Primes and primality testing

As we saw last time, prime numbers are important in number theory. This immediately creates two algorithmic problems – how do we tell whether a number is prime, and how do we generate all the prime numbers in a given range? First, a definition.

Definition 1:
An integer p > 1 is prime if and only if it has exactly two positive divisors - 1 and p.

This means that the first ten primes are 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29.

Lemma 1:
There is an infinite number of primes.

Proof:
Suppose not. Then let X be the largest prime. Now let Y = X! + 1. When we try to divide Y by any integer between 2 and X inclusive, the remainder is always 1 because all of those integers divide X!. Hence, Y is not divisible by any prime – it must be a prime itself. But Y > X, yet X is the largest prime. Contradiction.

Testing whether a given integer n is prime is an old problem in algorithmic number theory. Basically, there are 3 solutions: (1) the one that's easy to code, (2) the one that is fast in theory, and (3) the one that is fast in practice. Here is algorithm number 1.

Algorithm 1: Simple primality testing

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

We simply check whether n is divisible by any integer larger than 1 and smaller than n. If yes, then n is not prime. If we consider % to be a O(1) operation, then this algorithm is $O(\sqrt{n})$ because if n is not prime (composite), then it must have at least 2 non-trivial factors, one of which must be no larger than $\sqrt{n}$.

Algorithm number 2 was discovered in 2002 by Agrawal, Kayal and Saxena and is $O(\log^{7.5} n)$ in the worst case. However, if a certain widely believed conjecture is true, it is $O(\log^{6} n)$. It seems to run in $O(\log^{3} n)$ on average in practice. This algorithm is important because it was the first one that is polynomial in the number of digits of n.

The third algorithm is called the Rabin-Miller strong pseudo-prime test. It is $O(\log n)$, which is very fast, but it has a small probability of giving a wrong answer. This probability decreases the more times you run the algorithm. This algorithm is most often used in practice, and we may come back and discuss it in detail later.

Sieve

What if instead of testing only one number, we are interested to know whether a lot of different numbers are prime? With some precomputation, we can build a boolean array prime[] that will contain a `true` in every cell that is indexed by a prime, and `false` otherwise. This means we will get `prime[7]=true, prime[15]=false`, etc. This algorithm was discovered in ancient Greece by Erastosthenes and works as follows. First, we take a range that we are interested in: [1, N]. N is the largest number we need to test. We throw away 1 because it is not a prime by definition. Now we repeat a simple procedure – take the smallest number we haven't yet thrown away (it's prime) and throw away all multiples of that number. Repeat. In the end, we will be left with the primes in the range [2, N].

Algorithm 2: Slow sieve of Erastosthenes

```
bool prime[10000000];
void sieve( int N ) {
    for( int i = 0; i <= N; i++ ) prime[i] = true;
    prime[0] = prime[1] = false;
    for( int i = 2; i <= N; i++ ) if( prime[i] ) {
        for( int j = i + i; j <= N; j += i )
            prime[j] = false;
    }
}
```

After running sieve(N), prime[i] will be true if and only if i is prime, for any i in the range [0, N]. During step i of the outer loop, we set prime[j] to false for all j that are multiples of i – that is for j = 2i, 3i, 4i, etc. The inner loop is executed $N/i-1$ times for each iteration of the outer loop. With a little help from calculus, we can show that the total running time is $O(N \log N)$.

However, we can speed this up quite a lot. Note that during iteration i, the j loop starts at 2i and goes to 3i, 4i, etc. But 2i has already been taken care of during the first iteration because it is divisible by 2. 3i is divisible by 3 and has also been set to false already. In fact, the first index in prime[] that we need to care about is i*i. This means that we don't even need to execute the second loop if i*i is larger than N. A faster version looks like this.

Algorithm 3: A faster sieve of Erastosthenes

```
bool prime[10000000];
void sieve( int N ) {
    for( int i = 0; i <= N; i++ ) prime[i] = true;
    prime[0] = prime[1] = false;
    for( int i = 2; i * i <= N; i++ ) if( prime[i] ) {
        for( int j = i * i; j <= N; j += i )
            prime[j] = false;
    }
}
```

Note the two changes in bold. Now the outer loop only makes $\sqrt{N}$ iterations, and the inner loop makes $(N-i^2)/i$ iterations. This still amounts to $O(N \log N)$, but this simple fix usually speeds it up by at least 50%.

