{}^{T-1}{}_E$ can **already receive from them** .

Thus, it remains to learn quickly calculate U_E and V_E for odd E.

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

$U_{+JI} = (U_I V_J + U_J V_I) / 2 \pmod N$
$V_{+JI} = (V_I V_J + DU_I U_J) / 2 \pmod N$

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

These formulas are proved very simple and here are their proofs are omitted.

Now, having the formulas for addition and doubling sequences members Lucas and understandable way to accelerate computing U_E and V_E .

Indeed, consider the binary representation of the number of E. We first result - $U_E V$ and ${}_E$ - equal, respectively, U_1 and V_1 . Go through the bits of E 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 previous members using doubling formulas. Furthermore, if the current i-th bit is one, that will add to the response current $U_2{}^i$ and $V_2{}^i$ using addition formulas. At the end of the algorithm runs in O (log (E)), we **obtain the desired U_E and V_E** .

If the U_E , or V_E were zero (mod N), N is a prime number (or pseudosimple). If they are both different from zero, then compute V_{2E} , V_{4E} , ... $V_2{}^{2T-}{}_E$, $V_2{}^{T-1}{}_E$. 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, we can elaborate on its parameters D, P, Q, one of the ways upon which the algorithm is Selfridge.

Recall the basic requirements for the parameters:

P> 0 ,
D = P ^{2} - 4 * Q? 0 .

Now we 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 becomes almost Lucas conventional probabilistic test.

One of the best ways to avoid this - **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 suppose P = 1. Then Q = (1 - D) / 4. This method was proposed Selfridge.

However, there are other methods of selecting D. It is possible to select a sequence of 5, 9, 13, 17, 21, ... Also, let P - the smallest odd, privoskhodyaschee 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 methods of parameter selection. As shown, the algorithm proposed by Selfridge, was very successful all pseudosimple Lucas-Selfridge are not pseudosimple Miller-Rabin, at least, no counterexample was found.

Implementing a strong algorithm 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 for 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 would produce
an error
```

```
if (perfect_square (n))
	return false;

/ / Algorithm Selfridge: find the first number d such that:
/ / Jacobi (d, n) = -1, and it belongs to the range of {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)
		/ / Find divisor - 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);
		}
}

/ / Just a simple (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

Now it remains to simply combine the results of all three tests: checking for small trivial divisors, the 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 with base 2
	if (! miller_rabin (n, 2))
		return false;

	/ / Strong test Lucas-Selfridge
	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, abandoning 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 must be called to check for simplicity
        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 () {/ / called before the first call to 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 more than X X^{1-a} for all a> 0** .

Proof.

Let k> 4 - arbitrary but fixed number. Let T - some big 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 entirely of such prime q, q = 1 that (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 1/8 of all prime in the interval $[T; T^K]$ satisfies the condition (1). Also it can be shown that the conditions (2) and (5) retain a certain part number. Heuristically, the conditions (3) and (6) also let us leave some of the numbers from the interval $(T; T^K)$. Finally, the event (4) has a probability $(C (\text{Log } T)^{-1/2})$, 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 - 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 $T^2 / \text{Log } (T^K)$. Number of 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. Further, each n is a number less than e^{T^2}.

Let Q_1 product of prime q <T, for which q = 1 (mod 4), and by Q_3 - a product of primes q <T, for which q = 3 (mod 4). Then gcd $(Q_1, Q_3) = 1$ and $Q_1 Q_3 ? e^T$ .Thus, the number of ways to choose n **with the additional conditions**

$n = 1 \pmod{Q_1}$, $n = -1 \pmod{Q_3}$

be heuristically at least

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

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 test is wrong 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 will pseudosimple strong base 2.Since J (5, n) = -1, then every prime p | n satisfies J (5, p) = -1, and since p +1 | n +1 for every prime p | n, it follows that n - pseudosimple Lucas Lucas for any test with discriminant 5.

Thus, we have shown that for any fixed k and all large T, is at least $e^{T 2 (1 - 4 / k)}$ counterexamples to test BPSW of numbers less than $e^{T 2}$. Now, if we put $x = e^{T2}$, 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 all a> 0** .

Practical Test Test BPSW

This section will discuss the results obtained by testing me my test implementation BPSW. All tests were carried out on the internal type - including 64-bit long long. Long arithmetic untested.

