1.Euler function

Definition

Euler function $\phi(n)$ (sometimes denoted $\varphi(n)$or $phi(n)$) - the number of properties $1$to $n$prime to n. In other words, the quantity of such properties in a segment $[1;n]$, the greatest common divisor of which is to $n$unity.

The first few values of this function (A000010 in OEIS encyclopedia):

$$\phi(1) = 1,$$
$$\phi(2) = 1,$$
$$\phi(3) = 2,$$
$$\phi(4) = 2,$$
$$\phi(5) = 4.$$

Properties

The following three simple properties of the Euler - enough to learn how to calculate it for any number:

- If p- a prime, then $\phi(p) = p - 1$.

  (This is obvious, because any number, except for the $p$relatively easy with him.)

- If p- simple a- a natural number, then $\phi(p^a) = p^a - p^{a-1}$.

  (Since the number of p^anot only relatively prime numbers of the form , which pieces.)$pk\ (k \in \mathcal{N})$ $p^a/p = p^{a-1}$

- If $a$and $b$are relatively prime, then $\phi(ab) = \phi(a)\phi(b)$("multiplicative" Euler function).

  (This follows from the Chinese remainder theorem . Consider an arbitrary number $z \leq ab$. denote x and $y$the remnants of the division $z$at $a$and b, respectively. then $z$coprime abif and only if $z$is prime to $a$and $b$separately, or, equivalently, xa one- simply $a$and $y$relatively prime to b. Applying the Chinese remainder theorem, we see that any pair of numbers $x$and the number of one-to-one correspondence , which completes the proof.)$y\ (x \leq a,\ y \leq b) z\ (z \leq ab)$

  From here you can get the Euler function for all $n$through its **factorization** (decomposition $n$into prime factors):

  if

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$$

  (Where all p_i- common), then

$$\phi(n) = \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \cdot \ldots \cdot \phi(p_k^{a_k}) =$$
$$= (p_1^{a_1} - p_1^{a_1-1}) \cdot (p_2^{a_2} - p_2^{a_2-1}) \cdot \ldots \cdot (p_k^{a_k} - p_k^{a_k-1}) =$$
$$= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_k}\right).$$

Implementation

The simplest code that computes the Euler function, factoring in the number of elementary method $O(\sqrt{n})$:

```
int phi (int n) {
	int result = n;
	for (int i=2; i*i<=n; ++i)
		if (n % i == 0) {
			while (n % i == 0)
				n /= i;
			result -= result / i;
		}
	if (n > 1)
		result -= result / n;
	return result;
}
```

The key place for the calculation of the Euler function - is to find the **factorization of** the number n. It can be done in a time much smaller $O(\sqrt{n})$: see. Efficient algorithms for factorization .

Applications of Euler's function

The most famous and important property of Euler's function is expressed in **Euler's theorem** :

$$a^{\phi(m)} \equiv 1 \pmod{m},$$

where $a$and $m$are relatively prime.

In the particular case when ma simple Euler's theorem turns into the so-called **Fermat's little theorem** :

$$a^{m-1} \equiv 1 \pmod{m}$$

Euler's theorem occurs quite often in practical applications, for example, see. inverse element in the modulo .

Problem in online judges

A list of tasks that require the function to calculate the Euler or use Euler's theorem, or meaningfully Euler function to restore the original number:

- UVA # 10179 **"Irreducible Basic Fractions"** [Difficulty: Easy]
- UVA # 10299 **"Relatives"** [Difficulty: Easy]
- UVA # 11327 **"Enumerating Rational Numbers"** [Difficulty: Medium]
- TIMUS # 1673 **"tolerance for the exam"** [Difficulty: High]

2.Binary exponentiation

Binary (binary) exponentiation - is a te chnique that allows you to build any number $n$of th degree of the $O(\log n)$ multiplications (instead of n the usual approach of multiplications).

Moreover, the technique described here is applicable to any **of the associative** operation, and not only to the multiplication number. Recall operation is called associative if for any a, b, cexecuted:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

The most obvious generalization - on residues modulo some (obviously, associativity is maintained). The next "popularity" is a generalization to the matrix product (it is well-known associativity).

Algorithm

Note that for any number $a$and **an even** number of $n$feasible obvious identity (which follows from the associativity of multiplication):

$$a^n = (a^{n/2})^2 = a^{n/2} \cdot a^{n/2}$$

It is the main method in the binary exponentiation. Indeed, for even $n$we have shown how, by spending only one multiplication, one can reduce the problem to less than half the power.

It remains to understand what to do if the degree **is odd** . Here we do is very simple: move on to the power that will have an even:$n\ n - 1$

$$a^n = a^{n-1} \cdot a$$

So, we actually found a recursive formula on the degree $n$we go, if it even, to $n/2$, and otherwise - to $n - 1$. understandable that there will be no more $2 \log n$hits before we come to $n = 0$(the base of the recursion formula). Thus, we have an algorithm that works for $O(\log n)$multiplications.

Implementation

A simple recursive implementation:

```
int binpow (int a, int n) {
	if (n == 0)
		return 1;
	if (n % 2 == 1)
		return binpow (a, n-1) * a;
	else {
		int b = binpow (a, n/2);
		return b * b;
	}
}
```

Non-recursive implementation, also optimized (division by 2 replaced bit operations):

```
int binpow (int a, int n) {
	int res = 1;
	while (n)
		if (n & 1) {
			res *= a;
			--n;
		}
```

```
                else {
                        a *= a;
                        n >>= 1;
                }
        return res;
}
```

This implementation is still somewhat simplified by noting that the construction of $a$the square is always performed, regardless of whether the condition is odd worked $n$or not:

```
int binpow (int a, int n) {
        int res = 1;
        while (n) {
                if (n & 1)
                        res *= a;
                a *= a;
                n >>= 1;
        }
        return res;
}
```

Finally, it is worth noting that the binary exponentiation already implemented by Java, but only for a class of long arithmetic BigInteger (pow function in this class works under binary exponentiation algorithm).

Examples of solving problems

Efficient calculation of Fibonacci numbers

Condition . Given a number n. Required to calculate $F_n$where F_i- the Fibonacci sequence .

Decision . More detail the decision described in the article on the Fibonacci sequence . Here we only briefly we present the essence of the decision.

The basic idea is as follows. Calculation of the next Fibonacci number is based on the knowledge of the previous two Fibonacci numbers, namely, each successive Fibonacci number is the sum of the previous two. This means that we can construct a matrix 2×2that will fit this transformation: as the two Fibonacci numbers $F_i$and F_{i+1}calculate the next number, ie pass to the pair F_{i+1}, F_{i+2}. For example, applying this transformation $n$to a couple times $F_0$and F_1we get a couple $F_n$and F_{n+1} . Thus, elevating the matrix of this transformation in $n$the power of th, we thus find the required F_n time for $O(\log n)$what we required.

Erection of changes in the $k$degree of th

Condition . Given permutation of $p$length n. Required to build it in $k$the power of th, ie find what happens if to the identity permutation $k$permutation times apply p.

Decision . Just apply to the interchange of $p$the algorithm described above binary exponentiation. No differences compared with the construction of the power of numbers - no. Solution is obtained with the asymptotic behavior $O(n \log k)$.

(Note. This problem can be solved more efficiently, **in linear time** . To do this, just select all the cycles in the permutation, and then to consider separately each cycle, and taking $k$modulo the length of the current cycle, to find an answer for this cycle.)

The rapid deployment of a set of geometric operations to points

Condition . Given $n$points $p_i$and are $m$transformations that must be applied to each of these points. Each transformation - a shift or a predetermined vector, or scaling (multiplication coefficients set of coordinates) or a rotation around a predetermined axis at a predetermined angle. In addition, there is a composite operation is cyclical repetition: it has a kind of "repeat a specified number of times specified by the list of transformations" (cyclic repetition of operations can be nested).

Required to compute the result of applying these operations to all points (effectively, ie in less than what $O(n \cdot length)$, where $length$- the total number of operations that must be done).

Decision . Look at the different types of transformations in terms of how they change location:

- Shift operation - it just adds to all the coordinates of the unit, multiplied by some constant.
- Zoom operation - it multiplies each coordinate by a constant.
- Operation rotational axis - it can be represented as follows: new received coordinates can be written as a linear combination of the old.

(We will not specify how this is done. Example, you can submit it for simplicity as a combination of five-dimensional rotations: first, in the planes OXY , and OXZso that the axis of rotation coincides with the positive direction of the axis OX, then the desired rotation around an axis in the plane YZ, then reverse rotations in planes $OXZ$and OXY so that the axis of rotation back to its starting position.)

Easy to see that each of these transformations - a recalculation of coordinates on linear formulas. Thus, any such transformation can be written in matrix form 4×4:

$$\begin{pmatrix} a_1 1 & a_{12} & a_{13} & a_{14} \\ a_2 1 & a_{22} & a_{23} & a_{24} \\ a_3 1 & a_{32} & a_{33} & a_{34} \\ a_4 1 & a_{42} & a_{43} & a_{44} \end{pmatrix},$$

which, when multiplied (left) to the line of the old coordinates and constant unit gives a string of new coordinates and constants units:

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} x' & y' & z' & 1 \end{pmatrix}.$$

(Why it took to enter a dummy fourth coordinate that is always equal to one? Without this we would have to implement the shift operation: after the shift - it is just adding to the coordinates of the unit, multiplied by certain coefficients. Without dummy units we would be able to implement only linear combinations of the coordinates themselves, and add thereto predetermined constant - could not.)

Now the solution of the problem becomes almost trivial. Times each elementary operation described by the matrix, then the process described by the product of these matrices, and the operation of a cyclic repetition - the erection of this matrix to a power. Thus, during the time we $O(m \cdot \log repetition)$can predposchitat matrix 4×4that describes all the changes, and then just multiply each point $p_i$on the matrix - thus, we will answer all questions in time $O(n)$.

Number of tracks fixed length column

Condition . Given an undirected graph $G$with $n$vertices, and given a number k. Required for each pair of vertices i, and $j$find the number of paths between them containing exactly $k$the edges.

Decision . More details on this problem is considered in a separate article . Here, we only recall the essence of this solution: we simply erect a $k$degree of th adjacency matrix of the graph, and the matrix elements, and will be the solutions. Total asymptotics - $O(n^3 \log k)$.

(Note. In the same article, and the other is considered a variation of this problem: when the graph is weighted, and you want to find the path of minimum weight, containing exactly $k$the edges. As shown in this paper, this problem is also solved by using binary exponentiation of the adjacency matrix of the graph, but instead of the usual operation of multiplication of two matrices should be used modified: instead of multiplying the amount taken, and instead of summing - Take a minimum.)

Variation binary exponentiation: the multiplication of two numbers modulo

We give here an interesting variation of the binary exponentiation.

Suppose we are faced with such a **task** : to multiply two numbers $a$and $b$modulo m:

$$a \cdot b \pmod m$$

Assume that the numbers can be quite large: so that the numbers themselves are placed in a built-in data types, but their direct product $a \cdot b$- no longer exists (note that we also need to sum numbers placed in a built-in data type). Accordingly, the task is to find the desired value $(a \cdot b) \pmod m$, without the aid of a long arithmetic .

The decision is as follows. We simply apply the binary exponentiation algorithm described above, but instead of multiplication, we will make the addition. In other words, the multiplication of two numbers, we have reduced to $O(\log m)$the operations of addition and multiplication by two (which is also, in fact, is the addition).

(Note. This problem can be solved **in a different way** , by resorting to assistance operations with floating-point numbers. Namely, to calculate the floating point expression $a \cdot b/m$, and round it to the nearest integer. So we find **an approximate** quotient. Subtracting from his works $a \cdot b$(ignoring overflow), we are likely to get a relatively small number, which can be taken modulo m- and return it as an answer. This solution looks pretty unreliable, but it is very fast, and very briefly, is realized.)

Problem in online judges

List of tasks that can be solved using binary exponentiation:

- SGU # 265 **"Wizards"** [Difficulty: Medium]

3.Euclid's algorithm find the GCD (greatest common divisor)

Given two non-negative integers a and b. Required to find their greatest common divisor, ie the largest number that divides both a a and b. English "greatest common divisor" is spelled "greatest common divisor", and its common designation is $\gcd$:

$$\gcd(a, b) = \max_{k=1\ldots\infty\ :\ k|a\ \&\ k|b} k$$

(Here the symbol "$|$"denotes the divisibility, ie," $k|a$"means" $k$divide a")

When it is of the numbers is zero, and the other is non-zero, their greatest common divisor, by definition, it will be the second number. When the two numbers are equal to zero, the result is undefined (any suitable infinite number), we put in this case, the greatest common divisor of zero. Therefore, we can speak of such a rule: if one of the numbers is zero, the greatest common divisor equal to the second number.

Euclid's algorithm , discussed below, solves the problem of finding the greatest common divisor of two numbers $a$and $b$for $O(\log \min(a, b))$.

This algorithm was first described in the book of Euclid's "Elements" (around 300 BC), although it is possible, this algorithm has an earlier origin.

Algorithm

The algorithm itself is very simple and is described by the following formula:

$$\gcd(a, b) = \begin{cases} a, & \text{if b=0} \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

Implementation

```
int gcd (int a, int b) {
	if (b == 0)
		return a;
	else
		return gcd (b, a % b);
}
```

Using the ternary conditional operator C ++, the algorithm can be written even shorter:

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

Finally, we present an algorithm and a non-recursive form:

```
int gcd (int a, int b) {
	while (b) {
		a %= b;
		swap (a, b);
	}
	return a;
}
```

Proof of correctness

First, note that at each iteration of the Euclidean algorithm its second argument is strictly decreasing, therefore, since it is non-negative, then the Euclidean algorithm **always terminates** .

To **prove the correctness** , we need to show that $\gcd(a, b) = \gcd(b, a \bmod b)$for any $a \geq 0, b > 0$.

We show that the quantity on the left-hand side is divided by the present on the right and the right-hand - is divided into standing on the left. Obviously, this would mean that the left and right sides of the same, and that proves the correctness of Euclid's algorithm.

Denote $d = \gcd(a, b)$. Then, by definition, $d|a$and $d|b$.

Next, we expand the remainder of the division $a$on $b$through their private:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

But then it follows:

$$d \mid (a \bmod b)$$

So, remembering the statement $d|b$, we obtain the system:

$$\begin{cases} d \mid b, \\ d \mid (a \bmod b) \end{cases}$$

We now use the following simple fact: if for any three numbers p, q, rperformed: $p|q$and $p|r$then executed and: $p \mid \gcd(q, r)$. In our situation, we obtain:

$$d \mid \gcd(b, a \bmod b)$$

Or by substituting $d$its definition as $\gcd(a, b)$we obtain:

$$\gcd(a, b) \mid \gcd(b, a \bmod b)$$

So, we spent half of the proof: show that the left-right divide. The second half of the proof is similar.

Working time

Time of the algorithm is evaluated **Lame theorem** which establishes a surprising connection of the Euclidean algorithm and the Fibonacci sequence:

If $a > b \geq 1$and $b < F_n$for some n, the Euclidean algorithm performs no more $n - 2$recursive calls.

Moreover, it can be shown that the upper bound of this theorem - optimal. When $a = F_n, b = F_{n-1}$it will be done $n - 2$recursive call. In other words, **successive Fibonacci numbers - the worst input** for Euclid's algorithm.

Given that the Fibonacci numbers grow exponentially (as a constant in power n), we find that the Euclidean algorithm runs in $O(\log \min(a, b))$multiplication operations.

LCM (least common multiple)

Calculation of the least common multiple (least common multiplier, lcm) reduces to the calculation $\gcd$the following simple statement:

$$\text{lcm}(a,b) = \frac{a \cdot b}{\gcd(a,b)}$$

Thus, the calculation of the NOC can also be done using the Euclidean algorithm, with the same asymptotic behavior:

```
int lcm (int a, int b) {
	return a / gcd (a, b) * b;
}
```

(Here is advantageous first divided into $\gcd$, and only then is multiplied by b, as this will help to avoid overflows in some cases)

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Design and Analysis** [2005]

4.Sieve of Eratosthenes

Sieve of Eratosthenes - an algorithm for finding all prime numbers in the interval $[1;n]$for the $O(n \log \log n)$operations.

The idea is simple - write a series of numbers $1 \dots n$, and will strike out again all numbers divisible by 2, except for the numbers 2, then dividing by 3, except for the number 3, then on 5, then 7, $11$and all other simple to n.

