Number Theory

I. Prime Sieve, Prime Testing, Prime Factorization

A prime number is one that is only divisible by one and itself. If we are testing a single number of primality, we do trial division until the square root of the number. To see this, note that if n = ab, where a > 1 and b > 1, at least one of the two is less than or equal to the square root. If both were greater, than the product of ab and would greater than $\sqrt{n}\sqrt{n} = n$, but it would be impossible for n to be greater than n. Thus, it follows if n has a non-trivial divisor, then it must have at least one non-trivial divisor less than or equal to the square root of n. **Note: for the duration of these notes, long long will be used and typdefed to ll.** Here is a function that performs this task:

```
bool isprime(ll n) {
    if (n<2) return false;
    for (ll i=2; i*i<=n; i++)
        if (n%i == 0)
            return false;
    return true;
}
```

If we want to generate a list of all primes from 2 to n, we can use the Sieve of Eratosthenes (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). It works as follows:

1) Write down all the numbers from 2 to n.
2) Go through each number, in order.
3) For each of these, if it's not crossed off, circle it.
4) Then, cross off each multiple of that number. Thus, when we circle 2 at the beginning of the algorithm, then cross off 4, 6, 8, 10, and so forth, until we get to the last even number less than or equal to n.

The numbers not crossed off at the end of these (the circled ones) are all the primes in the range.

As previously mentioned, we can technically stop our outer loop when we get to the square root of the number we are checking because any non-prime has at least one non-trivial divisor less than or equal to its square root.

Here is a function that implements a basic prime sieve for all primes upto n and returns a bool vector which stores true in an index if that value is prime and false otherwise.

```
vector<bool> primesieve(int n) {

    // Set up sieve.
    vector<bool> sieve(n+1);
    sieve[0] = false; sieve[1] = false;
    for (int i=2; i<=n; i++) sieve[i] = true;

    // Run it.
    for (int i=2; i*i<=n; i++)
        for (int j=2*i; j<=n; j+=i)
            sieve[j] = false;

    return sieve;
}
```

If a number is already crossed off, there is no need to cross off its multiples. For example, when we get to 6, all of its multiples were crossed off when we circled 2, so there's no need to cross them off again. Can you think about what to edit in the code above to skip these unnecessary loop iterations? Also, what could you edit to stop looking for factors after you reach the square root of MAX?
After implementing the prime sieve, we are left with a boolean array such that sieve[i] is set to true if and only if i is prime. For some questions, this formation of the data is good enough to solve problems. For other problems, it's necessary to have a list of integers (or an array) with the prime numbers in successive order.

Here is a function that places all of the prime numbers identified by the sieve in an vector, in numerical order:

```
vector<int> primelist(int n) {

    // Run sieve.
    vector<bool> sieve = primesieve(n);

    // Add primes to list and return.
    vector<int> res;
    for (int i=2; i<=n; i++)
        if (sieve[i])
            res.push_back(i);
    return res;
}
```

One of these two storage methods should suffice for most problems that require the generation of all primes up to some bound.

Once we can check for primality, we can also calculate the prime factorization of an integer by repeatedly dividing out prime factors until the number left is prime. Here is some code that returns the prime factorization of an integer n as a vector of pairs:

```
vector< pair<ll,int> > primefact(ll n) {

    // Store each base, exponent pair here.
    vector< pair<ll,int> > res;

    ll i = 2;

    // We can stop here.
    while (i*i <= n) {

        // See how many times i goes into n.
        int exp = 0;
        while (n%i == 0) {
            exp++;
            n /= i;
        }

        // If necessary add the term.
        if (exp > 0) {
            res.push_back(pair<ll,int>(i,exp));
        }

        // Go to next integer.
        i++;
    }

    // In case we missed one.
    if (n>1) res.push_back(pair<ll,int>(n,1));
    return res;
}
```

II. GCD, LCM, Modular Inverse

The greatest common divisor of two positive integers is the largest number that divides evenly into both. The least common multiple of two positive integers is the smallest number such that both of the integers divide evenly into it.

