basics

Division

Let a and b be integers with a . We say a divides b . denoted by a|b, if there exists an integer c such that b = ac . When a divides b . we say that a is a divisor (or factor) of b. and b is a multiple of a . If a does not divide b. we write a fi b . If a |(b and 0 < a < b . then a is called a proper divisor of b .

Euclid's Theorems

First theorem: p | ab => p|a or p | b. A direct consequence of this is the fundamental theorem of arithmetic.
Second theorem : There are infinitely many primes. There are many simple proofs for this.

Sieve

. Given a positive integer n > 1, this algorithm will find all prime numbers up to n . [1] Create a list of integers from 2 to n ;
[2] For prime numbers p ; (i = 1,2 . . .') from 2,3.5 up to root(n), delete all the multiples pi < p,m < n from the list ; [3] Print the integers remaining in the list ..

Problems with Simple Sieve:

The Sieve of Eratosthenes looks good, but consider the situation when n is large, the Simple Sieve faces following issues.

- An array of size Θ(n) may not fit in memory
- The simple Sieve is not cache friendly even for slightly bigger n. The algorithm traverses the array without locality of reference

Segmented Sieve

The idea of segmented sieve is to divide the range [0..n-1] in different segments and compute primes in all segments one by one. This algorithm first uses Simple Sieve to find primes smaller than or equal to √(n). Below are steps used in Segmented Sieve.

1. Use Simple Sieve to find all primes upto square root of ‘n’ and store these primes in an array “prime[]”. Store the found primes in an array ‘prime[]’.
2. We need all primes in range [0..n-1]. We divide this range in different segments such that size of every segment is at-most √n
3. Do following for every segment [low..high]
    - Create an array mark[high-low+1]. Here we need only O(x) space where **x** is number of elements in given range.
    - Iterate through all primes found in step 1. For every prime, mark its multiples in given range [low..high].

If a and b are integers, not both zero then the set S={ar+by : :r,yEZ } is precisely the set of all multiples of d= gcd(a,b) .

Euclid

Let a, b, q . r be integers with b > 0 and 0 < r < b such that a = bq + r . Then gcd(a, b) = gcd(b, r) .

Intuition

Suppose a = xg, b = yg for some x, y, g. a%b = a-kb for some k.

So a%b = a-kb = xg - kyg = g(x-ky). So g is still a divisor of both a%b and b. A similar argument proves the converse: if g divides a%b and b, then g divides a and b. So (b, a%b) has the same set of divisors as (a, b).

Furthermore, a%b < b. So eventually we will get to the pair (h, 0) for some h.
Since (h, 0) has the same set of divisors as (a, b), the GCD is the same too. Since the GCD of (h, 0) is obviously h, so is the GCD of (a, b).

http://www.spoj.com/problems/MAIN74/

Let us say n = fibonacci(N) and m = fibonacci(N - 1)

fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)

OR n = m + k where k < m.

Therefore the step

```
n = n % m will make n = k

swap(n, m) will result in

n = fibonacci(N-1)

m = k = fibonacci(N-2)
```

So, it will take N steps before m becomes 0.

This means, in the worst case, this algorithm can take N step if **n** is Nth fibonacci number.

Think of what's the relation between N and n.

The numbers form the fibonacci sequence if you try solving it for some test cases.
Example for n= 2 loops the answer will be 5 (3,2).
for n= 3 loops, result is 8 (5,3).
for n=4 loops, output will be 11 (8,3).
And so on...
Hence it is obvious that for n >=2 output is (n + 3)th fibonacci term.
Now to solve the problem in O(logn) time you should use the optimized matrix multiplication method to generate the fibonacci term.

Linear Diophantine Equations

A Diophantine equation is a polynomial equation, usually in two or more unknowns, such that only the integral solutions are required. An Integral solution is a solution such that all the unknown variables take only integer values.

Given three integers a, b, c representing a linear equation of the form : ax + by = c. Determine if the equation has a solution such that x and y are both integral values.

General solution

(x, y) = (xo + b/d *t, yo — a/d *t)

Integer Factorization

The most commonly used algorithm for the integer factorization is the **Sieve of Eratosthenes**. It is sufficient to scan primes upto sqrt(N) while factorizing N. Also, if we need to factorize all numbers between 1 to N, this task can be done using a single run of this algorithm - For every integer k between 1 to N, we can maintain a single pair - the smallest prime that divides k, and its highest power , say (p,a). The remaining prime factors of k are then same as that of k/(p^a).