Implementation

Immediately we present implementation of the algorithm:

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
	if (prime[i])
		if (i * 1ll * i <= n)
			for (int j=i*i; j<=n; j+=i)
				prime[j] = false;
```

This code first checks all numbers except zero and one, as simple, and then begins the process of sifting composite numbers. To do this, we loop through all the numbers from2before n, and if the current number $i$is simple, then mark all the numbers that are multiples of him as a constituent.

At the same time we are beginning to go from i^2, as all lower multiples i, be sure to have a prime divisor less i, which means that they have already been screened before.(But as i^2can easily overwhelm type intin the code before the second nested loop is an additional check using the type $long\ long$.)

With this implementation, the algorithm consumes $O(n)$memory (obviously) and performs the $O(n \log \log n)$action (this is proved in the next section).

Asymptotics

We prove that the asymptotic behavior of the algorithm is $O(n \log \log n)$.

So, for each prime $p \leq n$will run the inner loop, which make $\frac{n}{p}$actions. Therefore, we need to estimate the following value:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{n}{p} = n \cdot \sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p}.$$

Let us recall here two known facts: that the number of primes less than or equal to $n$approximately equal $\frac{n}{\ln n}$, and that $k$flashover prime approximately equal $k \ln k$(this follows from the first statement). Then the sum can be written as follows:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}.$$

Here we have identified the first prime of the sum, since the $k = 1$approximation of the $k \ln k$turn 0, that will lead to division by zero.

We now estimate a sum by the integral of the same function on $k$from 2before $\frac{n}{\ln n}$(we can produce such an approximation, because, in fact, refers to the sum of the integral as its approximation by the formula of rectangles):

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk.$$

Primitive of the integrand there $\ln \ln k$. Performing substitution and removing members of the lower order, we get:

$$\int_{2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}\, dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n.$$

Now, returning to the initial sum, we obtain the approximate its assessment:

$$\sum_{\substack{p \leq n, \\ p \text{ is prime}}} \frac{n}{p} \approx n \ln \ln n + o(n),$$

QED.

More rigorous proof (and giving a more accurate estimate, up to constant factors) can be found in the book of Hardy and Wright "An Introduction to the Theory of Numbers" (p. 349).

Various optimizations sieve of Eratosthenes

The biggest drawback of the algorithm - that he "walks" on the memory permanently departing from the cache, which is why the constant hidden in the $O(n \log \log n)$relatively large.

In addition, for sufficiently large $n$bottleneck becomes the volume of consumed memory.

The following are the methods to both reduce the number of operations performed, as well as significantly reduce the memory consumption.

Sifting simple to the root

The most obvious point - that in order to find all simple to $n$sufficiently perform simple screening only, not exceeding a root n.

This will change the outer loop of the algorithm:

```
for (int i=2; i*i<=n; ++i)
```

On the asymptotic behavior of this optimization does not affect (indeed, repeating above proof, we obtain an estimate $n \ln \ln \sqrt{n} + o(n)$that, by the properties of the logarithm, is asymptotically the same), although the number of transactions decreased markedly.

Sieve only odd numbers

Since all even numbers, except 2, - components, we can not process any way at all even numbers and odd numbers operate only.

Firstly, it will halve the amount of memory required. Second, it will reduce the number of operations makes the algorithm by about half.

To reduce the amount of memory consumed

Note that Eratosthenes algorithm actually operates on $n$bits of memory. Therefore, it can save significant memory consumption, storing not $n$bytes - booleans and $n$bits, ie $n/8$bytes of memory.

However, this approach - **"bit compression"** - substantially complicate handling these bits. Any read or write bits will be of a few arithmetic operations, which ultimately will lead to a slowdown of the algorithm.

Thus, this approach is justified only if $n$sufficiently large that $n$bytes of memory to allocate anymore. Saving memory (in 8time), we will pay for it a substantial slowing of the algorithm.

In conclusion, it is worth noting that in C ++ containers have already been implemented, automatically performing the bit compression: vector <bool> and bitset <>. However, if speed is important, it is better to implement the compression bit manually, using bit operations - today compilers still not able to generate a reasonably fast code.

Block sieve

Optimization of "simple screening to the root" implies that there is no need to store all the time the whole array $prime[1 \dots n]$. To perform screening sufficient to store only the simple to the root of n, ie $prime[1 \dots \sqrt{n}]$, the remainder of the array $prime$to build a block by block, keeping the current time, only one block.

Let s- a constant determining the size of the block, then only will $\lceil \frac{n}{s} \rceil$the blocks kth block ($k = 0 \dots \lfloor \frac{n}{s} \rfloor$) contains a number in the interval $[ks; ks + s - 1]$. We processed blocks of the queue, i.e. for each kth block will go through all the simple (as 1before $\sqrt{n}$) and perform their screening only within the current block. Carefully handle should first block - firstly, from simple $[1; \sqrt{n}]$do not have to remove themselves, and secondly, the number $0$and 1should be

marked as not particularly easy. When processing the last block should also not forget the fact that the latter desired number $n$is not necessarily the end of the block.

We present the implementation of the sieve block. The program reads the number $n$and finds a number of simple $1$to n:

```
const int SQRT_MAXN = 100000; // корень из максимального значения N
const int S = 10000;
bool nprime[SQRT_MAXN], bl[S];
int primes[SQRT_MAXN], cnt;

int main() {

	int n;
	cin >> n;
	int nsqrt = (int) sqrt (n + .0);
	for (int i=2; i<=nsqrt; ++i)
		if (!nprime[i]) {
			primes[cnt++] = i;
			if (i * 1ll * i <= nsqrt)
				for (int j=i*i; j<=nsqrt; j+=i)
					nprime[j] = true;
		}

	int result = 0;
	for (int k=0, maxk=n/S; k<=maxk; ++k) {
		memset (bl, 0, sizeof bl);
		int start = k * S;
		for (int i=0; i<cnt; ++i) {
			int start_idx = (start + primes[i] - 1) / primes[i];
			int j = max(start_idx,2) * primes[i] - start;
			for (; j<S; j+=primes[i])
				bl[j] = true;
		}
		if (k == 0)
			bl[0] = bl[1] = true;
		for (int i=0; i<S && start+i<=n; ++i)
			if (!bl[i])
				++result;
	}
	cout << result;

}
```

Asymptotic behavior of the sieve block is the same as usual and the sieve of Eratosthenes (unless, of course, the size of $s$the blocks is not very small), but the amount of memory used will be reduced to $O(\sqrt{n}+s)$decrease and "walk" from memory. On the other hand, for each block for each of the simple $[1;\sqrt{n}]$division is performed, which will strongly affect in a smaller unit. Consequently, the choice of the constant $s$need to keep a balance.

Experiments show that the best performance is achieved when the $s$value is about $10^4$to 10^5.

Upgrade to a linear-time work

Eratosthenes algorithm can be converted to a different algorithm, which is already operational in linear time - see. Article "Sieve of Eratosthenes with linear time work" .(However, this algorithm has some limitations.)

5.Advanced Euclidean algorithm

While the "normal" Euclidean algorithm simply finds the greatest common divisor of two numbers a and b, extended Euclidean algorithm finds the GCD also factors in addition to x, and $y$such that:

$$a \cdot x + b \cdot y = \gcd(a, b).$$

Ie he finds the coefficients with which the GCD of two numbers expressed in terms of the numbers themselves.

Algorithm

Make the calculation of these coefficients in the Euclidean algorithm is simple enough to derive formulas by which they change from pair $(a, b)$to pair $(b\%a, a)$(percent sign denotes the modulo).

Thus, suppose we have found a solution $(x_1, y_1)$of the problem for a new pair $(b\%a, a)$:

$$(b\%a) \cdot x_1 + a \cdot y_1 = g,$$

and want to get a solution $(x, y)$for our couples (a, b):

$$a \cdot x + b \cdot y = g.$$

To do this, we transform the value of $b\%a$:

$$b\%a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a.$$

Substituting this in the above expression $x_1$and $y_1$obtain:

$$g = (b\%a) \cdot x_1 + a \cdot y_1 = \left(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a\right) \cdot x_1 + a \cdot y_1,$$

and performing regrouping terms, we obtain:

$$g = b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1\right).$$

Comparing this with the original expression of the unknown x, and $y$we obtain the required expression:

$$\begin{cases} x = y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1, \\ y = x_1. \end{cases}$$

Implementation

```
int gcd (int a, int b, int & x, int & y) {
```

```
        if (a == 0) {
                x = 0; y = 1;
                return b;
        }
        int x1, y1;
        int d = gcd (b%a, a, x1, y1);
        x = y1 - (b / a) * x1;
        y = x1;
        return d;
}
```

This is a recursive function, which still returns the GCD of the numbers $a$and b, but apart from that - as desired coefficients $x$and $y$as a function parameter, passed by reference.

Base of recursion - the case $a = 0$. Then GCD equal b, and, obviously, the desired ratio $x$and $y$are 0 and 1respectively. In other cases, the usual solution is working, and the coefficients are converted by the above formulas.

Advanced Euclidean algorithm in this implementation works correctly even for negative numbers.

Literature

- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. **Algorithms: Design and Analysis** [2005]

6.Fibonacci numbers

Definition

The Fibonacci sequence is defined as follows:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}.$$

The first few of its members:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ ,\ 55,\ 89,\ \ldots$$

History

These numbers introduced in 1202 by Leonardo Fibonacci (Leonardo Fibonacci) (also known as Leonardo of Pisa (Leonardo Pisano)). However, thanks to the 19th century mathematician Luca (Lucas) the name of "Fibonacci numbers" became common.

However, the Indian mathematicians mentioned number of this sequence even earlier: Gopal (Gopala) until 1135, Hemachandra (Hemachandra) - in 1150

Fibonacci numbers in nature

Fibonacci himself mentioned these numbers in connection with this task: "A man planted a pair of rabbits in a pen surrounded on all sides by a wall. How many pairs of rabbits per year can produce a pair of this, if you know that every month, starting from the second, each pair rabbits gives birth to a

pair? ". The solution to this problem and will be the number of sequences, now called in his honor. However, the situation described by Fibonacci - more mind game than real nature.

Indian mathematicians Gopala and Hemachandra mentioned this sequence number in relation to the number of rhythmic patterns resulting from the alternation of long and short syllables in verse, or the strong and weak beats in the music. The number of such patterns having generally $n$shares power F_n.

Fibonacci numbers appear in the work of Kepler in 1611, which reflected on the numbers found in nature (the work "On the hexagonal flakes").

An interesting example of plants - yarrow, in which the number of stems (and hence the flowers) is always a Fibonacci number. The reason for this is simple: as originally with a single stem, this stem is then divided by two, and then branches from the main stalk another, then the first two stems branch again, and then all the stems, but the last two, branch, and so on. Thus, each stalk after his appearance "skips" one branch, and then begins to divide at each level of branches, which results in a Fibonacci number.

Generally speaking, many colors (eg, lilies), the number of petals is a way or another Fibonacci number.

Also botanically known phenomenon of 'phyllotaxis'. As an example, the location of sunflower seeds: if you look down on their location, you can see two simultaneous series of spirals (like overlapping): some are twisted clockwise, the other - against. It turns out that the number of these spirals is roughly equal to two consecutive Fibonacci numbers: 34 and 55 or 89 and 144 Similar facts are true for some other colors, as well as pine cones, broccoli, pineapple, etc.

For many plants (according to some sources, 90% of them) are true and an interesting fact. Consider any sheet, and will descend downwardly until, until we reach the sheet disposed on the stem in the same way (i.e., directed exactly in the same way). Along the way, we assume that all the leaves that fall to us (ie, located at an altitude between the start and end sheet), but arranged differently. Numbering them, we will gradually make the turns around the stem (as the leaves are arranged on the stem in a spiral).Depending on whether the windings perform clockwise or counterclockwise will receive a different number of turns. But it turns out that the number of turns, committed us clockwise the number of turns, committed anti-clockwise, and the number of leaves encountered form 3 consecutive Fibonacci numbers.

However, it should be noted that there are plants for which The above calculations give the number of all other sequences, so you can not say that the phenomenon of phyllotaxis is the law - it is rather entertaining trend.

Properties

Fibonacci numbers have many interesting mathematical properties.

Here are just a few of them:

- Value for Cassini:

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n.$$

- Rule "addition":

$$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n.$$

- From the previous equation at $k = n$follows:

$$F_{2n} = F_n(F_{n+1} + F_{n-1}).$$

- From the previous equality by induction we can show that

  F_{nk}always divisible F_n.

- Converse is also true to the previous statement:

  if $F_m$fold F_n, the $m$fold n.

- GCD-equality:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}.$$

- With respect to the Euclidean algorithm Fibonacci numbers have the remarkable property that they are the worst input data for this algorithm (see. "Theorem Lame" inEuclid's algorithm).

Fibonacci number system

Zeckendorf theorem asserts that every positive integer $n$can be uniquely represented as a sum of Fibonacci numbers:

$$N = F_{k_1} + F_{k_2} + \ldots + F_{k_r}$$

where $k_1 \geq k_2 + 2, k_2 \geq k_3 + 2, \ldots, k_r \geq 2$(ie, can not be used in the recording of two adjacent Fibonacci numbers).

It follows that any number can be written uniquely in **the Fibonacci value** , for example:

$$9 = 8 + 1 = F_6 + F_1 = (10001)_F,$$
$$6 = 5 + 1 = F_5 + F_1 = (1001)_F,$$
$$19 = 13 + 5 + 1 = F_7 + F_5 + F_1 = (101001)_F,$$

And in any number can not go two units in a row.

It is easy to get and usually adding one to the number in the Fibonacci value: if the least significant digit is 0, then it is replaced by 1, and if it is equal to 1 (ie, in the end there is a 01), then 01 then 10 is replaced by the "fix" record sequentially correcting all 011 by 100 As a result, the linear time is obtained by recording a new number.

Translation numbers in the Fibonacci number system with a simple "greedy" algorithm: just iterate through the Fibonacci numbers from high to low, and if for some $F_k \leq n$, it $F_k$is included in the record number n, and we take away $F_k$from n, and continue to search.

The formula for the n-th Fibonacci number

Formula by radicals

There is a wonderful formula, called by the name of the French mathematician Binet (Binet), although it was known to him Moivre (Moivre):

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

This formula is easy to prove by induction, but you can bring it by the concept of forming or using the solution of the functional equation.

Immediately you will notice that the second term is always less than 1 in absolute value, and furthermore, decreases very rapidly (exponentially). This implies that the value of the first term gives the "almost" value F_n. This can be written simply as:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right],$$

where the square brackets denote rounding to the nearest integer.

However, for practical use in the calculation of these formulas hardly suitable, because they require very high precision work with fractional numbers.

Matrix formula for the Fibonacci numbers

It is easy to prove the following matrix equation:

$$\begin{pmatrix} F_{n-2} & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_{n-1} & F_n \end{pmatrix}.$$

But then, denoting

$$P \equiv \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

obtain

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix}.$$

Thus, to find nth Fibonacci number necessary to build a matrix $P$of power n.

Remembering that the construction of the matrix in the $n$degree of th can be accomplished $O(\log n)$ (see. binary exponentiation), it turns out that $n$the number of Fibonacci retracement can be easily calculated for $O(\log n)$c using only integer arithmetic.

Periodicity of the Fibonacci sequence modulo

Consider the Fibonacci sequence $F_i$modulo some p. We prove that it is periodic, and moreover the period begins with $F_1 = 1$(ie preperiod contains only F_0).

We prove this by contradiction. Consider $p^2 + 1$pairs of Fibonacci numbers taken modulo p:

$$(F_1, F_2),\ (F_2, F_3),\ \ldots,\ (F_{p^2+1}, F_{p^2+2}).$$

Since modulo $p$may be only p^2different pairs, there exists the sequence of at least two of the same pair. This already means that the sequence is periodic.

We now choose among all such identical pairs of two identical pairs with the lowest numbers. Let this pair with some of the rooms $(F_a, F_{a+1})$and (F_b, F_{b+1}). We will prove that $a = 1$. Indeed, otherwise they will have to previous couple $(F_{a-1}, F_a)$and (F_{b-1}, F_b)that, by the property of the Fibonacci numbers, will also be equal to each other.However, this contradicts the fact that we chose the matching pairs with the lowest numbers, as required.

Literature

- Ronald Graham, Donald Knuth, and Oren Patashnik. **Concrete Mathematics** [1998]

8.Reverse member ring modulo

Definition

Suppose we are given a natural module m, and consider the ring formed by the module (ie, consisting of numbers from 0before$m - 1$). Then for some elements of this ring can be found **an inverse element** .

The inverse of the number $a$modulo $m$called a number $b$that:

$$a \cdot b \equiv 1 \pmod{m},$$

and it is often denoted by a^{-1}.

It is clear that for the zero return item does not exist ever; for the remaining elements of the inverse can exist or not. It is argued that the inverse exists only for those elements $a$that **are relatively prime** to the modulus m.

Consider the following two ways of finding the inverse element employed, provided that it exists.

Finally, we consider an algorithm which allows you to find backlinks to all numbers modulo some linear time.

Finding using the Extended Euclidean algorithm

Consider the auxiliary equation (in the unknown $x$and y):

$$a \cdot x + m \cdot y = 1.$$

This linear Diophantine equation of second order . As shown in the corresponding article from the condition $\gcd(a, m) = 1$, it follows that this equation has a solution which can be found using the Extended Euclidean algorithm (hence the same way, it follows that when $\gcd(a, m) \neq 1$, solutions, and therefore the inverse element does not exist).

On the other hand, if we take from both sides of the residue modulo m, we get:

$$a \cdot x = 1 \pmod{m}.$$

Thus found $x$and will be the inverse of a.

Implementation (including that found $x$necessary to take on the module m, and $x$may be negative):

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
        cout << "no solution";
```

```
else {
        x = (x % m + m) % m;
        cout << x;
}
```

Asymptotic behavior of the solutions obtained $O(\log m)$.