Here is a recursive solution (Euclid's Algorithm) to determine the gcd of two values:

```
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Note that if we have the prime factorizations of two integers, we can find their gcd as follows:

Let X = $\prod_{p_i \in Prime} {p_i}^{x_i}$, and Y = $\prod_{p_i \in Prime} {p_i}^{y_i}$. Then, we have

gcd(X, Y) = $\prod_{p_i \in Prime} {p_i}^{\min(x_i, y_i)}$.

Similarly, we find that lcm(X, Y) = $\prod_{p_i \in Prime} {p_i}^{\max(x_i, y_i)}$.

Since it's always true that a + b = max(a, b) + min(a, b) and the exponent rule shows that $p^a p^b = p^{a+b}$, it can be proved that $XY = \gcd(X, Y) \times lcm(X, Y)$. This allows us to find the least common multiple of two integers via the GCD algorithm:

$$lcm(X, Y) = \frac{XY}{\gcd(X, Y)}$$

Thus, we can write an lcm function as follows:

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

Modular Inverse Problem

The modular inverse problem is as follows, given integer A and N that don't share any common factors, for what value(s) of X is $AX \equiv 1\ (modN)$. A slight modification of the Euclidean Algorithm discovers the value of X as follows (assume ll is typedef'ed for long long)

```
// Wrapper function, returns a^-1 mod n.
ll modinv(ll a, ll n) {
     vector<ll> res = modinvrec(a, n);
     return res[1] >= 0 ? res[1] : res[1] + n;
}

// returns [x,y] such that nx + ay = 1.
vector<ll> modinvrec(ll a, ll n) {
     if (a == 1) return vector<ll>{0,1};
     vector<ll> res = modinvrec(n%a, a);
     return vector<ll>{res[1], res[0]-res[1]*(n/a)};
}
```

There turn out to be problems where you end up needing to do one of two things:

(1) Calculate some expression under mod where you need to divide by an integer. (For example, let's say you need to calculate $\frac{xy}{z}\ mod\ n$. Division isn't allowed, but instead, you can calculate $xy(z^{-1}) mod\ n$. You have to mod after each multiplication so overflow doesn't occur.

(2) Solve an equation of the form $ax \equiv b\ (mod\ n)$, where gcd(a, n) = 1 and x is the variable for which you want a solution, while a and b are know. We're not allowed to divide by a here, but what we CAN do is multiply by $a^{-1}(mod\ n)$. When we do this, the right-hand side simplifies to just x, and the solution is $(a^{-1}b)\ (mod\ n)$.

Also, whenever n is prime, it turns out that a^{-1} mod n is equal to a^{n-2} mod n. Many times, in competitive programming, the modulus value is prime, so calling fast modular exponentiation with those values will also compute the modular inverse.

Number and Sum of Divisors of an Integer

The fundamental theorem of arithmetic states that for any positive integer, n, we can prime factorize it in a unique way. Mathematically, we have:

$$n = \prod_{p_i \in Prime} {p_i}^{n_i},$$

as a unique representation of n. Consider an example of $n = 2^5 3^3 7^{10}$. Any divisor of n must take the form $2^a 3^b 7^c$, where $0 \le a \le 5$, $0 \le b \le 3$, and $0 \le c \le 10$. Since the choice of a, b and c are independent of one another, there are exactly $(5 + 1)(3 + 1)(10 + 1) = 264$ total divisors of n, since we just multiply the number of possible values of a, b and c together. More generally, this means that given the prime factorization of n, the number of divisors it has is:

$$d(n) = \prod_{p_i \in Prime} (n_i + 1)$$

If we wanted to sum the divisors, let's go back to our example. We would want to add all numbers of the form $2^a 3^b 7^c$, where $0 \le a \le 5$, $0 \le b \le 3$, and $0 \le c \le 10$. Consider the following product:

$$(2^0 + 2^1 + 2^2 + \ldots + 2^5)\,(3^0 + 3^1 + 3^2 + 3^3)\,(7^0 + 7^1 + 7^2 + \ldots + 7^{10})$$

Notice that when we foil this out, it'll have 6 x 4 x 11 terms, and that each term will be a unique term of the form $2^a 3^b 7^c$, where $0 \le a \le 5$, $0 \le b \le 3$, and $0 \le c \le 10$. This means that this expression ***is the sum of the divisors of the original number!!!*** Since each of the summations are a geometric sequence, we can create a closed form formula for each sum to derive this formula:

$$\sigma(n) = \prod_{p_i \in Prime} \left(\sum_{j=0}^{n_i} {p_i}^j\right) = \prod_{p_i \in Prime} \left(\frac{{p_i}^{n_i+1} - 1}{p_i - 1}\right)$$

Euler Phi/Totient Function

One useful value in many problems is the Euler Totient function (or Phi function). We define the function as follows:

$\phi(n)$ = the number of values in the set {1, 2, 3, …, n} that are relatively prime with n.

We can calculate φ(n) as follows:

1) Find all unique prime factors, p_1, p_2, …, p_k of n. (Doesn't matter how many times each of them appears in n.)

2) $\phi(n) = n \prod_{i=1}^{k} (1 - \frac{1}{p_i})$

Note that the pi symbol simply indicates multiplying each term instead of adding each enumerated term.

An alternate formula with the prime factorization of n is:

3) $\phi(n) = \prod_{i=1}^{k} ({p_i}^{n_i} - {p_i}^{n_i - 1})$

Both are mathematically equivalent. Either should be equally easy to code. There are several types of programming team questions where one wants to find the number of values from 1 to n that don't share a common factor with n. Here is a function that determines $\phi(n)$:

