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.

Kattis Problem: Happy Happy Prime Prime

Here is the link to the problem:

https://open.kattis.com/problems/happyprime

In this problem, we are given an input number and need to determine if it is a happy prime or not. A number can only be a happy prime if it is a prime number, so as our first step, we can generate all the primes upto 10,000. The reason we want to do this is because this problem has many test cases, and it's faster to generate all of these primes than to run the prime test 10,000 times.

If a number isn't prime, it's not a happy prime. Otherwise, we have to run the "happiness test" that is described. In this test, either we follow the steps and arrive at 1, which would infinitely repeat, OR, we cycle through a set of numbers infinitely but never hit 1. Thus, to do the happiness test, we must detect a repeated number. We can do this with either a boolean vector that's big enough (probably 500 is big enough because each digit square is less than 100 and there aren't going to be more than 5 digit), or a set.

In the implementation below, we use a set. Thus, there are two tasks:

1. Running the Sieve of Eratosthenes
2. Determining if a prime number is happy or not using a set to detect a cycle. (A cycle is when we start repeating the same set of items in sequence.)

```
// Arup Guha
// 5/23/2024
// Solution to Kattis Problem: Happy Happy Prime Prime
// https://open.kattis.com/problems/happyprime

using namespace std;
#include <bits/stdc++.h>
vector<bool> primesieve(int n);
bool ishappy(int n);
int sumdsq(int n);

int main() {

    int nC;
    cin >> nC;

    // Get prime list.
    vector<bool> sieve = primesieve(10000);

    // Process cases.
    for (int loop=0; loop<nC; loop++) {

        int cnum, n;
        cin >> cnum >> n;

        // Echo...
        cout << cnum << " " << n << " ";
        // Output accordingly.
        if (sieve[n] && ishappy(n))
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
    return 0;
}

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;
}

bool ishappy(int n) {
    set<int> used;
    used.insert(n);

    // Keep going until we get to 1.
    while (n != 1) {
        n = sumdsq(n);

        // Detected a cycle.
        if (used.find(n) != used.end())
            return false;

        // Add it to the set.
        used.insert(n);
    }
    return true;
}

// Returns the sum of the squares of the digits of n.
int sumdsq(int n) {
    int res = 0;
    while (n > 0) {
        // Get units digit, square and add.
        int digit = n%10;
        res += (digit*digit);

        // Chop off last digit.
        n /= 10;
    }

    return res;
}
```

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);
}
```

With 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) = $\prod_{p_i \in Prime} p_i^{\min(x_i, y_i)}$. 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;
}
```

Kattis Problem: GCD

Here is a quick solution to the exact gcd problem on Katts:
https://open.kattis.com/problems/gcd

```
using namespace std;
#include <bits/stdc++.h>
int gcd(int a, int b);

int main() {
    int a, b;
    cin >> a >> b;
    cout << gcd(a, b) << endl;
    return 0;
}

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%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:

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

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;
}
```

Kattis Problem: Factorial Power
Here is a problem that will utilize our prime factorization code and the function we just derived above:

https://open.kattis.com/problems/factorialpower

First, we must prime factorize the unput value n, because in order to answer how many times n divides evenly into m! we have to break our work into each of the prime number components.

Secondly, we must realize that if 2 divides into n! 13 times, then 2^3 will only divide into n! 13/3 = 4 times.

Finally, we simply want the minimum of all prime factors.

Let’s work out a quick example by hand:

How many times does 12 divide evenly into 30!

$12 = 2^2 \times 3$

Running our code, we find that 2 divides into 30! 26 times, so that means that 2^2 divides into 30! 13 times.

Running our code, we find that 3 divides into 30! 14 times.

Thus, our answer is the minimum of 13 and 14, which is 13. (In essence, 2^2 divides into 30! fewer times than 3, so that's our limiting factor.)

Here's the solution, which is completely uses two of the functions from the notes without any changes:

```
// Arup Guha
// 5/23/2024
// Solution to Kattis Problem: Factorial Power
// https://open.kattis.com/problems/factorialpower

using namespace std;
#include <bits/stdc++.h>
typedef long long ll;

ll numTimesDivide(ll n, ll p);
vector< pair<ll,int> > primefact(ll n);

int main() {

    ll n, m;
    cin >> n >> m;

    // Get prime factorization.
    vector< pair<ll,int> > terms = primefact(n);

    // Can't be bigger than this!
    ll res = m;

    for (pair<ll,int> mypair: terms) {

        // Determine the number of times this prime divides into m!
        ll limit = numTimesDivide(m, mypair.first);

        // Adjust this limit for the number of times this prime is in the
        // prime factorization of n.
        limit /= mypair.second;

        // Update...
        res = min(res, limit);
    }

    // Ta da!
    cout << res << endl;
    return 0;
}
```

```
// Returns the number of times prime p divides evenly into n!
ll numTimesDivide(ll n, ll p) {
    ll res = 0;
    while (n >= p) {
        res += n/p;
        n /= p;
    }
    return res;
}

// Returns the prime factorization of n.
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;
}
```

Kattis Problem: Non-Prime Factors

Here is a Kattis problem that uses multiple ideas from these notes and requires a non-trivial application of those ideas:

https://open.kattis.com/problems/nonprimefactors

Note that we just want to count the number of divisors of an integer, for which we have a formula in the notes that was previously derived. Unfortunately, we have to solve 3,000,000 of these cases, so if we use this formula individually (run time on a single integer would be over 1000 simple steps), we'd be doing over 3 billion steps, which would take too long. So, we have to solve ALL of the cases upto 2,000,000 (the upper bound of the problem) first and then answer each query quickly.