Finding the binary exponentiation

We use Euler's theorem:

$$a^{\phi(m)} \equiv 1 \pmod m,$$

which is true just in the case of relatively prime $a$and m.

Incidentally, in the case of a simple module m, we get even more simple statement - Fermat's little theorem:

$$a^{m-1} \equiv 1 \pmod m.$$

Multiply both sides of each equation by a^{-1}, we obtain:

- for any module m:

$$a^{\phi(m)-1} \equiv a^{-1} \pmod m,$$

- for a simple module m:

$$a^{m-2} \equiv a^{-1} \pmod m.$$

Thus, we have obtained the formula for the direct calculation of the inverse. For practical applications typically use an efficient algorithm for binary exponentiation , which in this case will bring about for exponentiation $O(\log m)$.

This method seems to be a little bit easier as described in the previous paragraph, but it requires knowledge of the values of the Euler function that actually requires the factorization of the module m, which can sometimes be very difficult.

If the factorization of the number is known, then this method also works for the asymptotic behavior $O(\log m)$.

Finding all simple modulo a given linear time

Let a simple module m. Required for each number in the interval $[1; m-1]$to find its inverse.

Applying the algorithms described above, we get a solution with the asymptotic behavior $O(m \log m)$. Here we present a simple solution to the asymptotic behavior $O(m)$.

The decision is as follows. We denote $r[i]$the inverse of the desired number of $i$modulo m. Then the $i > 1$true identity:

$$r[i] = -\left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i]. \pmod m$$

The implementation of this amazingly concise solutions:

```
r[1] = 1;
for (int i=2; i<m; ++i)
        r[i] = (m - (m/i) * r[m%i] % m) % m;
```

The proof of this solution is a chain of simple transformations:

We write out the value $m \bmod i$:

$$m \bmod i = m - \left\lfloor \frac{m}{i} \right\rfloor \cdot i,$$

whence, taking both sides modulo m, we get:

$$m \bmod i = - \left\lfloor \frac{m}{i} \right\rfloor \cdot i. \pmod m$$

Multiplying both sides by the inverse of $i$the inverse to $(m \bmod i)$obtain the required formula:

$$r[i] = - \left\lfloor \frac{m}{i} \right\rfloor \cdot r[m \bmod i], \pmod m$$

QED.

9. Gray code

Definition

Gray code is called a system of numbering negative numbers when the codes of two adjacent numbers differ in exactly one bit.

For example, for the numbers of length 3 bits, we have a sequence of Gray codes: 000, 001, 011, 010, 110, 111, 101, 100.Eg $G(4) = 6$.

This code was invented by Frank Gray (Frank Gray) in 1953.

Finding the Gray code

Consider the number of bits $n$and the number of bits $G(n)$. Note that ith bit $G(n)$is equal to one only in the case where ith bit $n$equal to one and $i + 1$th bit is zero, or vice versa (ith bit is zero, and $i + 1$th is equal to unity). Thus, we have $G(n) = n \oplus (n >> 1)$:

```
int g (int n) {
        return n ^ (n >> 1);
}
```

Finding the inverse Gray code

Required by the Gray code $g$to restore the original number n.

We shall go on to junior high order bits (albeit the least significant bit is numbered 1, and the oldest - k). Obtain the following relations between the bits $n_i$of $n$bits and $g_i$number g:

$$\begin{aligned}
&n_k = g_k, \\
&n_{k-1} = g_{k-1} \oplus n_k = g_k \oplus g_{k-1}, \\
&n_{k-2} = g_{k-2} \oplus n_{k-1} = g_k \oplus g_{k-1} \oplus g_{k-2}, \\
&n_{k-3} = g_{k-3} \oplus n_{k-2} = g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3}, \\
&\dots
\end{aligned}$$

In the form of program code is the easiest way to record as follows:

```
int rev_g (int g) {
       int n = 0;
       for (; g; g>>=1)
              n ^= g;
       return n;
}
```

Applications

Gray codes have several applications in different areas, sometimes quite unexpected:

- n-bit Gray code corresponds to a Hamiltonian cycle on the nn-dimensional cube.
- In the art, Gray codes are used to **minimize errors** when converting the analog signals into digital signals (for example, sensors). In particular, the Gray code and are visible in connection with this application.
- Gray codes are used in solving the problem of **the Tower of Hanoi** .

  Let n- number of disks. Let's start with the Gray code of length n, consisting of all zeros (ie $G(0)$), and will move on Gray codes (from $G(i)$proceeding to $G(i+1)$).With every $i$bit of this -th Gray code ith disk (and the most significant bit corresponds to the smallest size drive, and the most significant bit - the largest). Since at each step exactly one bit is changed, then we can understand the bit change $i$as moving ith disc. Note that for all drives except the smallest, at each step there is exactly one option course (except for the starting and final products). For the smallest disk always has two variations, however, there is the strategy of choice course, always leads to the answer: if $n$is odd, then the sequence of movements of the smallest drive is of the form $f \to t \to r \to f \to t \to r \to \ldots$(where f- starting rod t- the final rod r- the remainder of the rod), and if it $n$is even then $f \to r \to t \to f \to r \to t \to \ldots$.

- Gray codes also are used in the theory of **genetic algorithms** .

Problem in online judges

List of tasks that can be taken, using Gray codes:

- SGU # 249 **"Matrix"** [Difficulty: Medium]

10. Long arithmetic

Long arithmetic - a set of tools (data structures and algorithms) that allow you to work with numbers much larger quantities than permitted by the standard data types.

Types of long integer arithmetic

Generally speaking, even if only in the Olympiad set of problems is large enough, so proizvedëm classification of different types of long arithmetic.

Classical long arithmetic

The basic idea is that the number is stored as an array of its digits.

The numbers can be used from one system or another value, commonly used decimal system and its power (ten thousand billion), or binary system.

Operations on numbers in the form of a long arithmetic produced by a "school" of algorithms for addition, subtraction, multiplication, long division. However, they are also useful algorithms for fast multiplication: Fast Fourier transform and the Karatsuba algorithm.

Described here only work with non-negative long numbers. To support negative numbers must enter and maintain additional flag "negativity" numbers, or else work in complementary codes.

Data Structure

Keep long numbers will be in the form of a vector of numbers int, where each element - a single digit number.

```
typedef vector<int> lnum;
```

To improve the efficiency of the system will work in base billion, i.e. each element of the vector $lnum$ contains not one, but 9numbers:

```
const int base = 1000*1000*1000;
```

The numbers will be stored in a vector in such a manner that at first there are the least significant digit (ie, ones, tens, hundreds, etc.).

Furthermore, all the operations are implemented in such a manner that after any of them leading zeros (i.e., extra zeros at the beginning number) does not (of course, on the assumption that, before each operation the leading zeros, also available). It should be noted that the implementation representation for the number zero is well supported from two views: the empty vector numbers and vector numbers containing a single element - zero.

Conclusion

The most simple - it's the conclusion of a long number.

First, we simply display the last element of the vector (or 0, if the vector is empty), and then derive all the remaining elements of the vector, adding zeros to their 9characters:

```
printf ("%d", a.empty() ? 0 : a.back());
for (int i=(int)a.size()-2; i>=0; --i)
	printf ("%09d", a[i]);
```

(Here, a little subtle point: not to forget to write down the cast (int), because otherwise the number $a.size()$will be unsigned, and if $a.size() \le 1$, it will happen in the subtraction overflow)

Reading

Reads a line in $string$, and then convert it into a vector:

```
for (int i=(int)s.length(); i>0; i-=9)
	if (i < 9)
		a.push_back (atoi (s.substr (0, i).c_str()));
	else
		a.push_back (atoi (s.substr (i-9, 9).c_str()));
```

If used instead of $string$the array $char$'s, the code will be more compact:

```
for (int i=(int)strlen(s); i>0; i-=9) {
      s[i] = 0;
      a.push_back (atoi (i>=9 ? s+i-9 : s));
}
```

If the input number has leading zeros may be, they can remove after reading the following way:

```
while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Addition

Adds to the number of $a$the number $b$and stores the result in a:

```
int carry = 0;
for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
      if (i == a.size())
            a.push_back (0);
      a[i] += carry + (i < b.size() ? b[i] : 0);
      carry = a[i] >= base;
      if (carry)  a[i] -= base;
}
```

Subtraction

Takes the number of $a$the number of b($a \geq b$) and stores the result in a:

```
int carry = 0;
for (size_t i=0; i<b.size() || carry; ++i) {
      a[i] -= carry + (i < b.size() ? b[i] : 0);
      carry = a[i] < 0;
      if (carry)  a[i] += base;
}
while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Here we after subtraction remove leading zeros in order to maintain a predicate that they do not exist.

Multiplying the length of a short

Multiplies long $a$to short b($b < \text{base}$) and stores the result in a:

```
int carry = 0;
for (size_t i=0; i<a.size() || carry; ++i) {
      if (i == a.size())
            a.push_back (0);
      long long cur = carry + a[i] * 1ll * b;
      a[i] = int (cur % base);
      carry = int (cur / base);
}
while (a.size() > 1 && a.back() == 0)
      a.pop_back();
```

Here we after dividing remove leading zeros in order to maintain a predicate that they do not exist.

(Note: The method **further optimization** . If performance is critical, you can try to replace the two division one: to count only the integer portion of a division (in the code is a variable $carry$), and then count on it the remainder of the division (with the help of one multiplication) . Typically, this technique allows faster code, but not very much.)

Multiplication of two long numbers

Multiplies $a$on $b$and results are stored in c:

```
lnum c (a.size()+b.size());
for (size_t i=0; i<a.size(); ++i)
	for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {
		long long cur = c[i+j] + a[i] * 1ll * (j < (int)b.size() ?
b[j] : 0) + carry;
		c[i+j] = int (cur % base);
		carry = int (cur / base);
	}
while (c.size() > 1 && c.back() == 0)
	c.pop_back();
```

Long division for a short

Divides long $a$for short b($b < \text{base}$), private preserves a, balance $carry$:

```
int carry = 0;
for (int i=(int)a.size()-1; i>=0; --i) {
	long long cur = a[i] + carry * 1ll * base;
	a[i] = int (cur / b);
	carry = int (cur % b);
}
while (a.size() > 1 && a.back() == 0)
	a.pop_back();
```

Long arithmetic in factored form

Here the idea is to keep not the number itself, and its factorization, ie power of each of her children simple.

This method is also very simple to implement, and it is very easy to perform multiplication and division, but it is impossible to perform the addition or subtraction. On the other hand, this method saves memory in comparison with the "classical" approach, and allows you to divide and multiply significantly (asymptotically) faster.

This method is often used when you need to make the division on the delicate module: then it is enough to store a number in the form of powers to the prime divisors of this module, and another number - balance on the same module.

Long arithmetic in simple modules (Chinese theorem or scheme Garner)

The bottom line is that you choose a certain system modules (usually small, fit into standard data types), and the number is stored as a vector of residuals from his division to each of these modules.

According to the Chinese remainder theorem, it is enough to uniquely store any number between 0 and the product of these modules minus one. Thus there Garner algorithm , which allows to make a recovery from the modular form in the usual "classical" form number.

Thus, this method saves memory compared to the "classical" long arithmetic (although in some cases not as radically as the factorization method). In addition, in a modular fashion, you can very quickly make addition, subtraction and multiplication, - all for adding identical time asymptotically proportional to the number of modules in the system.

However, all this is very time consuming given the price of the translation of this modular form in the usual form, which, in addition to considerable time-consuming, require also the implementation of "classical" long arithmetic multiplication.

In addition, to make **the division** of numbers in this representation in simple modules is not possible.

Types of fractional long arithmetic

Operations on fractional numbers found in the Olympiad problems are much less common, and work with large fractional numbers is much more difficult, so in the Olympic Games found only a specific subset of the fractional long arithmetic.

Long arithmetic in an irreducible fraction

Number is represented as an irreducible fraction $\frac{a}{b}$, where $a$and b- integers. Then all operations on fractional numbers easily reduced to operations on the numerator and denominator of the fractions.

Normally this storage numerator and denominator have to use long arithmetic, but, however, it is the simplest form - the "classical" long arithmetic, although sometimes is sufficiently embedded 64-bit numeric type.

Isolation of floating point position as a separate type

Sometimes the problem is required to make calculations with very large or very small numbers, but it does not prevent them from overflowing. Built $8-10$-baytovy type $double$is known allows the exponent value in a range $[-308;308]$, which sometimes may be insufficient.

Reception, in fact, very simple - introduce another integer variable that is responsible for the exponential, and after each operation, the fractional number of "normal", ie, returns to the segment $[0.1;1)$, by increasing or decreasing exponential.

When multiplying or dividing two such numbers it is necessary to lay down accordingly or subtract their exponents. When adding or subtracting before proceeding number should lead to an exponential one, which one of them is multiplied by the 10difference in the degree of exponents.

Finally, it is clear that it is not necessary to choose 10as the base of the exponent. Based on the device embedded floating-point types, the best seems to put an equal basis 2.

11. Discrete logarithm

The discrete logarithm problem is that according to an a, b, $m$solve the equation:

$$a^x = b \pmod m,$$

where $a$and m- **are relatively prime** (note: if they are not relatively prime, then the algorithm described below is incorrect, though, presumably, it can be modified so that it is still working).

Here we describe an algorithm, known as the **"Baby-Step-giant-Step algorithm"** , proposed by **Shanks (Shanks)** in 1971, working at the time for $O(\sqrt{m}\log m)$. Often, this algorithm is simply

called an algorithm **"meet-in-the-middle"** (because it is one of the classic applications technology "meet-in-the-middle": "separation of tasks in half").

Algorithm

So, we have the equation:

$$a^x = b \pmod m,$$

where $a$and $m$are relatively prime.

Transform equation. Put

$$x = np - q,$$

where n- is a preselected constant (as her chosen depending on m, we will understand later). Sometimes $p$called "giant step" (because increasing it by one increases $x$at once n), as opposed to it q- "baby step".

It is clear that any x(in the interval $[0; m)$- it is clear that such a range of values will be enough) can be represented in a form where, for this will be enough values:

$$p \in \left[1; \left\lceil \frac{m}{n} \right\rceil\right], \qquad q \in [0; n].$$

Then the equation becomes:

$$a^{np-q} = b \pmod m,$$

hence, using the fact that $a$and $m$are relatively prime, we obtain

$$a^{np} = ba^q \pmod m.$$

In order to solve the original equation, you need to find the appropriate values $p$and $q$to the values of the left and right parts of the match. In other words, it is necessary to solve the equation:

$$f_1(p) = f_2(q).$$

This problem is solved using the meet-in-the-middle as follows. The first phase of the algorithm: calculate the value of the function $f_1$for all values of the argument p, and we can sort the values. The second phase of the algorithm: Let's take the value of the second variable q, compute the second function f_2, and look for the value of the predicted values of the first function using a binary search.

Asymptotics

First, we estimate the computation time of each of the functions $f_1(p)$and $f_2(q)$. And she and the other contains the construction of the power that can be performed using the algorithm binary exponentiation . Then both of these functions, we can compute in time $O(\log m)$.

The algorithm itself in the first phase comprises computing functions $f_1(p)$for each possible value p and the further sorting of values that gives us the asymptotic behavior:

$$O\left(\left\lceil \frac{m}{n} \right\rceil \left(\log m + \log \left\lceil \frac{m}{n} \right\rceil\right)\right) = O\left(\left\lceil \frac{m}{n} \right\rceil \log m\right).$$

In the second phase of the algorithm is evaluated function $f_2(q)$for each possible value $q$and a binary search on an array of values $f_1$that gives us the asymptotic behavior:

$$O\left(n\left(\log m + \log\left\lceil\frac{m}{n}\right\rceil\right)\right) = O\left(n \log m\right).$$

Now, when we add these two asymptotes, we can do it $\log m$, multiplied by the sum $n$and m/n, and almost obvious that the minimum is attained when $n \approx m/n$, that is, for optimum performance of the algorithm constant $n$should be chosen:

$$n \approx \sqrt{m}.$$

Then the asymptotic behavior of the algorithm takes the form:

$$O\left(\sqrt{m}\ \log m\right).$$

Note. We could exchange roles $f_1$and f_2(ie the first phase to calculate the values of the function f_2, and the second - f_1), but it is easy to understand that the result will not change, and the asymptotic behavior that we can not improve.

Implementation

The simplest implementation

The function powmodperforms a binary construction of $a$the power $b$mod m, see. binary exponentiation .

The function solveproduces a proper solution to the problem. This function returns a response (the number in the interval $[0; m)$), or more precisely, one of the answers.Function will return -1if there is no solution.

```
int powmod (int a, int b, int m) {
	int res = 1;
	while (b > 0)
		if (b & 1) {
			res = (res * a) % m;
			--b;
		}
		else {
			a = (a * a) % m;
			b >>= 1;
		}
	return res % m;
}

int solve (int a, int b, int m) {
	int n = (int) sqrt (m + .0) + 1;
	map<int,int> vals;
	for (int i=n; i>=1; --i)
		vals[ powmod (a, i * n, m) ] = i;
	for (int i=0; i<=n; ++i) {
		int cur = (powmod (a, i, m) * b) % m;
		if (vals.count(cur)) {
			int ans = vals[cur] * n - i;
			if (ans < m)
				return ans;
		}
```

```
        }
        return -1;
}
```

Here we are for the convenience of the implementation of the first phase of the algorithm used the data structure "map" (red-black tree) that for each value of the function $f_1(i)$stores the argument $i$in which this value is reached. Here, if the same value is achieved repeatedly recorded smallest of all the arguments. This is done to ensure that subsequently, in the second phase of the algorithm found in the response interval $[0; m)$.