```
ll phi(ll n) {

    ll res = n;
    int i = 2;

    while (i*i <= n) {
        int exp = 0;
        while (n%i == 0) {
            exp++;
            n /= i;
        }

        if (exp > 0) res = res/i*(i-1);
        i++;
    }

    if (n>1) res = res/n*(n-1);

    // This is phi.
    return res;
}
```

Note that if we had to calculate phi of many values (say 1 to 10^6), we could pre-compute them quickly because once we find one prime factor and factor its copies out, we can use the third formula to build $\phi(n)$ from $\phi(m)$, where *m* is what's left when you divide out all copies of some prime *p* from n. Here is the code for that function:

```
vector<int> philist(int n) {

    // To speed this up.
    vector<bool> isprime = primesieve(n);

    // Initialize phi list.
    vector<int> res(n+1);
    res[0] = 0; res[1] = 1; res[2] = 1;

    // Solve for rest.
    for (int i=3; i<=n; i++) {

        // Take care of these cases.
        if (isprime[i]) {
            res[i] = i-1;
            continue;
        }

        // Find smallest prime divisor.
        int j = 2;
        while (i%j != 0) j++;

        // Divide out all copies of j.
        int newi = i, mypow = 1;
        while (newi%j == 0) {
            newi /= j;
            mypow *= j;
        }

        // Phi formula.
        res[i] = res[newi]*(mypow - mypow/j);
    }

    return res;
}
```

Euler's Theorem

Modular exponentiation is cyclic. For any base, as you raise it to higher and higher exponents, eventually the list of remainders mod some given n repeats in a cyclic pattern. The length of any of these cycles always divides evenly into $\phi(n)$.

Euler's Theorem is as follows:

if gcd(a, n) = 1, then $a^{\phi(n)} \equiv 1(mod\ n)$

This theorem allows us to quickly figure out modular exponentiation results. A nice trick based on this theorem is determining the modular inverse of a value. Note that:

$$a \times a^{\phi(n)-1} \equiv 1(mod\ n)$$

It follows that

$$a^{-1} \equiv a^{\phi(n)-1}(mod\ n)$$

A completely random fact about primes and factorials

The number of times a prime p divides evenly into n! is $\sum_{k=1}^{n}\left\lfloor \frac{n}{p^k} \right\rfloor$

Couple notes: the brackets stand for the floor function. The sum doesn't really need to go to n. You'll notice that p^k fairly quickly exceeds the value of n. When it does, all subsequent terms in the sum are 0. So you just have to sum all the terms until one is zero. A brief explanation as to why this works is imagine dividing out p from the written out expression n! You would cancel out one out of every p values. But this would leave some extra terms, since when you got to p^2 or any mutliple thereof, you would have only cancelled out one of the two p's in that term. That's where the rest of the sum comes in. k=2 knocks out the extra factors in each term that is divisible by p^2,
k=3 knocks out the extra factors in each term that is divisible by p^3, etc.

Here is code that does this computation:

```
ll numTimesDivide(ll n, ll p) {
    ll res = 0;
    while (n >= p) {
        res += n/p;
        n /= p;
    }
    return res;
}
```