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

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

Average time on the segment number depending on the trivial limit busting

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** from the interval, the results turn out to be:

the beginning of the segment	end segment	limit> iterate>	0	10^2	10^3	10^4	10^5
1	10^5		8.1	4.5	0.7	0.7	0.9

10^6	10^6 10^5		12.8	6.8	7.0	1.6	1.6
10^9	10^9 10^5		28.4	12.6	12.1	17.0	17.1
10^{12}	10^{12} 10^5		41.5	16.5	15.3	19.4	54.4
10^{15}	10^{15} 10^5		66.7	24.4	21.1	24.8	58.9

If the test is run **only on prime numbers** in the interval, the speed is as follows:

the beginning of the segment	end segment	limit> iterate>	0	10^2	10^3	10^4	10^5
1	10^5		42.9	40.8	3.1	4.2	4.2
10^6	10^6 10^5		75.0	76.4	88.8	13.9	15.2
10^9	10^9 10^5		186.5	188.5	201.0	294.3	283.9
10^{12}	10^{12} 10^5		288.3	288.3	302.2	387.9	1069.5
10^{15}	10^{15} 10^5		485.6	489.1	496.3	585.4	1267.4

Thus, the optimal filter **limit trivial enumeration of 100 or 1000** .

For all these tests, I chose the 1000 limit.

Average time on the segment number

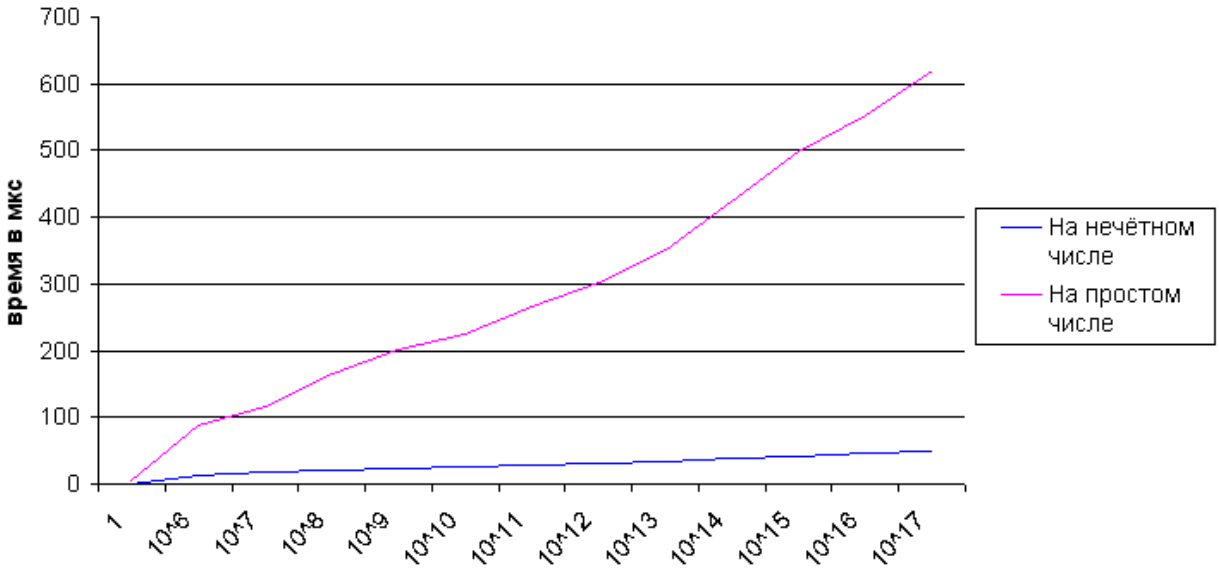
Now that we have chosen the trivial limit busting, can more accurately test the speed at various intervals.

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

the beginning of the segment	end segment	while working at 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 to ^{17}), **the algorithm runs in O (Log N)** . This is because the built-in type for int64 division operation is performed in O (1), ie complexity of fission zavisisit the number of bits in the 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 from Prime numbers 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/1887/2136/1/346_065.pdf
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.pdf
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:

1. Zhaiyu Mo; James P. Jones **A New Primality test using Lucas sequences** Preprint (1997)
2. Hans Riesel **Prime numbers and Computer Methods for Factorization** Boston: Birkhauser (1994)