Given that the argument of $f_1()$the first phase we pawing away from one and up n, and the argument of the function $f_2()$in the second phase moves from zero to n, in the end, we cover the entire set of possible answers, because segment $[0; n^2]$contains a gap $[0; m)$. In this case, the negative response could not get, and the responses of greater than or equal m, we can not ignore - should still be in the corresponding period of the answers $[0; m)$.

This function can be changed in the event that if you want to find **all the solutions** of the discrete logarithm. To do this, replace the "map" on any other data structure that allows for a single argument to store multiple values (for example, "multimap"), and to amend the code of the second phase.

An improved

When **optimizing for speed** , you can proceed as follows.

Firstly, immediately catches the eye uselessness binary exponentiation in the second phase of the algorithm. Instead, you can just make it multiply the variable and every time a.

Secondly, in the same way you can get rid of the binary exponentiation, and in the first phase: in fact, once is enough to calculate the value a^n, and then simply multiply the at her.

Thus, the logarithm of the asymptotic behavior will remain, but it will only log associated with the data structure $map <>$(ie, in terms of the algorithm, with sorting and binary search values) - ie it will be the logarithm of $\sqrt{m}$that in practice gives a noticeable boost.

```
int solve (int a, int b, int m) {
	int n = (int) sqrt (m + .0) + 1;

	int an = 1;
	for (int i=0; i<n; ++i)
		an = (an * a) % m;

	map<int,int> vals;
	for (int i=1, cur=an; i<=n; ++i) {
		if (!vals.count(cur))
			vals[cur] = i;
		cur = (cur * an) % m;
	}

	for (int i=0, cur=b; i<=n; ++i) {
		if (vals.count(cur)) {
			int ans = vals[cur] * n - i;
			if (ans < m)
				return ans;
```

```
            }
            cur = (cur * a) % m;
        }
        return -1;
}
```

Finally, if the unit $m$is small enough, it can and does get rid of the logarithm in the asymptotic behavior - instead of just having got map <>a regular array.

You can also recall the hash table: on average, they work well for $O(1)$that, in general, gives the asymptotic behavior $O(\sqrt{m})$.

12. Linear Diophantine equations in two variables

Diophantine equation with two unknowns has the form:

$$a \cdot x + b \cdot y = c,$$

where a, b, c- given integers, $x$and y- unknown integers.

Below we consider some classical problems on these equations: finding any solution, obtaining all solutions, finding the number of solutions and the solutions themselves in a certain interval, to find a solution with the least amount of unknowns.

Degenerate case

A degenerate case we immediately excluded from consideration when $a = b = 0$. In this case, of course, the equation has infinite number of random or solutions, or it has no solution at all (depending on whether $c = 0$or not).

Finding the solution

Find one of the solutions of the Diophantine equation with two unknowns, you can use the Extended Euclidean algorithm . Assume first that the numbers $a$and $b$non-negative.

Advanced Euclidean algorithm to specify a non-negative numbers $a$and $b$finds their greatest common divisor g, as well as such factors $x_g$and $y_g$that:

$$a \cdot x_g + b \cdot y_g = g.$$

It is argued that if $c$divisible by $g = \gcd(a, b)$, the Diophantine equation $a \cdot x + b \cdot y = c$has a solution; otherwise Diophantine equation has no solutions. This follows from the obvious fact that a linear combination of two numbers still can be divided by a common divisor.

Suppose that $c$is divided into g, then obviously performed:

$$a \cdot x_g \cdot (c/g) + b \cdot y_g \cdot (c/g) = c,$$

ie one of the solutions of the Diophantine equation are the numbers:

$$\begin{cases} x_0 = x_g \cdot (c/g), \\ y_0 = y_g \cdot (c/g). \end{cases}$$

We have described the decision in the case where the number $a$and $b$non-negative. If one of them or both are negative, then we can proceed as follows: take their modulus and apply them Euclid's algorithm, as described above, and then found to change the sign $x_0$and $y_0$the present symbol numbers $a$and b, respectively.

Implementation (recall here, we believe that the input data $a = b = 0$are not allowed):

```
int gcd (int a, int b, int & x, int & y) {
	if (a == 0) {
		x = 0; y = 1;
		return b;
	}
	int x1, y1;
	int d = gcd (b%a, a, x1, y1);
	x = y1 - (b / a) * x1;
	y = x1;
	return d;
}

bool find_any_solution (int a, int b, int c, int & x0, int & y0, int & g)
{
	g = gcd (abs(a), abs(b), x0, y0);
	if (c % g != 0)
		return false;
	x0 *= c / g;
	y0 *= c / g;
	if (a < 0)   x0 *= -1;
	if (b < 0)   y0 *= -1;
	return true;
}
```

Getting all the solutions

We show how to obtain all the other solutions (and there are an infinite number) of the Diophantine equation, knowing one of the solutions (x_0, y_0).

Thus, suppose $g = \gcd(a, b)$, and the numbers x_0, y_0satisfy the condition:

$$a \cdot x_0 + b \cdot y_0 = c.$$

Then we note that, by adding to the $x_0$number $b/g$and at the same time taking away $a/g$from y_0, we do not disturb the equality:

$$a \cdot (x_0 + b/g) + b \cdot (y_0 - a/g) = a \cdot x_0 + b \cdot y_0 + a \cdot b/g - b \cdot a/g = c.$$

Obviously, this process can be repeated any number, ie all numbers of the form:

$$\begin{cases} x = x_0 + k \cdot b/g, \\ y = y_0 - k \cdot a/g, \end{cases} \quad k \in Z$$

are solutions of the Diophantine equation.

Moreover, only the number of this type are solutions, ie we describe the set of all solutions of the Diophantine equation (it turned out to be infinite if not imposed additional conditions).

Finding the number of solutions and the solutions themselves in a given interval

Given two segments $[min_x; max_x]$and $[min_y; max_y]$, and you want to find the number of solutions $(x, y)$of the Diophantine equation lying in these segments, respectively.

Note that if one of the numbers $a, b$is zero, then the problem has at most one solution, so these cases in this section, we exclude from consideration.

First, find a solution with the minimum appropriate x, ie $x \geq min_x$. To do this, first find any solution to the Diophantine equation (see para. 1). Then get out of it the solution with the least $x \geq min_x$- for this we use the procedure described in the preceding paragraph, and will increase / decrease x, until it is $\geq min_x$, and thus minimal. This can be done $O(1)$, considering a coefficient with which this conversion must be applied to obtain a minimum number greater than or equal to min_x. Denote found $x$through $lx1$.

Similarly, we can find an appropriate solution with a maximum $x = rx1$, ie $x \leq max_x$.

Then move on to the satisfaction of restrictions y, ie consideration of the segment $[min_y; max_y]$. The method described above will find a solution with the minimum $y \geq min_y$and maximum solution $y \leq max_y$. Denote $x$the coefficients of these solutions through $lx2$and $rx2$, respectively.

Cross the line segments $[lx1; rx1]$and $[lx2; rx2]$; denote the resulting cut through $[lx; rx]$. It is argued that any decision which $x$the coefficient is in $[lx; rx]$- any such decision is appropriate. (This is true in virtue of the construction of this segment: we first met separately restrictions $x$and $y$getting two segments, and then crossed them, having an area in which both conditions are satisfied.)

Thus, the number of solutions will be equal to the length of this interval, divided by $|b|$(since $x$the coefficient may be changed only $\pm b$) plus one.

We give implementation (it is difficult to obtain because it requires carefully consider the cases of positive and negative coefficients $a$and b)

```
void shift_solution (int & x, int & y, int a, int b, int cnt) {
	x += cnt * b;
	y -= cnt * a;
}

int find_all_solutions (int a, int b, int c, int minx, int maxx, int miny,
int maxy) {
	int x, y, g;
	if (! find_any_solution (a, b, c, x, y, g))
		return 0;
	a /= g;  b /= g;

	int sign_a = a>0 ? +1 : -1;
	int sign_b = b>0 ? +1 : -1;

	shift_solution (x, y, a, b, (minx - x) / b);
	if (x < minx)
		shift_solution (x, y, a, b, sign_b);
```

```
        if (x > maxx)
                return 0;
        int lx1 = x;

        shift_solution (x, y, a, b, (maxx - x) / b);
        if (x > maxx)
                shift_solution (x, y, a, b, -sign_b);
        int rx1 = x;

        shift_solution (x, y, a, b, - (miny - y) / a);
        if (y < miny)
                shift_solution (x, y, a, b, -sign_a);
        if (y > maxy)
                return 0;
        int lx2 = x;

        shift_solution (x, y, a, b, - (maxy - y) / a);
        if (y > maxy)
                shift_solution (x, y, a, b, sign_a);
        int rx2 = x;

        if (lx2 > rx2)
                swap (lx2, rx2);
        int lx = max (lx1, lx2);
        int rx = min (rx1, rx2);

        return (rx - lx) / abs(b) + 1;
}
```

Also it is easy to add to this realization the withdrawal of all the solutions found: it is enough to enumerate $x$in a segment $[lx;rx]$with a step $|b|$by finding for each of them corresponding $y$directly from Eq $ax+by=c$.

Finding solutions in a given interval with the least amount of x + y

Here on $x$and $y$should also be imposed any restrictions, otherwise the answer will almost always be negative infinity.

The idea of the solution is the same as in the previous paragraph: first find any solution to the Diophantine equation, and then apply this procedure in the previous section, we arrive at the best solution.

Indeed, we have the right to do the following transformation (see. Previous paragraph):

$$\begin{cases} x' = x + k \cdot (b/g), \\ y' = y - k \cdot (a/g), \end{cases} \quad k \in Z.$$

Note that the sum of $x+y$changes as follows:

$$x'+y' = x+y+k \cdot (b/g - a/g) = x+y+k \cdot (b-a)/g.$$

Ie if $a < b$it is necessary to choose the smallest possible value k, if $a > b$it is necessary to choose the largest possible value k.

If $a = b$we can not improve the solution - all solutions will have the same amount.

Problem in online judges

List of tasks that can be taken on the subject of Diophantine equations with two unknowns:

- SGU # 106 **"The Equation"** [Difficulty: Medium]

13. Modular linear equation of the first order

Statement of the Problem

This equation of the form:

$$a \cdot x = b \pmod{n},$$

where a, b, n- given integers, x- the unknown integer.

Required to find the desired value $x$lying in the interval $[0; n-1]$(as on the real line, it is clear there can be infinitely many solutions that are different to each other on $n \cdot k$where k- any integer). If the solution is not unique, then we will see how to get all the solutions.

Solution by finding the inverse element

Consider first the simplest case - when $a$and **are relatively prime** . Then we can find the inverse of a number , and multiplying it by both sides of the equation to get a solution (and it will be **the only**):n a

$$x = b \cdot a^{-1} \pmod{n}$$

Now consider the case $a$and **are not relatively prime** . Then, obviously, the decision will not always exist (for example).n $2 \cdot x = 1 \pmod{4}$

Suppose $g = \gcd(a, n)$, that is, their greatest common divisor (which in this case is greater than one).

Then, if $b$not divisible by $g$then no solution exists. In fact, if any $x$left side of the equation, i.e. $(a \cdot x) \pmod{n}$, is always divisible by g, while the right part it is not divided, which implies that there are no solutions.

If it $b$is divisible by g, then dividing both sides by it g(ie, dividing a, $b$and $n$on g), we arrive at a new equation:

$$a' \cdot x = b' \pmod{n'}$$

where $a'$and n'already be relatively prime, and this equation we have learned to solve. We denote its solution through x'.

Clearly, this x'will also be a solution of the original equation. If, however $g > 1$, it is **not the only** solution. It can be shown that the original equation will have exactly $g$the decisions and they will look like:

$$x_i = (x' + i \cdot n') \pmod{n},$$
$$i = 0 \ldots (g-1).$$

To summarize, we can say that **the number of solutions** of linear modular equations is either $g = \gcd(a, n)$, or zero.

Solution with the Extended Euclidean algorithm

We give our modular equation to a Diophantine equation as follows:

$$a \cdot x + n \cdot k = b,$$

where $x$and k- unknown integers.

The method of solving this equation is described in the relevant article of linear Diophantine equations of the second order , and it is in the application of the Extended Euclidean algorithm .

There is also described a method for obtaining all solutions of this equation for one found the solution, and, by the way, this way on closer examination is absolutely equivalent to the method described in the preceding paragraph.

14. The Chinese remainder theorem

The wording

In its modern formulation of the theorem is as follows:

Let $p = p_1 \cdot p_2 \cdot \ldots \cdot p_k$, where p_i- pairwise relatively prime numbers.

We associate with an arbitrary number of tuple , where :$a\ (0 \le a < p)(a_1, \ldots, a_k)$
$a_i \equiv a \pmod{p_i}$

$$a \iff (a_1, \ldots, a_k).$$

Then the correspondence (between numbers and tuples) will be **one to one** . And, moreover, the operations performed on the number a, can be equivalently performed on the corresponding element of the tuple - by the independent performance of operations on each component.

That is, if

$$a \iff \Big(a_1, \ldots, a_k\Big),$$
$$b \iff \Big(b_1, \ldots, b_k\Big),$$

then we have:

$$(a + b) \pmod{p} \iff \Big((a_1 + b_1) \pmod{p_1}, \ldots, (a_k + b_k) \pmod{p_k}\Big),$$
$$(a - b) \pmod{p} \iff \Big((a_1 - b_1) \pmod{p_1}, \ldots, (a_k - b_k) \pmod{p_k}\Big),$$
$$(a \cdot b) \pmod{p} \iff \Big((a_1 \cdot b_1) \pmod{p_1}, \ldots, (a_k \cdot b_k) \pmod{p_k}\Big).$$

In its original formulation of this theorem was proved by the Chinese mathematician Sun Tzu around 100 AD Namely, he showed in the particular case of the equivalence of solutions of the system of modular equations and solutions of one of the modular equation (see. Corollary 2 below).

Corollary 1

Modular system of equations:

$$\begin{cases} x \equiv a_1 \pmod{p_1}, \\ \dots, \\ x \equiv a_k \pmod{p_k} \end{cases}$$

has a unique solution modulo p.

(As above $p = p_1 \cdot \ldots \cdot p_k$, the numbers $p_i$are relatively prime, and the set $a_1, \ldots, a_k$- an arbitrary set of integers)

Corollary 2

The consequence is a connection between the system of modular equations and a corresponding modular equation:

The equation:

$$x \equiv a \pmod{p}$$

equivalent to the system of equations:

$$\begin{cases} x \equiv a \pmod{p_1}, \\ \dots, \\ x \equiv a \pmod{p_k} \end{cases}$$

(As above, it is assumed that $p = p_1 \cdot \ldots \cdot p_k$, the number $p_i$of pairwise relatively prime, and a- an arbitrary integer)

Algorithm Garner

Of the Chinese remainder theorem, it follows that it is possible to replace operations on the number of operations on tuples. Recall, each number $a$is assigned a tuple $(a_1, \ldots, a_k)$, where:

$$a_i \equiv a \pmod{p_i}.$$

It can be widely used in practice (in addition to the direct application for the restoration of its residues on the different modules), as we thus can replace surgery in the long arithmetic operations with an array of "short" numbers. Say, an array of 1000elements of "enough" to number around 3000 characters (if selected as $p_i$the first -s 1000simple); and if selected as a $p_i$simple -s about a billion, then enough already by with about 9000signs. But, of course, then you need to learn how to **restore** by $a$this tuple. From Corollary 1 shows that such a recovery is possible and, moreover, the only (provided $0 \le a < p_1 \cdot p_2 \cdot \ldots \cdot p_k$). **Garner algorithm** is an algorithm that allows to perform this restoration, with effectively.

We seek a solution in the form of:

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1},$$

ie mixed value with digit weights $p_1, p_2, \ldots, p_k$.