Congruences

Let a and b be integers and n a positive integer . We say that "a. is congruent to b modulo n" . denoted by a b (mod n) if n is a divisor of a — b, or equivalently, if n (a. — b) .

Linear Congruences
ax=_b (mod n) => ax — ny = b .

Chinese Remainder Theorem

Typical problems of the form "Find a number which when divided by 2 leaves remainder 1, when divided by 3 leaves remainder 2, when divided by 7 leaves remainder 5" etc can be reformulated into a system of linear congruences and then can be solved using Chinese Remainder theorem. For example, the above problem can be expressed as a system of three linear congruences: "x ≡ 1 (mod 2), x ≡ 2 mod(3), x ≡ 5 mod (7)".

```
  x % num[0]    =  rem[0],
     x % num[1]    =  rem[1],
```

```
......................
x % num[k-1]  =  rem[k-1]
```

A **Naive Approach to find x** is to start with 1 and one by one increment it and check if dividing it with given elements in num[] produces corresponding remainders in rem[]. Once we find such a x, we return it

CRT

```
x =  ( Σ (rem[i]*pp[i]*inv[i]) ) % prod
   Where 0 <= i <= n-1

rem[i] is given array of remainders

prod is product of all given numbers
prod = num[0] * num[1] * ... * num[k-1]

pp[i] is product of all but num[i]
pp[i] = prod / num[i]

inv[i] = Modular Multiplicative Inverse of
         pp[i] with respect to num[i]
```

Euler Phi Function, divisor function, sum of divisors, Mobius function

Euler's **Phi function** (also known as totient function, denoted by φ) is a function on natural numbers that gives the count of positive integers coprime with the corresponding natural number. Thus, φ(8) = 4, φ(9) = 6

The value φ(n) can be obtained by Euler's formula : Let $n = p_1^{a_1} * p_2^{a_2} * .... * p_k^{a}$ be the prime factorization of n. Then

$\varphi(n) = n * (1- 1/p_1)) * (1- 1/p_2)) * ... * (1- 1/p_k))$

CODE

```
int phi[] = new int[n+1];

for(int i=2; i <= n; i++) phi[i] = i; //phi[1] is 0


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

if( phi[i] == i )

for(int j=i; j <= n; j += i )

phi[j] = (phi[j]/i)*(i-1);
```

CODE

PROPERTIES

1. If P is prime then φ(p^k) = (p-1)p^(k-1)

2.φ function is multiplicative, i.e. if (a,b) = 1 then φ(ab) = φ(a)φ(b).

3.Programmatically, if we want to find φ for 1 to n, then we can very well use the sieve algorithm along with the multiplicative property of φ. The central idea is this: if n is a prime, then φ(n) = n-1. Otherwise, if n is a power of a prime, say n= p^k, then φ(n) = (p-1)p^k-1. Otherwise, let n=pk*q for some prime p. Using multiplicative property, we have φ(n) = φ(p^k)φ(q)

a^φ(n) ≡ 1 (mod n) whenever (a,n) = 1. Specifically, for a prime p, if p does not divide a then ap-1 ≡ 1 (mod p).

Let d_1, d_2, ...d_k be all divisors of n (including n). Then $\varphi(d_1) + \varphi(d_2) + ... + \varphi(d_k) = n$

For example: the divisors of 18 are 1,2,3,6,9 and 18. Observe that φ(1) + φ(2) + φ(3) + φ(6) + φ(9) + φ(18) = 1 + 1 + 2 + 2 + 6 + 6 = 18

divisor function

d(n) = (a1+1) * (a2+1) * ... (ak + 1)

Sum of divisors, prod of divisors, perfect numbers

(Wilson's theorem) . If p is a prime, then (p — 1)! = -1 (mod p) .

- when N <= 10, then both O(N!) and O(2^N) are ok (for 2^N probably N <= 20 is ok too)
- when N <= 100, then O(N^3) is ok (I guess that N^4 is also ok, but never tried)
- when N <= 1.000, then N^2 is also ok
- when N <= 1.000.000, then O(N) is fine (I guess that 10.000.000 is fine too, but I never tried in contest)
- finally when N = 1.000.000.000 then O(N) is NOT ok, you have to find something better...