Notice that since we have a formula for number of divisors, we need to see if we can "build up" the answers for all the numbers in range more quickly. Notice that we are NOT supposed to count the prime divisors. This means that the second subproblem we need to solve is:

For all integers in between 1 and n, how many unique primes divide evenly into each of them?

If we think about how the prime sieve works, it marks each multiple of every number. But, we can adjust it so that it only runs the inner loop for primes! Thus, the idea is as follows in adjusting the prime sieve:

1. Run the same outer loop. This value will represent the prime number we are processing.

2a. If we find that this number has a previous prime that divides into it, we "skip it".

2b. If there was no previous prime that divides into it, then add 1 to its prime count and then add 1 to the prime count of each of its multiples!

When we're done with this, instead of having an array of trues and falses indicating which numbers are prime, for each number, we'll store how many unique primes divide into it!

The second part of the question is how do we quickly build up the number of total divisors. Recall that the formula for this requires knowledge all the exponents in the prime factorization of the number. But, as previously described, we don't have time to determine each of the prime divisors of a number separately.

A nice property of the number of divisors formula is that it can build off of the number of divisors of smaller numbers. Take, for example, the number $120 = 2^3 \times 3 \times 5$. The number of divisors is a multiplicative function, so we have:

$\text{numdiv}(120) = \text{numdiv}(2^3) \times \text{numdiv}(3 \times 5)$

If we are building these answers up, then by the time we need to calculate numdiv(120), we would already have numdiv(15) stored in our array.

Thus, in order to make our previous information useful, we would simply need to know the smallest prime that divides evenly into 120.

BUT, if we think about it, when we were running our original code to count the number of unique prime divisors of an integer, we could have just "marked" when we found the first prime that divided into an integer.

Thus, we make an adjustment to our original pre-computation, not only computing the number of unique prime divisors of an integer, but also the smallest prime divisor of an integer.

Now, if we need to calculate the number of divisors of an integer, we can very quickly divide out each copy of the smallest prime divisor (there won't be too many copies…), and then use that exponent as well as the previous number of divisors information to quickly calculate the number of divisors of the current number.

Thus, we have two "sweeps" through the data, so to speak. In our first pass, we calculate the minimum prime and number of unique prime divisors for each integer in between 1 and 2,000,000. In our second pass, we use the information from our first pass to quickly calculate the number of divisors for each integer, from small to big.

Once we have both of these values, then the answer for each query, k is:

numdiv[k] – numprime[k]

Here's the code:

```
// Arup Guha
// 5/23/2024
// Solution to Kattis Problem: Non-Prime Factors
// https://open.kattis.com/problems/nonprimefactors

using namespace std;
#include <bits/stdc++.h>

#define MAX 2000001

int main() {

    // First we'll generate a list of the minimum prime that divides
    // into each integer and the number of distinct primes that do.
    vector<int> minprime(MAX);
    vector<int> numprime(MAX);
    for (int i=0; i<MAX; i++) {
        minprime[i] = -1;
        numprime[i] = 0;
    }

    // Go through each number.
    for (int i=2; i<MAX; i++) {

        // We were marked before, so we aren't prime,
        // don't mark anyone else.
        if (minprime[i] != -1) continue;

        // We are prime store this.
        minprime[i] = i;
        numprime[i] = 1;
        // Here we mark other instances of minprime as necessary and
        // add 1 to all multiples.
        for (int j=2*i; j<MAX; j+=i) {
            if (minprime[j] == -1) minprime[j] = i;
            numprime[j]++;
        }
    }

    // We'll store total number of divisors here.
    vector<int> numd(MAX);
    numd[1] = 1;

    // Go through each number.
    for (int i=2; i<MAX; i++) {

        int exp = 0, n=i;

        // Figure out how many times the smallest prime divides into i.
        while (n%minprime[i] == 0) {
            exp++;
            n /= minprime[i];
        }

        // So here we build our formula for number of divisors off of
the number of divisors
        // of n with the smallest prime divided out.
        numd[i] = numd[n]*(exp+1);

    }

    // This is really annoying C++ I/O is too slow,
    // so I used old school C to get this to pass.
    int nC;
    scanf("%d", &nC);

    // Process cases.
    for (int loop=0; loop<nC; loop++) {
        int val;
        scanf("%d", &val);
        printf("%d\n", numd[val] - numprime[val]);
    }

    return 0;
}
```