We denote by r_{ij}($i = 1 \ldots k-1$, $j = i+1 \ldots k$) the number of which is the inverse p_i modulo p_j(finding the inverse elements in the ring mod described here :

$$r_{ij} = (p_i)^{-1} \pmod{p_j}.$$

Substituting the expression $a$in a mixed value in the first equation, we get:

$$a_1 \equiv x_1.$$

We now substitute in the second equation:

$$a_2 \equiv x_1 + x_2 \cdot p_1 \pmod{p_2}.$$

Transform this expression, subtracting on both sides $x_1$and dividing by p_1:

$$\begin{aligned} a_2 - x_1 &\equiv x_2 \cdot p_1 \pmod{p_2}; \\ (a_2 - x_1) \cdot r_{12} &\equiv x_2 \pmod{p_2}; \\ x_2 &\equiv (a_2 - x_1) \cdot r_{12} \pmod{p_2}. \end{aligned}$$

Substituting into the third equation, we obtain a similar manner:

$$\begin{aligned} a_3 &\equiv x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 \pmod{p_3}; \\ (a_3 - x_1) \cdot r_{13} &\equiv x_2 + x_3 \cdot p_2 \pmod{p_3}; \\ ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} &\equiv x_3 \pmod{p_3}; \\ x_3 &\equiv ((a_3 - x_1) \cdot r_{13} - x_2) \cdot r_{23} \pmod{p_3}. \end{aligned}$$

Already clearly visible pattern that is easier to express the code:

```
for (int i=0; i<k; ++i) {
      x[i] = a[i];
      for (int j=0; j<i; ++j) {
            x[i] = r[j][i] * (x[i] - x[j]);

            x[i] = x[i] % p[i];
            if (x[i] < 0)  x[i] += p[i];
      }
}
```

So we learned how to calculate the coefficients $x_i$of the time $O(k^2)$, the very same answer - number a- can be restored by the formula:

$$a = x_1 + x_2 \cdot p_1 + x_3 \cdot p_1 \cdot p_2 + \ldots + x_k \cdot p_1 \cdot \ldots \cdot p_{k-1}.$$

It should be noted that in practice almost always need to calculate the answer using the Long arithmetic , but the coefficients themselves $x_i$are still calculated on the built-in types, but because the whole algorithm Garner is very effective.

Implementation of the algorithm Garner

Most convenient to implement this algorithm in Java, and because it contains a standard length arithmetic, and therefore there are no problems with the transfer of the number of modular system in an ordinary number (a standard class BigInteger).

The below implementation of the algorithm Garner support addition, subtraction and multiplication, with support work with negative numbers (see about this. Explanation after the code). Implemented transfer of conventional desyatichkogo represent a modular system and vice versa.

In this example, taken 100after the simple 10^9, allowing you to work with numbers up to about 10^{900}.

```
final int SZ = 100;
int pr[] = new int[SZ];
int r[][] = new int[SZ][SZ];
```

```
void init() {
	for (int x=1000*1000*1000, i=0; i<SZ; ++x)
		if (BigInteger.valueOf(x).isProbablePrime(100))
			pr[i++] = x;

	for (int i=0; i<SZ; ++i)
		for (int j=i+1; j<SZ; ++j)
			r[i][j] = BigInteger.valueOf( pr[i] ).modInverse(
					BigInteger.valueOf( pr[j] )
).intValue();
}


class Number {

	int a[] = new int[SZ];

	public Number() {
	}

	public Number (int n) {
		for (int i=0; i<SZ; ++i)
			a[i] = n % pr[i];
	}

	public Number (BigInteger n) {
		for (int i=0; i<SZ; ++i)
			a[i] = n.mod( BigInteger.valueOf( pr[i] )
).intValue();
	}

	public Number add (Number n) {
		Number result = new Number();
		for (int i=0; i<SZ; ++i)
			result.a[i] = (a[i] + n.a[i]) % pr[i];
		return result;
	}

	public Number subtract (Number n) {
		Number result = new Number();
		for (int i=0; i<SZ; ++i)
			result.a[i] = (a[i] - n.a[i] + pr[i]) % pr[i];
		return result;
	}

	public Number multiply (Number n) {
```

```
            Number result = new Number();
            for (int i=0; i<SZ; ++i)
                  result.a[i] = (int)( (a[i] * 1l * n.a[i]) % pr[i]
);
            return result;
      }

      public BigInteger bigIntegerValue (boolean can_be_negative) {
            BigInteger result = BigInteger.ZERO,
                  mult = BigInteger.ONE;
            int x[] = new int[SZ];
            for (int i=0; i<SZ; ++i) {
                  x[i] = a[i];
                  for (int j=0; j<i; ++j) {
                        long cur = (x[i] - x[j]) * 1l * r[j][i];
                        x[i] = (int)( (cur % pr[i] + pr[i]) % pr[i]
);
                  }
                  result = result.add( mult.multiply(
BigInteger.valueOf( x[i] ) ) );
                  mult = mult.multiply( BigInteger.valueOf( pr[i] )
);
            }

            if (can_be_negative)
                  if (result.compareTo( mult.shiftRight(1) ) >= 0)
                        result = result.subtract( mult );

            return result;
      }
}
```

Support for the **negative** numbers deserves mention (flag $\mathrm{can\_be\_negative}$ function $\mathrm{bigIntegerValue}()$). Modular scheme itself does not imply differences between positive and negative numbers. However, it can be seen that if a particular problem the answer modulo does not exceed half of the product of all primes, the positive numbers will be different from the negative that the positive numbers turn out less than this mid-, and negative - more. Therefore, we are after classical algorithm Garner compare the result with the middle, and if it is, then we derive a minus, and invert the result (ie, subtract it from the product of all primes, and print it already).

15. Finding a power divider factorial

Given two numbers $n$and k. Required to calculate with any degree of the divisor $k$is one $n!$, ie find the largest $x$such that $n!$is divided into k^x.

Solution for the case of simple k

Consider first the case when $k$simple.

We write down the expression for the factorial explicitly:

$$n! = 1\ 2\ 3\ \ldots\ (n-1)\ n$$

Note that each $k$member of the th of this work is divided into k, ie allows one to answer; the number of such members of the same $\lfloor n/k \rfloor$.

Further, we note that each k^2th term of this series is divided into k^2, ie gives one more to the answer (given that $k$in the first degree has already been considered before); the number of such members of the same $\lfloor n/k^2 \rfloor$.

And so on, every k^ith term of the series gives one to answer, and the number of members equal $\lfloor n/k^i \rfloor$.

Thus, the magnitude of response is:

$$\frac{n}{k} + \frac{n}{k^2} + \ldots + \frac{n}{k^i} + \ldots$$

This amount, of course, is not infinite, because only the first about $\log_k n$the members are non-zero. Consequently, the asymptotic behavior of the algorithm is $O(\log_k n)$.

Implementation:

```
int fact_pow (int n, int k) {
	int res = 0;
	while (n) {
		n /= k;
		res += n;
	}
	return res;
}
```

Solution for the case of composite k

The same idea is applied directly anymore.

But we can factor k, solve the problem for each of its prime divisors, and then select the minimum of the answers.

More formally, let k_i- this is ith factor of the number $k$belongs to him in power p_i. We solve the problem for $k_i$using the above formula for $O(\log n)$; though we got an answer Ans_i. Then the answer for the composite $k$will be a minimum of values Ans_i / p_i.

Given that the factorization is performed in the simplest way $O(\sqrt{k})$, we obtain the asymptotic behavior of the final $O(\sqrt{k})$.

16. Ternary balanced system of value

Ternary balanced system of value - a non-standard positional numbering system. The base system is equal 3, but it differs from the usual ternary system that figures are $-1, 0, 1$. As used -1 for single digit is very uncomfortable, it usually takes some special notation. Conditions are denoted by minus one letter z.

For example, the number 5 in the ternary system is balanced as written $1zz$, and the number -5 - as $z11$. Ternary balanced system value allows you to record negative numbers without writing a single sign "minus". Balanced ternary system allows fractional numbers (for example, $1/3$ is written as 0.1).

Translation algorithm

Learn how to translate the numbers in a balanced ternary system.

To do this, we must first convert the number in the ternary system.

It is clear that now we have to get rid of the numbers 2, for which we note that $2 = 3 - 1$, ie we can replace the two in the current discharge on -1, while increasing the next (ie, to the left of it in a natural writing) on the discharge 1. If we move from right to left on the record and perform the above operation (in this case in some discharges can overflow more 3, in this case, of course, "reset" extra triple in the MSB), then arrive at a balanced ternary recording. As is easily seen, the same rule holds true for fractional numbers.

More gracefully above procedure can be described as follows. We take the number in the ternary value is added to it an infinite number $\ldots 11111.11111\ldots$, then each bit of the result subtract one (already without any hyphens).

Knowing now the translation algorithm from the usual ternary system in a balanced, we can easily implement the operations of addition, subtraction, and division - just reducing them to the corresponding operations on ternary unbalanced numbers.

16. Factorial calculation modulo

In some cases it is necessary to consider on some prime modulus p complex formulas, which may contain, including factorials.Here we consider the case when the module p is relatively small. It is clear that this problem is meaningful only when the factorials included in the numerator and denominator of the fractions. Indeed, factorial $p!$ and all subsequent vanish modulo p, but fractions of all the factors containing p, may be reduced, and the resulting expression has to be different from zero modulo p.

Thus, formally **problem** such. Required to compute $n!$ modulo a prime p, thus not taking into account all the multiple p factors included in the factorial. By learning to effectively compute a factorial, we can quickly calculate the value of a variety of combinatorial formulas (eg, Binomial coefficients).

Algorithm

Let us write down this "modified" factorial explicitly:

$$n!_{\%p} = \\ = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p} \cdot (2p+1) \cdot \ldots \cdot$$

$$\cdot(p^2-1)\cdot\underbrace{1}_{p^2}\cdot(p^2+1)\cdot\ldots\cdot n=$$
$$=1\cdot2\cdot3\cdot\ldots\cdot(p-2)\cdot(p-1)\cdot\cdot\underbrace{1}_{p}\cdot1\cdot2\cdot\ldots\cdot(p-1)\cdot\underbrace{2}_{2p}\cdot1\cdot2\cdot\ldots\cdot(p-1)\cdot\underbrace{1}_{p^2}\cdot$$
$$\cdot1\cdot2\cdot\ldots\cdot(n\%p)\pmod p.$$

When such a record shows that the "modified" factorial divided into several blocks of length p(the last block may have shorter), which are all identical, except for the last element:

$$n!_{\%p}=\underbrace{1\cdot2\cdot\ldots\cdot(p-2)\cdot(p-1)\cdot1}_{1st}\cdot\underbrace{1\cdot2\cdot\ldots\cdot(p-1)\cdot2}_{2nd}\cdot\ldots\underbrace{1\cdot2\cdot\ldots\cdot(p-1)\cdot1}_{p-th}\cdot\ldots\cdot$$
$$\cdot\underbrace{1\cdot2\cdot\ldots\cdot(n\%p)}_{tail}\pmod p.$$

Total count of blocks is easy - it's just $(p-1)!\bmod p$that you can find software or Theorem Wilson (Wilson) immediately find $(p-1)!\bmod p=p-1$. To multiply the common parts of blocks, the obtained value must be raised by the power mod $p$that can be done for $O(\log n)$operations (see. binary exponentiation , however, you can see that we actually erect minus one to some degree, and therefore the result of will always be either 1or $p-1$, depending on the parity index. Meaning in the last, incomplete block, too, can be calculated separately for $O(p)$. Only the last elements of the blocks, we consider them carefully:

$$n!_{\%p}=\underbrace{\ldots\cdot1}\cdot\underbrace{\ldots\cdot2}\cdot\underbrace{\ldots\cdot3}\cdot\ldots\cdot\underbrace{\ldots\cdot(p-1)}\cdot\underbrace{\ldots\cdot1}\cdot\underbrace{\ldots\cdot1}\cdot\underbrace{\ldots\cdot2}\ldots$$

And again we come to the "modified" factorial, but has a smaller dimension (as much as it was full of blocks, and they were $\lfloor n/p\rfloor$). Thus, the calculation of "modified" factorial$n!_{\%p}$we have reduced due $O(p)$to the computation operations already $(n/p)!_{\%p}$. Expanding this recurrence relation, we find that the depth of recursion is $O(\log_p n)$, total**asymptotic behavior of** the algorithm is obtained $O(p\log_p n)$.

Implementation

It is clear that the implementation is not necessary to use recursion explicitly: as tail recursion, it is easy to deploy in the cycle.

```
int factmod (int n, int p) {
	int res = 1;
	while (n > 1) {
		res = (res * ((n/p) % 2 ? p-1 : 1)) % p;
		for (int i=2; i<=n%p; ++i)
			res = (res * i) % p;
		n /= p;
	}
	return res % p;
}
```

This implementation works for $O(p\log_p n)$.

17. Enumeration of all subpatterns this mask

Bust subpatterns fixed mask

Dana bitmask m. Required to effectively sort out all its subpatterns, ie such masks s, which can be included only those bits that were included in the mask m.

Immediately look at the implementation of this algorithm, based on tricks with Boolean operations:

```
int s = m;
while (s > 0) {
       ... можно использовать s ...
       s = (s-1) & m;
}
```

or by using a more compact operator for:

```
for (int s=m; s; s=(s-1)&m)
       ... можно использовать s ...
```

The only exception for the two versions of the code - subpattern is equal to zero, will not be processed. Her treatment will have to take out of the loop, or use a less elegant design, for example:

```
for (int s=m; ; s=(s-1)&m) {
       ... можно использовать s ...
       if (s==0)  break;
}
```

Let us examine why the code above is really all subpatterns this mask, with no repetitions in O (number), and in descending order.

Suppose we have a current subpattern is s, and we want to move to the next subpattern. Subtract from the mask $s$unit, thus we remove the rightmost single bit, and all the bits to the right of him to put in 1. Next, remove all the "extra" one bits that are not included in the mask m, and therefore can not be included in the subpattern. Removal operation is performed bit $\&m$. As a result, we "cut off the" mask $s - 1$before the greatest importance that it may take, ie until the next subpattern following $s$in descending order.

Thus, the algorithm generates all subpatterns this mask in order strictly decreasing, spending on each transition on two elementary operations.

Particularly consider when $s = 0$. After performing $s - 1$we get the mask in which all bits are turned on (the bit representation of the number -1), and after removing the extra bit operation $(s - 1)\&m$ will not nothing but a mask m. Therefore, the mask $s = 0$should be careful - if time does not stop at zero mask, the algorithm may enter an infinite loop.

Through all the masks with their subpatterns. Qualification3^n

In many problems, especially in the dynamic programming masks are required to sort out all the masks, and masks for each - all subpatterns:

```
for (int m=0; m<(1<<n); ++m)
       for (int s=m; s; s=(s-1)&m)
```

```
... использование s и m ...
```

We prove that the inner loop will execute a total $O(3^n)$iterations.

Proof: 1 way . Consider ith bit. For him, generally speaking, there are exactly three ways: it is not included in the mask m(and therefore in the subpattern s); it is included in m, but is not included s; it enters $m$in s. Total bits n, so all the different combinations will 3^n, as required.

Proof: 2 way . Note that if the mask $m$has $k$included bits, it will have 2^ksubpatterns. Since the length of the mask $n$with the $k$bits is enabled C_n^k(see. "binomial coefficients"), then all combinations will be:

$$\sum_{k=0}^{n} C_n^k 2^k.$$

Calculate this amount. To do this, we note that it is nothing like the binomial theorem expansion in the expression $(1+2)^n$, ie 3^n, as required.

18. Primitive roots

Definition

A primitive root modulo n(Primitive root modulo n) is called a number $g$that all his power modulo n run through all the numbers relatively prime to n. Mathematically, it is formulated as follows: if $g$is a primitive root modulo n, then for any integer $a$such that $\gcd(a,n) = 1$, there exists an integer such $k$that $g^k \equiv a \pmod n$.

In particular, for the case of the simple $n$power of a primitive root run through all the numbers from 1 before $n-1$.

The existence of

Primitive root modulo $n$if and only if there $n$is a degree of odd prime or twice a prime power, and also in cases $n = 1$, $n = 2$, $n = 4$.

This theorem (which was fully proved by Gauss in 1801) is given here without proof.

Communication with the function of the Euler

Let g- a primitive root modulo n. Then we can show that the smallest number $k$for which $g^k \equiv 1 \pmod n$(ie k- index g(multiplicative order)), power $\phi(n)$. Moreover, the converse is also true, and this fact will be used by us in the following algorithm for finding a primitive root.

Furthermore, if the modulus $n$is at least one primitive root, the total of $\phi(\phi(n))$(because cyclic group with $k$elements has $\phi(k)$generators).

Algorithm for finding a primitive root

A naive algorithm would require for each test value of time to calculate all of its power and verify that they are all different. It's too slow algorithm, below we are using several well-known theorems of number theory, we obtain a faster algorithm.g $O(n)$

Above we present a theorem that if the smallest number $k$for which $g^k \equiv 1 \pmod n$(ie k- index g) as well $\phi(n)$, then g- a primitive root. Since for any number $a$relatively prime to n, performed Euler's theorem ($a^{\phi(n)} \equiv 1 \pmod n$), then to check that the $g$primitive root, it suffices to verify that for all numbers $d$smaller $\phi(n)$, satisfied $g^d \not\equiv 1 \pmod n$. However, while it is too slow algorithm.

From Lagrange's theorem that the rate of any number modulo na divisor $\phi(n)$. Thus, it suffices to show that for all proper divisors $d \mid \phi(n)$is performed $g^d \not\equiv 1 \pmod n$.This is a much faster algorithm, but we can go further.

We factor the number $\phi(n) = p_1^{a_1} \cdots p_s^{a_s}$. We prove that in the previous algorithm, it suffices to consider as da number of the form $\frac{\phi(n)}{p_i}$. Indeed, suppose that d- any proper subgroup $\phi(n)$. Then, obviously, there exists such $j$that $d \mid \frac{\phi(n)}{p_j}$, ie $d \cdot k = \frac{\phi(n)}{p_j}$. However, if $g^d \equiv 1 \pmod n$we would get:

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv \left(g^d\right)^k \equiv 1^k \equiv 1 \pmod n,$$

ie still among the numbers of the form $\frac{\phi(n)}{p_i}$ there would be something for which the condition is not satisfied, as required.

Thus, an algorithm for finding a primitive root. Find $\phi(n)$, factorize it. Now iterate through all the numbers $g = 1 \ldots n$, and for each consider all the values $g^{\frac{\phi(n)}{p_i}} \pmod n$.If the current $g$all these numbers were different from 1, this $g$is the desired primitive root.

Time of the algorithm (assuming that the number $\phi(n)$has $O\left(\log \phi(n)\right)$divisors, and exponentiation algorithm performed binary exponentiation , ie, for $O(\log n)$) power $O\left(\text{Ans} \cdot \log \phi(n) \cdot \log n\right)$, plus the time of the factorization $\phi(n)$, where Ans- the result, ie value of the unknown primitive root.

About the growth rate of the growth of primitive roots $n$are known only estimates. It is known that primitive roots - a relatively small amount. One of the famous assessment - assessment Shupe (Shoup), that, assuming the truth of the Riemann hypothesis, there is a primitive root $O(\log^6 n)$.

Implementation

Function powmod () performs a binary exponentiation modulo a function generator (int p) - is a primitive root modulo a prime p(factorization of $\phi(n)$the simplest algorithm is implemented for $O(\sqrt{\phi(n)})$).

To adapt this function to arbitrary p, just add the calculation of Euler's function in a variable phi, as well as weed out resnon-prime to n.

```
int powmod (int a, int b, int p) {
	int res = 1;
	while (b)
		if (b & 1)
			res = int (res * 1ll * a % p),  --b;
		else
```

```
                        a = int (a * 1ll * a % p),  b >>= 1;
        return res;
}

int generator (int p) {
        vector<int> fact;
        int phi = p-1,  n = phi;
        for (int i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        fact.push_back (i);
                        while (n % i == 0)
                                n /= i;
                }
        if (n > 1)
                fact.push_back (n);

        for (int res=2; res<=p; ++res) {
                bool ok = true;
                for (size_t i=0; i<fact.size() && ok; ++i)
                        ok &= powmod (res, phi / fact[i], p) != 1;
                if (ok)  return res;
        }
        return -1;
}
```

19. Discrete rooting

Problem of discrete root extraction (similar to the discrete logarithm problem) is as follows. According to n(n- prime) a, $k$you want to find all $x$satisfying:

$$x^k \equiv a \pmod n$$

An algorithm for solving

We will solve the problem by reducing it to the problem of discrete logarithm.

For this, we apply the concept of a primitive root modulo$n$. Let g- a primitive root modulo n (because n- simple, it exists). We can find it, as described in the corresponding article of the $O(\text{Ans} \cdot \log \phi(n) \cdot \log n) = O(\text{Ans} \cdot \log^2 n)$time plus the number factorization $\phi(n)$.

Discard from the case when $a = 0$- in this case immediately find the answer $x = 0$.

Since in this case (n- prime) any number of $1$to $n - 1$be represented in the form of a power of a primitive root, the root of the discrete problem, we can provide in the form of:

$$(g^y)^k \equiv a \pmod n$$

where

$$x \equiv g^y \pmod n$$

Trivial transformation we obtain:

$$(g^k)^y \equiv a \pmod n$$

Here is an unknown quantity y, so we came to the discrete logarithm problem in a pure form. This problem can be solved by an algorithm baby-step-giant-step Shanks for $O(\sqrt{n} \log n)$, ie find one of the solutions $y_0$of this equation (or find that this equation has no solutions).

Suppose we have found a solution to $y_0$this equation, then one of the solutions of the discrete root is $x_0 = g^{y_0} \pmod n$.

Finding all solutions, we know one of them

To completely solve the problem, we must learn one found $x_0 = g^{y_0} \pmod n$to find all the other solutions.

For this recall is the fact that a primitive root always has order $\phi(n)$(see. article about primitive root), ie the least degree g, giving the unit is $\phi(n)$. Therefore, the addition of the term with the exponent $\phi(n)$does not change anything:

$$x^k \equiv g^{y_0 \cdot k + l \cdot \phi(n)} \equiv a \pmod n \quad \forall\, l \in Z$$

Hence, all the solutions have the form:

$$x = g^{y_0 + \frac{l \cdot \phi(n)}{k}} \pmod n \quad \forall\, l \in Z$$

where $l$is chosen so that the fraction $\frac{l \cdot \phi(n)}{k}$was intact. To this fraction was intact, the numerator must be a multiple of the least common multiple $\phi(n)$and $k$where (recalling that the least common multiple of two numbers $\operatorname{lcm}(a,b) = \frac{a \cdot b}{\gcd(a,b)}$), we obtain:

$$x = g^{y_0 + i \frac{\phi(n)}{\gcd(k, \phi(n))}} \pmod n \quad \forall\, i \in Z$$

This is the final convenient formula, which gives a general view of all the solutions of the discrete root.

Implementation

We give a full implementation, including finding a primitive root, and finding the discrete logarithm and the withdrawal of all decisions.

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

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

```
int generator (int p) {
	vector<int> fact;
	int phi = p-1,  n = phi;
	for (int i=2; i*i<=n; ++i)
		if (n % i == 0) {
			fact.push_back (i);
			while (n % i == 0)
				n /= i;
		}
	if (n > 1)
		fact.push_back (n);

	for (int res=2; res<=p; ++res) {
		bool ok = true;
		for (size_t i=0; i<fact.size() && ok; ++i)
			ok &= powmod (res, phi / fact[i], p) != 1;
		if (ok)  return res;
	}
	return -1;
}

int main() {

	int n, k, a;
	cin >> n >> k >> a;
	if (a == 0) {
		puts ("1\n0");
		return 0;
	}

	int g = generator (n);

	int sq = (int) sqrt (n + .0) + 1;
	vector < pair<int,int> > dec (sq);
	for (int i=1; i<=sq; ++i)
		dec[i-1] = make_pair (powmod (g, int (i * sq * 1ll * k % (n -
1)), n), i);
	sort (dec.begin(), dec.end());
	int any_ans = -1;
	for (int i=0; i<sq; ++i) {
		int my = int (powmod (g, int (i * 1ll * k % (n - 1)), n) * 1ll
* a % n);
		vector < pair<int,int> >::iterator it =
			lower_bound (dec.begin(), dec.end(), make_pair (my,
0));
		if (it != dec.end() && it->first == my) {
			any_ans = it->second * sq - i;
			break;
		}
	}
	if (any_ans == -1) {
```

```
            puts ("0");
            return 0;
        }

        int delta = (n-1) / gcd (k, n-1);
        vector<int> ans;
        for (int cur=any_ans%delta; cur<n-1; cur+=delta)
            ans.push_back (powmod (g, cur, n));
        sort (ans.begin(), ans.end());
        printf ("%d\n", ans.size());
        for (size_t i=0; i<ans.size(); ++i)
            printf ("%d ", ans[i]);

}
```

20. Sieve of Eratosthenes with linear time work

Given a number n. You want to find **all the prime** in the interval $[2; n]$.

The classic way to solve this problem - **the sieve of Eratosthenes** . This algorithm is very simple, but it works for a while $O(n \log \log n)$.

Although at the moment we know a lot of algorithms working in sublinear time (ie $o(n)$), the algorithm described below is interesting for its **simplicity** - it is practically difficult to classical sieve of Eratosthenes.

In addition, the algorithm presented here as a "side effect" is actually computes **a factorization of all the numbers** in the interval $[2; n]$, which can be useful in many practical applications.

The drawback of the algorithm is driven by the fact that it uses **more memory** than the classical sieve of Eratosthenes: start requires an array of $n$numbers, while the classical sieve of Eratosthenes only enough $n$bits of memory (which is obtained in 32half).

Thus, the described algorithm should be applied only up to the order of numbers 10^7, no more.

Authorship algorithm apparently belongs Grice and Misra (Gries, Misra, 1978 - see. Bibliography at the end). (And, in fact, to call this algorithm "Sieve of Eratosthenes" correctly too the difference between these two algorithms.)

Description of the algorithm

Our goal - to find each number $i$of the segment in $[2; n]$its **minimal prime divisor** $lp[i]$.

In addition, we need to keep a list of found primes - let's call it an array $pr[]$.

Initially all values are $lp[i]$filled with zeros, which means that we are assuming all the numbers simple. In the course of the algorithm, this array will be gradually filled.

We now sort out the current number $i$of 2up n. We can have two cases:

- $lp[i] = 0$- This means that the number i- easy because for it had not found other subgroups.

  Therefore, it is necessary to assign $lp[i] = i$and add $i$to the end of the list $pr[]$.

- $lp[i] \neq 0$- This means that the current number i- a composite, and it is a minimal prime divisor $lp[i]$.

In both cases, then begins the process of **alignment of values** in the array $lp[]$: we will take the number, **multiples** i , and update their value $lp[]$. However, our goal - to learn how to do this so that in the end each of the value $lp[]$would be set more than once.

Argues that it is possible to do so. Consider the number of the form:

$$x_j = i \cdot p_j,$$

where the sequence p_j- it's simple, do not exceed $lp[i]$(just for this, we need to keep a list of all primes).

All properties of this kind places a new value $lp[x_j]$- obviously, it will be equal p_j.

Why such an algorithm is correct, and why it works in linear time - see. Below, but for now we present its implementation.

Implementation

Sieve performed until the number of the constant N.

```
const int N = 10000000;
int lp[N+1];
vector<int> pr;

for (int i=2; i<=N; ++i) {
	if (lp[i] == 0) {
		lp[i] = i;
		pr.push_back (i);
	}
	for (int j=0; j<(int)pr.size() && pr[j]<=lp[i] && i*pr[j]<=N; ++j)
		lp[i * pr[j]] = pr[j];
}
```

This implementation can speed up a little, getting rid of the vector pr(replacing it with a regular array with counter), as well as getting rid of duplicate multiplication in the nested loop for(which result is the product must be easy to remember in any variable).

Proof of correctness

We prove **the correctness of** the algorithm, ie he correctly puts all the values $lp[]$, and each of them will be set only once. This will imply that the algorithm works in linear time - as all the other steps of the algorithm is obviously working for $O(n)$.

For this, note that any number of **unique representation** of this form:i

$$i = lp[i] \cdot x,$$

where $lp[i]$- (as before) a minimal prime divisor of the number i, and the number $x$has no divisors less $lp[i]$, ie .:

$$lp[i] \le lp[x].$$

Now compare this to what makes our algorithm - it is actually for everyone $x$through all simple, for which it can multiply, ie Just prior $lp[x]$inclusive, to obtain the number of the above representation.

Consequently, the algorithm does take place for each composite number exactly once, putting it the correct value $lp[]$.

This means the correctness of the algorithm and the fact that it runs in linear time.

Time and the required memory

Although the asymptotic behavior of $O(n)$the asymptotic behavior of the best $O(n \log \log n)$ classical sieve of Eratosthenes, the difference between them is small. In practice this means a two-fold difference in the speed and optimized versions sieve of Eratosthenes and does not lose the algorithm given here.

Given the cost of memory, which requires the algorithm - array $lp[]$length $n$and an array of all the simple $pr[]$length about $n / \ln n$- this algorithm seems to be inferior to the classical sieve on all counts.

However, it makes the fact that the array $lp[]$, which is calculated by the algorithm, allows you to find the factorization of any number in the interval $[2; n]$of the time order of the size of the factorization. Moreover, the cost of one more additional array, you can do that in this factorization is not required of the division operation.

Knowledge of the factorization of all the numbers - very useful information for some tasks, and this algorithm is one of the few that allow you to look for it in linear time.

Literature

- David Gries, Jayadev Misra. **A Linear Sieve Algorithm for Finding Prime Numbers** [1978]

21. test BPSW the simplicity of numbers

Introduction

Algorithm BPSW - this is a test of the 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 was tested on all numbers 10 to ^{15} . Furthermore, counterexample to find using the PRIMO (cm. [6]), based on simple test using elliptic curves. The program worked for three years, did not find any counterexample, based on which Martin has 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 is 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.

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

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 composite 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 simple. 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 pseudosimple still have 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), should also 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 implements 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 this article. 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 increase the b
	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 performed with this 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, the bust D 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 difficult.

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 penalty of at 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 in practice 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 interval [1; 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 observation (see. 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:

$$U_M = 0 \pmod N$$

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 algorithm Lucas.

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:

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

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

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

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

It is understood that **it suffices to calculate U_E V and $_E$** , and all subsequent Multipliers $V_{2E} V_{4E} \ldots V_{2^{T-2}E} V_{2^{T-1}E}$ can **already receive from them** .

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:

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

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

These formulas are proved 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 and V_E .

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 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 preceding terms by 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 formula. At the 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} , ... $V_{2^{T-2,}E}$, $V_{2^{T-1}E}$. 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 can elaborate on its parameters D, P, Q, one of the ways to obtain and which is the algorithm of Selfridge.

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. This 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, 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 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 some of the 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 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 - 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 $T^2 / \text{Log } (T^K)$. 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. 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 $(Q_1, Q_3) = 1$ and $Q_1 Q_3 ? e^T$ .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

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

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 is also a strong base 2 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 - pseudosimple 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^2} . Now, if 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 internal type - 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**	
1	10^5		8.1	4.5	
10^6	10^6 10^5		12.8	6.8	
10^9	10^9 10^5		28.4	12.6	
10^{12}	10^{12} 10^5		41.5	16.5	
10^{15}	10^{15} 10^5		66.7	24.4	

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	
1	10^5		42.9	40.8	
10^6	10^6 10^5		75.0	76.4	
10^9	10^9 10^5		186.5	188.5	
10^{12}	10^{12} 10^5		288.3	288.3	
10^{15}	10^{15} 10^5		485.6	489.1	

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

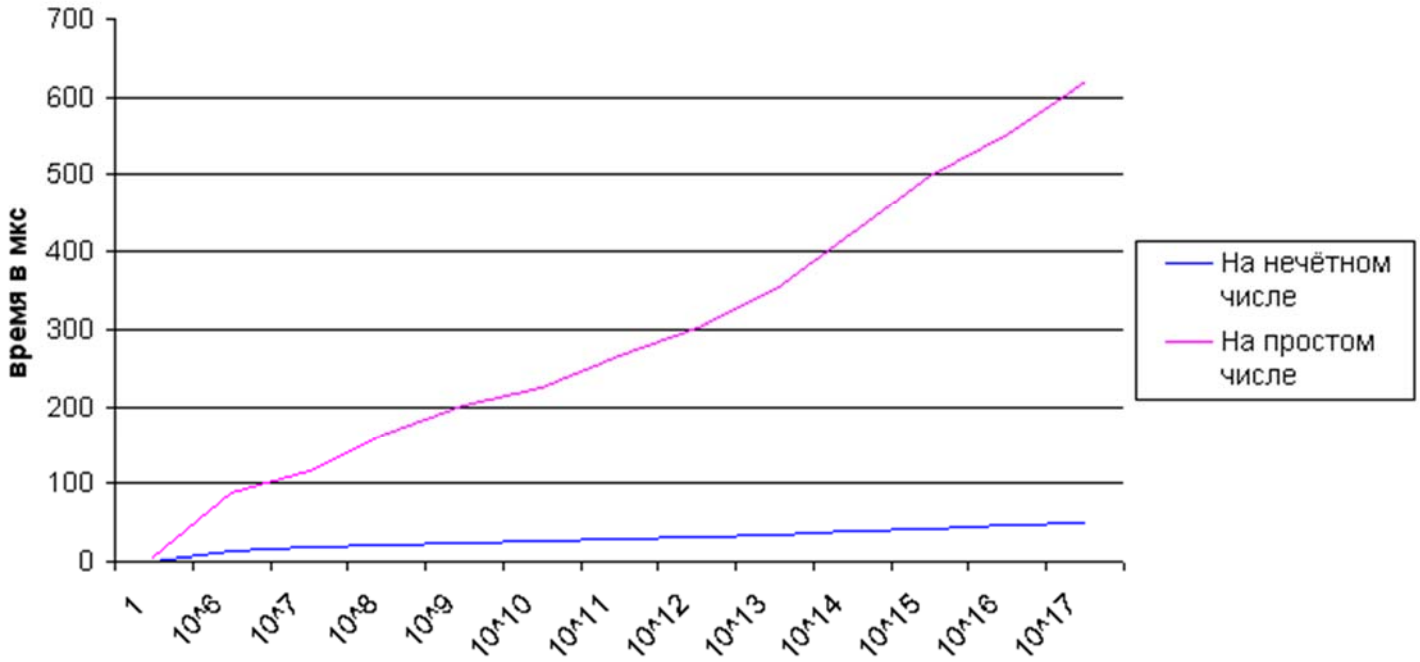
Now, when we chose the limit of trivial enumeration, you can more accurately test the speed at various intervals.

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 type int64 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/1887/2136/1/346_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:

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)

22. Efficient algorithms for factorization

Here are the implementation of several factorization algorithms, each of which individually may not work as quickly or very slowly, but together they provide a very fast method.

Descriptions of these methods are given, the more that they are well described on the Internet.

Method Pollard p-1

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_p_1 (T n)
{
	// Algorithm parameters significantly affect the performance and the
quality of search
	const T b = 13;
	const T q [] = {2, 3, 5, 7, 11, 13};

	// Several attempts algorithm
	T a = 5% n;
	for (int j = 0; j <10; j ++)
	{

		// Looking for is a, which is relatively prime to n
		while (gcd (a, n)! = 1)
		{
			mulmod (a, a, n);
			a + = 3;
			a% = n;
		}

		// Calculate a ^ M
		for (size_t i = 0; i <sizeof q / sizeof q [0]; i ++)
		{
			T qq = q [i];
			T e = (T) floor (log ((double) b) / log ((double) qq));
			T aa = powmod (a, powmod (qq, e, n), n);
```

```
                    if (aa == 0)
                          continue;

                    // Check not found the answer
                    T g = gcd (aa-1, n);
                    if (1 <g && g <n)
                          return g;
             }

      }

      // If nothing found
      return 1;

}
```

Pollard's method "Ro"

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_rho (T n, unsigned iterations_count = 100000)
{
      T
            b0 = rand ()% n,
            b1 = b0,
            g;
      mulmod (b1, b1, n);
      if (++ b1 == n)
            b1 = 0;
      g = gcd (abs (b1 - b0), n);
      for (unsigned count = 0; count <iterations_count && (g == 1 || g ==
n); count ++)
      {
            mulmod (b0, b0, n);
            if (++ b0 == n)
                  b0 = 0;
            mulmod (b1, b1, n);
            ++ B1;
            mulmod (b1, b1, n);
            if (++ b1 == n)
                  b1 = 0;
            g = gcd (abs (b1 - b0), n);
      }
      return g;
}
```

Bent method (modification of Pollard "Ro")

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
```

```
T pollard_bent (T n, unsigned iterations_count = 19)
{
	T
		b0 = rand ()% n,
		b1 = (b0 * b0 + 2)% n,
		a = b1;
	for (unsigned iteration = 0, series_len = 1; iteration
<iterations_count; iteration ++, series_len * = 2)
	{
		T g = gcd (b1-b0, n);
		for (unsigned len = 0; len <series_len && (g == 1 && g == n);
len ++)
		{
			b1 = (b1 * b1 + 2)% n;
			g = gcd (abs (b1-b0), n);
		}
		b0 = a;
		a = b1;
		if (g! = 1 && g! = n)
			return g;
	}
	return 1;
}
```

Pollard's method of Monte Carlo

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_monte_carlo (T n, unsigned m = 100)
{
	T b = rand ()% (m-2) + 2;

	static std :: vector <T> primes;
	static T m_max;
	if (primes.empty ())
		primes.push_back (3);
	if (m_max <m)
	{
		m_max = m;
		for (T prime = 5; prime <= m; ++++ prime)
		{
			bool is_prime = true;
			for (std :: vector <T> :: const_iterator iter =
primes.begin (), end = primes.end ();
				iter! = end; ++ Iter)
			{
				T div = * iter;
				if (div * div> prime)
					break;
				if (prime% div == 0)
				{
```

```
                                is_prime = false;
                                break;
                        }
                }
                if (is_prime)
                        primes.push_back (prime);
        }
    }

    T g = 1;
    for (size_t i = 0; i <primes.size () && g == 1; i ++)
    {
        T cur = primes [i];
        while (cur <= n)
                cur * = primes [i];
        cur / = primes [i];
        b = powmod (b, cur, n);
        g = gcd (abs (b-1), n);
        if (g == n)
                g = 1;
    }

    return g;
}
```

Method Farm

This wide method, but it can be very slow if the number is small divisors.

Therefore, it should run only after all other methods.

```
template <class T, class T2>
T ferma (const T & n, T2 unused)
{
    T2
        x = sq_root (n),
        y = 0,
        r = x * x - y * y - n;
    for (;;)
        if (r == 0)
                return x! = y? xy: x + y;
        else
                if (r> 0)
                {
                        r - = y + y + 1;
                        ++ Y;
                }
                else
                {
                        r + = x + x + 1;
                        ++ X;
                }
}
```

Trivial division

This basic method is useful to immediately handle numbers with very small divisors.

```
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m)
{

	// First check the trivial cases
	if (n == 2 || n == 3)
		return 1;
	if (n <2)
		return 0;
	if (even (n))
		return 2;

	// Generate a simple 3 to m
	T2 pi;
	const vector <T2> & primes = get_primes (m, pi);

	// Divisible by all prime
	for (std :: vector <T2> :: const_iterator iter = primes.begin (),
end = primes.end ();
		iter! = end; ++ Iter)
	{
		const T2 & div = * iter;
		if (div * div> n)
			break;
		else
			if (n% div == 0)
				return div;
	}

	if (n <m * m)
		return 1;
	return 0;

}
```

Putting it all together

Combine all the methods in the same function.

Also, the function uses the simplicity of the test, otherwise the factorization algorithms can work for very long. For example, you can select a test BPSW (read the article on BPSW).

```
template <class T, class T2>
void factorize (const T & n, std :: map <T, unsigned> & result, T2 unused)
{
	if (n == 1)
		;
	else
		// Check whether the number is not prime
		if (isprime (n))
			++ Result [n];
		else
```

```
                    // If the number is small enough that it expand the
simple search
                    if (n <1000 * 1000)
                    {
                          T div = prime_div_trivial (n, 1000);
                          ++ Result [div];
                          factorize (n / div, result, unused);
                    }
                    else
                    {
                          // Number of large, run it factorization
algorithms
                          T div;
                          // First go fast algorithms Pollard
                          div = pollard_monte_carlo (n);
                          if (div == 1)
                                div = pollard_rho (n);
                          if (div == 1)
                                div = pollard_p_1 (n);
                          if (div == 1)
                                div = pollard_bent (n);
                          // Will run 100% algorithm Farm
                          if (div == 1)
                                div = ferma (n, unused);
                          // Recursively Point Multipliers
                          factorize (div, result, unused);
                          factorize (n / div, result, unused);
                    }
}
```

Appendix

Download [5k] source program that uses all of these methods and test factorization BPSW on simplicity.

23. Fast Fourier transform of the O (N log N). Application to the multiplication of two polynomials or long numbers

Here we consider an algorithm which allows to multiply two polynomials of length n during $O(n \log n)$that time much better $O(n^2)$achievable trivial multiplication algorithm. It is clear that the multiplication of two long numbers can be reduced to a multiplication of polynomials, so the two long numbers can also multiply during $O(n \log n)$.

The invention Fast Fourier Transform attributed Cooley (Coolet) and Taki (Tukey) - 1965 Actually FFT repeatedly invented before, but its importance was not fully realized until the advent of modern computers. Some researchers credited with the discovery of the FFT Runge (Runge) and König (Konig) in 1924 Finally, the discovery of this method is attributed to more Gauss (Gauss) in 1805

Discrete Fourier Transform (DFT)

Suppose there is a polynomial $n$of degree th:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Without loss of generality, we can assume that $n$is a power of 2 If in fact $n$not a power of 2, then we just add the missing coefficients by setting them equal to zero.

Of the theory of functions of a complex variable is known that the complex roots $n$of unity th exists exactly n. Denote these roots by $w_{n,k}, k = 0 \ldots n-1$then known that $w_{n,k} = e^{i\frac{2\pi k}{n}}$. In addition, one of these roots $w_n = w_{n,1} = e^{i\frac{2\pi}{n}}$ (called the principal value of the root of nth roots of unity) is such that all other roots are his powers: $w_{n,k} = (w_n)^k$.

Then **the discrete Fourier transform (DFT)** (discrete Fourier transform, DFT) of the polynomial $A(x)$(or, equivalently, the DFT of the vector of its coefficients $(a_0, a_1, \ldots, a_{n-1})$) are the values of the polynomial at the points $x = w_{n,k}$, ie, is a vector:

$$\begin{aligned}&\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) = (y_0, y_1, \ldots, y_{n-1}) = (A(w_{n,0}), A(w_{n,1}), \ldots, A(w_{n,n-1})) = \\ &= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1})).\end{aligned}$$

Defined similarly and **inverse discrete Fourier transform** (InverseDFT). Inverse DFT to the vector of a polynomial $(y_0, y_1, \ldots y_{n-1})$- is the vector of coefficients of the polynomial $(a_0, a_1, \ldots, a_{n-1})$:

$$\mathrm{InverseDFT}(y_0, y_1, \ldots, y_{n-1}) = (a_0, a_1, \ldots, a_{n-1}).$$

Thus, if the direct proceeds from the DFT coefficients of the polynomial to its values in the complex roots of nth roots of unity, the inverse DFT - on the contrary, from the values of the coefficients of the polynomial recovers.

The use of DFT for fast multiplication of polynomials

Given two polynomials $A$and B. Calculate the DFT for each of them: $\mathrm{DFT}(A)$and $\mathrm{DFT}(B)$- two vector values of polynomials.

Now, what happens when you multiply polynomials? Obviously, in each point of their values are simply multiplied, that is,

$$(A \times B)(x) = A(x) \times B(x).$$

But it does mean that if we multiply the vector $\mathrm{DFT}(A)$and $\mathrm{DFT}(B)$, by simply multiplying each element of a vector to the corresponding element of another vector, then we get nothing but a DFT of a polynomial $A \times B$:

$$\mathrm{DFT}(A \times B) = \mathrm{DFT}(A) \times \mathrm{DFT}(B).$$

Finally, applying the inverse DFT, we obtain:

$$A \times B = \mathrm{InverseDFT}(\mathrm{DFT}(A) \times \mathrm{DFT}(B)),$$

where, again, right under the product of two DFT mean pairwise products of the elements of the vectors. This work obviously requires to compute only $O(n)$operations. Thus, if we learn to calculate the DFT and inverse DFT of the time $O(n \log n)$, then the product of two polynomials (and, consequently, the two long numbers) we can find for the same asymptotic behavior.

It should be noted that, firstly, the result should be two polynomials of degree one (simply adding the coefficients of one of these zeros). Secondly, as a result of the product of two polynomials of degree $n$polynomial of degree obtained $2n-1$, so that the result is correct, you need to double the pre-degree of each polynomial (again, adding to their coefficients equal to zero).

Fast Fourier Transform

Fast Fourier Transform (fast Fourier transform) - a method to calculate the DFT of the time $O(n \log n)$. This method relies on the properties of the complex roots of unity (namely, that some degree of roots give other roots).

The basic idea is to divide the FFT coefficient vector into two vectors, the recursive computation of the DFT for them, and the union results in a single FFT.

Thus, suppose there is a polynomial $A(x)$of degree n, where n- a power of two, and $n > 1$:

$$A(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}.$$

Divide it into two polynomials, one - with even and the other - with the odd coefficients:

$$A_0(x) = a_0 x^0 + a_2 x^1 + \ldots + a_{n-2} x^{n/2-1},$$
$$A_1(x) = a_1 x^0 + a_3 x^1 + \ldots + a_{n-1} x^{n/2-1}.$$

It is easy to verify that:

$$A(x) = A_0(x^2) + x A_1(x^2). \qquad (1)$$

The polynomials $A_0$and A_1have twice lower degree than the polynomial A. If we can in linear time from the calculated $\mathrm{DFT}(A_0)$and $\mathrm{DFT}(A_1)$calculate $\mathrm{DFT}(A)$, then we obtain the desired fast Fourier transform algorithm (since it is a standard chart of "divide and conquer", and it is known asymptotic estimate $O(n \log n)$).

So, suppose we have calculated the vector $\{y_k^0\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_0)$and $\{y_k^1\}_{k=0}^{n/2-1} = \mathrm{DFT}(A_1)$. Let us find the expression for $\{y_k\}_{k=0}^{n-1} = \mathrm{DFT}(A)$.

Firstly, recalling (1), we immediately obtain the values for the first half of the coefficients:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1.$$

For the second half of the coefficients after transformation also get a simple formula:

$$y_{k+n/2} = A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2} A_1(w_n^{2k+n}) = A_0(w_n^{2k} w_n^n) + w_n^k w_n^{n/2} A_1(w_n^{2k} w_n^n) =$$
$$= A_0(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1.$$

(Here we have used (1), as well as identities $w_n^n = 1$, $w_n^{n/2} = -1$.)

So as a result we got the formula to calculate the total vector $\{y_k\}$:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1,$$
$$y_{k+n/2} = y_k^0 - w_n^k y_k^1, \quad k = 0 \ldots n/2-1.$$

(These formulas, ie two formulas of the form $a + bc$, and $a - bc$are sometimes called "butterfly transformation" ("butterfly operation"))

Thus, we finally built the FFT algorithm.

Inverse FFT

So, let a vector $(y_0, y_1, \ldots, y_{n-1})$- the values of a polynomial $A$of degree $n$at points $x = w_n^k$. Need to recover the coefficients $(a_0, a_1, \ldots, a_{n-1})$of the polynomial. This well-known problem is called **interpolation** , for this task, there are some common algorithms for the solution, but in this case will be obtained by a very simple algorithm (a simple fact that it is virtually identical to the FFT).

DFT, we can write, according to his definition, in matrix form:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

The vector $(a_0, a_1, \ldots, a_{n-1})$can be found by multiplying the vector $(y_0, y_1, \ldots, y_{n-1})$by the inverse matrix to the matrix, which stands on the left (which, incidentally, is called a Vandermonde matrix):

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

A direct check shows that this inverse matrix is as follows:

$$\frac{1}{n} \begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^{-1} & w_n^{-2} & w_n^{-3} & \cdots & w_n^{-(n-1)} \\ w_n^0 & w_n^{-2} & w_n^{-4} & w_n^{-6} & \cdots & w_n^{-2(n-1)} \\ w_n^0 & w_n^{-3} & w_n^{-6} & w_n^{-9} & \cdots & w_n^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{-(n-1)} & w_n^{-2(n-1)} & w_n^{-3(n-1)} & \cdots & w_n^{-(n-1)(n-1)} \end{pmatrix}.$$

Thus, we obtain:

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j w_n^{-kj}.$$

Comparing it with the formula for y_k:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

we observe that these two tasks are not real, so that the coefficients $a_k$can be found in the same algorithm "divide and rule" as a direct FFT, but instead w_n^kshould be used everywhere w_n^{-k}, and every element of the result should be divided into n.

Thus, the calculation of the inverse DFT is not very different from the direct computation of the DFT, and it can also be performed during the time $O(n \log n)$.

Implementation

Consider a simple recursive **implementation of the FFT** and IFFT, implement them in a single function, as the difference between direct and inverse FFT minimal. For storing complex numbers using the standard in C ++ STL type complex (defined in the header file <complex>).

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
	int n = (int) a.size();
	if (n == 1)  return;

	vector<base> a0 (n/2),  a1 (n/2);
	for (int i=0, j=0; i<n; i+=2, ++j) {
		a0[j] = a[i];
		a1[j] = a[i+1];
	}
	fft (a0, invert);
	fft (a1, invert);

	double ang = 2*PI/n * (invert ? -1 : 1);
	base w (1),  wn (cos(ang), sin(ang));
	for (int i=0; i<n/2; ++i) {
		a[i] = a0[i] + w * a1[i];
		a[i+n/2] = a0[i] - w * a1[i];
		if (invert)
			a[i] /= 2,  a[i+n/2] /= 2;
		w *= wn;
	}
}
```

In the argument of $\mathrm{a}$the function is passed the input vector of coefficients in the same and it will contain the result. Argument invertshows direct or inverse DFT should be calculated. Inside the function first checks if the vector length $\mathrm{a}$is equal to one, there is nothing else to do - he is the answer. Otherwise, the vector $\mathrm{a}$is split into two vectors $\mathrm{a0}$and $\mathrm{a1}$for which recursively calculated DFT. Then we calculate the value of w_n, and the plant variable $w$containing the current degree w_n. Then calculated the elements of the result of the DFT on the above formulas.

If the flag is specified $\mathrm{invert} = \mathrm{true}$, then $w_n$replaced by w_n^{-1}, and each element of the result is divided by 2 (given that these dividing by 2 will take place in each level of recursion, the result just happens that all the elements on the share n).

Then the function for **multiplying two polynomials** is as follows:

```
void multiply (const vector<int> & a, const vector<int> & b, vector<int> &
res) {
	vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
	size_t n = 1;
	while (n < max (a.size(), b.size()))  n <<= 1;
	n <<= 1;
	fa.resize (n),  fb.resize (n);

	fft (fa, false),  fft (fb, false);
	for (size_t i=0; i<n; ++i)
		fa[i] *= fb[i];
	fft (fa, true);

	res.resize (n);
	for (size_t i=0; i<n; ++i)
		res[i] = int (fa[i].real() + 0.5);
}
```

This feature works with polynomials with integer coefficients (although, of course, in theory there is nothing stopping her work with fractional coefficients). However, it appears the problem of a large error in the calculation of DFT: the error can be significant, so the rounding of the best most reliable way - by adding 0.5 and then rounding down (**note** : this will not work properly for negative numbers, if any, may appear in your application).

Finally, the function for **multiplying two long numbers** is practically no different from the function for multiplying polynomials. The only feature - that after the multiplication of numbers as polynomials should be normalized, ie, perform all transfers bits:

```
	int carry = 0;
	for (size_t i=0; i<n; ++i) {
		res[i] += carry;
		carry = res[i] / 10;
		res[i] %= 10;
	}
```

(Since the length of the product of two numbers is never surpass the total length of the number, the size of the vector reswill be enough to fulfill all the transfers.)

Improved execution: computing "on the spot" without additional memory

To increase the efficiency abandon recursion explicitly. In the above recursive implementation, we explicitly separated the vector $\mathrm{a}$into two vectors - elements on the even positions attributed to the same time to create a vector, and on the odd - to another. However, if we re-ordered elements in a certain way, the need for creating temporary vectors would then be eliminated (ie, all the calculations we could produce "in situ", right in the vector a).

Note that the first level of recursion elements, junior (first) position bits are zero, refer to the vector a_0, and the least significant bits of positions which are equal to one - to the vector a_1. At the second level of recursion is done the same thing, but for the second bit, etc. So if we are in the position $i$of each

element $a[i]$invert the bit order, and reorder elements of the array $a$according to the new indexes, we obtain the desired order (it is called a **bitwise inverse permutation** (bit-reversal permutation)).

For example, in $n = 8$this order is as follows:

$$a = \Big\{ \big[(a_0, a_4), (a_2, a_6)\big], \big[(a_1, a_5), (a_3, a_7)\big] \Big\}.$$

Indeed, on the first level of recursion (surrounded by curly braces) conventional recursive algorithm is a division of the vector into two parts: $[a_0, a_2, a_4, a_6]$and $[a_1, a_3, a_5, a_7]$. As we can see, in the bitwise inverse permutation, this corresponds to a separation of the vector into two halves: the first $n/2$element and the last $n/2$element. Then there is a recursive call on each half; let the resulting DFT of each of them was returned in place of the elements themselves (ie, the first and second halves of the vector a, respectively):

$$a = \Big\{ \big[y_0^0,\ y_1^0,\ y_2^0,\ y_3^0\big], \big[y_0^1,\ y_1^1,\ y_2^1,\ y_3^1\big] \Big\}.$$

Now we have to perform the union of two into one DFT for the vector. But the elements stood out so well, and that the union can be performed directly in the array. Indeed, we take the elements y_0^0 and $y_0^1$is applicable to them transform butterflies, and the result is put in their place - and this place and would thereby and which should have been received:

$$a = \Big\{ \big[y_0^0 + w_n^0 y_0^1,\ y_1^0,\ y_2^0,\ y_3^0\big], \big[y_0^0 - w_n^0 y_0^1,\ y_1^1,\ y_2^1,\ y_3^1\big] \Big\}.$$

Similarly, we apply the transformation to a butterfly $y_1^0$and $y_1^1$the result put in their place, etc. As a result, we obtain:

$$a = \Big\{ \big[y_0^0 + w_n^0 y_0^1,\ y_1^0 + w_n^1 y_1^1,\ y_2^0 + w_n^2 y_2^1,\ y_3^0 + w_n^3 y_3^1\big],$$
$$\big[y_0^0 - w_n^0 y_0^1,\ y_1^0 - w_n^1 y_1^1,\ y_2^0 - w_n^2 y_2^1,\ y_3^0 - w_n^3 y_3^1\big] \Big\}.$$

Ie We got exactly the desired DFT of the vector a.

We describe the process of calculating the DFT on the first level of recursion, but it is clear that the same arguments hold for all other levels of recursion. Thus, **after applying the bitwise inverse permutation to calculate the DFT can be on the spot** , without any additional arrays.

But now you can **get rid of the recursion** explicitly. So, we applied bitwise inverse permutation elements. Now do all the work being done by the lower level of recursion, ievector $a$divide into pairs of elements for each applicable transformation of butterflies, resulting in the vector $a$will be the results of the lower level of recursion. In the next step the vector divide $a$at quadruple elements applied to each butterfly transform, thus obtaining a DFT for every fours. And so on, finally, the last step, we received the results of the DFT for the two halves of the vector a, it is applicable to the transformation of butterflies and obtain the DFT for the vector a.

Thus, the implementation of:

```
typedef complex<double> base;
```

```
int rev (int num, int lg_n) {
	int res = 0;
	for (int i=0; i<lg_n; ++i)
		if (num & (1<<i))
			res |= 1<<(lg_n-1-i);
	return res;
}

void fft (vector<base> & a, bool invert) {
	int n = (int) a.size();
	int lg_n = 0;
	while ((1 << lg_n) < n)  ++lg_n;

	for (int i=0; i<n; ++i)
		if (i < rev(i,lg_n))
			swap (a[i], a[rev(i,lg_n)]);

	for (int len=2; len<=n; len<<=1) {
		double ang = 2*PI/len * (invert ? -1 : 1);
		base wlen (cos(ang), sin(ang));
		for (int i=0; i<n; i+=len) {
			base w (1);
			for (int j=0; j<len/2; ++j) {
				base u = a[i+j],  v = a[i+j+len/2] * w;
				a[i+j] = u + v;
				a[i+j+len/2] = u - v;
				w *= wlen;
			}
		}
	}
	if (invert)
		for (int i=0; i<n; ++i)
			a[i] /= n;
}
```

Initially, a vector $a$is used bitwise inverse permutation, for which calculated the number of significant bits ($\lg n$) including n, for each position $i$is the corresponding position, which has a bit write bit representation of the number $i$recorded in the reverse order. If as a result of the resulting position was more i, the elements in these two positions need to be exchanged (unless this condition, each couple will exchange twice, and in the end nothing will happen).

Then, the $\lg n - 1$algorithm steps, on kth of which ($k = 2 \dots \lg n$) are computed for the DFT block length 2^k. For all of these units will be the same value of a primitive root w_{2^k}, and is stored in a variable wlen. Cycle through $i$iterated by block, and invested in it by the cycle $j$applies the transformation to all elements of the butterfly unit.

You can perform further **optimizations reverse bits** . In the previous implementation, we obviously took over all bits of the number, simultaneously building bitwise inverted number. However, reversing the bits can be performed in a different way.

For example, suppose that j- already counted the number equal to the inverse permutation of bits i . Then, during the transition to the next number $i + 1$we have and the number of $j$add one, but add it to this "inverted" value. In a conventional binary value add one - so remove all units standing at the end of the number (ie, a group of younger units), and put the unit in front of them. Accordingly, in the "inverted" system we have to go the bit number, starting with the oldest, and while there are one, delete them and move on to the next bit; when will meet the first zero bit, put it in the unit and stop.

Thus, we obtain a realization:

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
	int n = (int) a.size();

	for (int i=1, j=0; i<n; ++i) {
		int bit = n >> 1;
		for (; j>=bit; bit>>=1)
			j -= bit;
		j += bit;
		if (i < j)
			swap (a[i], a[j]);
	}

	for (int len=2; len<=n; len<<=1) {
		double ang = 2*PI/len * (invert ? -1 : 1);
		base wlen (cos(ang), sin(ang));
		for (int i=0; i<n; i+=len) {
			base w (1);
			for (int j=0; j<len/2; ++j) {
				base u = a[i+j],  v = a[i+j+len/2] * w;
				a[i+j] = u + v;
				a[i+j+len/2] = u - v;
				w *= wlen;
			}
		}
	}
	if (invert)
		for (int i=0; i<n; ++i)
			a[i] /= n;
}
```

Additional optimization

We give a list of other optimizations, which together can significantly speed up the preceeding "improved" implementation:

- **Predposchitat reverse bits** for all numbers in a global table. It is especially easy when the size $n$is the same for all calls.

This optimization becomes noticeable when a large number of calls $fft()$. However, the effect of it can be seen even in the three calls (three calls - the most common situation, ie when it is required once to multiply two polynomials).

- Refuse to use vector(**go to normal arrays**).

  The effect of this depends upon the particular compiler, but typically it is present and approximately 10% -20%.

- Predposchitat **all power** numbers $wlen$. In fact, in this cycle of the algorithm repeatedly made the passage in all degrees of $wlen$on $0$to $len/2-1$:

```
for (int i=0; i<n; i+=len) {
        base w (1);
        for (int j=0; j<len/2; ++j) {
                [...]
                w *= wlen;
        }
}
```

  Accordingly, before this cycle we can predposchitat some array all the required power, and thus get rid of unnecessary multiplications in the nested loop.

  Tentative acceleration - 5-10%.

- Get rid of **references to arrays in the indices** , instead use pointers to the current array elements, promoting their right to 1 at each iteration.

  At first glance, optimizing compilers should be able to cope with this, but in practice it turns out that the replacement of references to arrays $a[i+j]$and $a[i+j+len/2]$pointers to accelerate the program in popular compilers. Prize is 5-10%.

- **Abandon the standard type of complex numbers** $\mathrm{complex}$, rewriting it for your own implementation.

  Again, this may seem surprising, but even in modern compilers benefit from such a rewriting can be up to several tens of percent! This indirectly confirms a widespread assertion that compilers perform worse with sample data types, optimizing work with them much worse than non-formulaic types.

- Another useful optimization is the **cut-off length** : when the length of the working unit becomes small (say, 4), to calculate the DFT for it "manually". If you paint these cases in the form of explicit formulas for a length equal to $4/2$, the values of the sine-cosine take integer values, due to which you can get a speed boost for another few tens of percent.

  Here we present the realization of the described improvements (except the last two items which lead to the proliferation of codes):

```
int rev[MAXN];
base wlen_pw[MAXN];

void fft (base a[], int n, bool invert) {
       for (int i=0; i<n; ++i)
              if (i < rev[i])
                     swap (a[i], a[rev[i]]);

       for (int len=2; len<=n; len<<=1) {
```

```
                double ang = 2*PI/len * (invert?-1:+1);
                int len2 = len>>1;

                base wlen (cos(ang), sin(ang));
                wlen_pw[0] = base (1, 0);
                for (int i=1; i<len2; ++i)
                        wlen_pw[i] = wlen_pw[i-1] * wlen;

                for (int i=0; i<n; i+=len) {
                        base t,
                                *pu = a+i,
                                *pv = a+i+len2,
                                *pu_end = a+i+len2,
                                *pw = wlen_pw;
                        for (; pu!=pu_end; ++pu, ++pv, ++pw) {
                                t = *pv * *pw;
                                *pv = *pu - t;
                                *pu += t;
                        }
                }
        }

        if (invert)
                for (int i=0; i<n; ++i)
                        a[i] /= n;
}

void calc_rev (int n, int log_n) {
        for (int i=0; i<n; ++i) {
                rev[i] = 0;
                for (int j=0; j<log_n; ++j)
                        if (i & (1<<j))
                                rev[i] |= 1<<(log_n-1-j);
        }
}
```

On common compilers, this implementation faster than the previous "improved" version of the 2-3.

The discrete Fourier transform of the modular arithmetic

At the heart of the discrete Fourier transform are complex numbers, roots nth roots of unity. To effectively calculate it used features such roots as the existence of $n$different roots, forming a group (ie, the degree of the same root - always another square, among them there is one element - the generator of the group, called a primitive root).

But the same is true of the roots of nth roots of unity in modular arithmetic. Well, not for any module $p$there exists na variety of roots of unity, but these modules do exist. Is still important for us to find among them a primitive root, ie .:

$$(w_n)^n = 1 \pmod p,$$
$$(w_n)^k \neq 1 \pmod p, \quad 1 \leq k < n.$$

All other $n-1$roots nth roots of unity in modulus $p$can be obtained as a power of a primitive root w_n (as in the complex case).

For use in the fast Fourier transform algorithm we needed to primivny root existed for some n, a power of two, as well as all the lesser degrees. And if in the complex case, there was a primitive root for anyone n, in the case of modular arithmetic is generally not the case. However, note that if $n = 2^k$, ie kStar power of two, the modulo $m = 2^{k-1}$have:

$$(w_n^2)^m = (w_n)^n = 1 \pmod p,$$
$$(w_n^2)^k = w_n^{2k} \neq 1 \pmod p, \qquad 1 \leq k < m.$$

Thus, if w_n- a primitive root of $n = 2^k$th degree of unity, then w_n^2- a primitive root of 2^{k-1}th roots of unity. Therefore, all powers of two smaller n, the primitive roots of the desired extent also exist and can be calculated as the corresponding power w_n.

The final touch - for the inverse DFT, we used instead of $w_n$the inverse element: w_n^{-1}. But modulo a prime $p$element inverse also always be found.

Thus, all the required properties are observed in the case of modular arithmetic, provided that we have chosen some rather large unit $p$and found it to be a primitive root nth roots of unity.

For example, you can take the following values: module $p = 7340033$, $w_{2^{20}} = 5$. If this module is not enough to find another pair, you can use the fact that for the modules of the form $c2^k + 1$(but still necessarily simple) there is always a primitive cube root $2^k$of unity.

```
const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1<<20;

void fft (vector<int> & a, bool invert) {
	int n = (int) a.size();

	for (int i=1, j=0; i<n; ++i) {
		int bit = n >> 1;
		for (; j>=bit; bit>>=1)
			j -= bit;
		j += bit;
		if (i < j)
			swap (a[i], a[j]);
	}

	for (int len=2; len<=n; len<<=1) {
		int wlen = invert ? root_1 : root;
		for (int i=len; i<root_pw; i<<=1)
			wlen = int (wlen * 1ll * wlen % mod);
		for (int i=0; i<n; i+=len) {
			int w = 1;
			for (int j=0; j<len/2; ++j) {
				int u = a[i+j],  v = int (a[i+j+len/2] * 1ll * w
% mod);
				a[i+j] = u+v < mod ? u+v : u+v-mod;
				a[i+j+len/2] = u-v >= 0 ? u-v : u-v+mod;
```

```
                        w = int (w * 1ll * wlen % mod);
                }
            }
        }
        if (invert) {
            int nrev = reverse (n, mod);
            for (int i=0; i<n; ++i)
                a[i] = int (a[i] * 1ll * nrev % mod);
        }
}
```

Here, the function $\mathrm{reverse}$is the inverse of $n$an element modulo mod(see. inverse element in the mod). Constants mod, determine the module and a primitive root, and - the inverse of an element modulo .root $\mathrm{root\_pw}\mathrm{root\_1}\mathrm{root}\mathrm{mod}$

As practice shows, the implementation of integer DFT works even slower implementation of complex numbers (due to the huge number of operations modulo), but it has advantages such as low memory usage and the lack of rounding errors.

Some applications

In addition to direct application to multiply polynomials, or long numbers, we describe here are some other applications of the discrete Fourier transform.

All possible sums

Problem: given two arrays $a[]$and $b[]$. You want to find all sorts of species $a[i]+b[j]$, and for each of the number of prints the number of ways to get it.

For example, in $a=(1,2,3)$and $b=(2,4)$obtain the number of 3 may be prepared 1 by the method, 4 - and one, 5 - 2, 6 - 1 7 - 1.

Construct from the arrays $a$and $b$two polynomials $A$and B. As the degree of the polynomial will be performing numbers themselves, ie values $a[i]$($b[i]$), and as the coefficients of them - the number of times it occurs in the array by a(b).

Then, multiplying these two polynomials in $O(n \log n)$, we get a polynomial C, where the powers are all sorts of species $a[i]+b[j]$, and their coefficients are just the required number of

All kinds of scalar products

Given two arrays $a[]$and $b[]$the same length n. You want to display the values of each of the inner product of the vector $a$for the next cyclic shift vector b.

Invert the array $a$and assign it to the end of the $n$zeros, and the array b- simply assign himself. Then multiply them as polynomials. Now consider the coefficients of the product $c[n \dots 2n-1]$(as always, all the indices in the 0-indexed). We have:

$$c[k] = \sum_{i+j=k} a[i]b[j].$$

Since all the elements $a[i]=0,\ i=n \dots 2n-1$, we get:

$$c[k] = \sum_{i=0}^{n-1} a[i]b[k-i].$$

It is easy to see in this sum, it is the scalar product $a$on $k-n-1$th cyclic shift. Thus, these coefficients (since $n-1$th and pumping $2n-2$of th) - is the answer to the problem.

The decision came with the asymptotic behavior $O(n \log n)$.

Two strips

Given two strips defined as two Boolean (ie numeric with values 0 or 1) of the array $a[]$and $b[]$. Want to find all such positions in the first strip that if you apply, starting with this position, the second strip, in any place will not work trueright on both strips. This problem can be reformulated as follows: given a map of the strip, as 0/1 - you can get up into the cell or not, and has some figure as a template (in the form of an array, in which 0 - no cells, 1 - yes), requires find all the positions in the strip, which can be attached figure.

This problem is in fact no different from the previous problem - the problem of the scalar product. Indeed, the dot product of two arrays 0/1 - the number of elements in which both were unity. Our task is to find all cyclic shifts of the second strip so that there was not a single element, which would be in both strips were one. Ie we have to find all cyclic shifts of the second array, in which the scalar product is zero.

Thus, this problem we decided for $O(n \log n)$.