Factoring

Another old problem in mathematics is factoring. Given an integer N, find a smaller integer, a, that divides N. Clearly, if N is prime, the we can not find any factors larger than 1. Ideally, we would like to factor N completely into primes, but finding just one factor is enough to do that. Once we have one factor, we simply divide the N by that factor and continue to factor the quotient in a similar manner.

Once again, there are several algorithms. Here is a simple $O(\sqrt{N})$ algorithm. It returns 1 if N is prime, and a factor larger than 1 and smaller than N otherwise. The code is almost exactly the same as for the simple primality testing algorithm.

Algorithm 4: Simple factoring

```
int findFactor( int N ) {
    for( int i = 2; i * i <= N; i++ )
        if( N % i == 0 ) return i;
    return 1;
}
```

The algorithms used in practice are Quadratic Sieve and Number Field Sieve, the best of which has a $O(n^{1/3})$ running time. There is no known algorithm that runs in time polynomial in the number of digits of N.

Since factoring seems to be closely related to primality testing, and since the two fancy algorithms both have the word "sieve" in their name, perhaps we could adapt our sieve of Erastosthenes to factor numbers. Indeed, we can. The only change we need is to replace the prime[] array with an array that will store, for each number x, its smallest non-trivial factor.

Algorithm 5: Factoring sieve

```
int f[10000000];
void fsieve( int N ) {
    for( int i = 1; i <= N; i++ ) f[i] = i;
    for( int i = 2; i <= N; i++ ) if( f[i] == i ) {
        for( int j = i * i; j <= N; j += i )
            if( f[j] == j ) f[j] = i;
}
```

f[x] will be the smallest integer larger than 1 that divides x. At first, we assume that every number is prime. Then, like in the usual sieve(), we iterate over the primes, i, we have so far and mark all their multiples as non-prime. Only this time, we mark them by setting i to be their smallest divisor.

After running fsieve(N), we can take any integer $x \leq N$ and trace it through the f[] array, recovering all of its divisors in non-decreasing order. First, we get the smallest divisor, f[x]. Then we divide x by it and look in f[] for the next factor. We continue until x becomes 1, and we have found all the factors.

Euler's phi-Function

Recall that the Euler's phi-function $\phi(n)$ counts the number of positive remainders in [1,n-1] that are relatively prime to n. Computing this function is vital in many Number Theory applications because of Euler's Theorem from last time. We now discuss how to compute this function in several steps. First we establish an easy case

Lemma 1. Suppose p is a prime number. Then $\phi(p)=p-1$.

Proof. When p is prime, the numbers 1,2,...,p-1 are all relatively prime to p , so there are exactly $p-1$ positive numbers smaller than p that are relatively prime to p . Q.E.D.

The next step is to compute $\phi(n)$ when n is a prime power. Suppose $n=p^r$ where p is prime and $r>1$. How many numbers in {1, 2, ..., p^r-1} are relatively prime to p^r? Well, if a number a is **not** relatively prime to p^r , which means $gcd(a, p^r)>1$, then the two numbers share some common factor. What factors does p^r have?

If we factor out p^r completely, we see that its factors are { 1, p, p^2, p^3, ..., p^r }. So if $gcd(a, p^r)>1$, then a is divisible by p^k for some $k\geqslant 1$. But if it is divisible by p^k , then it is also divisible by p itself. So any number in {1, 2, ..., p^r-1} that is divisible by p is **not** relatively prime to p^r , and vice versa. How many numbers are divisible by p ? There are {p, 2p, 3p, ... }. In fact, any **positive** multiple of p is divisible by p . So we have

Lemma 2. Suppose p s a prime number and $r>1$. Then $\phi(p^r)=p^r-p^{r-1}$.

Proof. First, there are exactly p^r-1 positive remainders less than p^r . Out of these, the numbers

$$\{p, 2p, 3p, \ldots, (p^{r-1}-1)p\}$$

are all the numbers divisible by p , and thus are **not** relatively prime to p^r . So there are exactly $(p^{r-1}-1)$ numbers that are **not** relatively prime to p^r , and we get

$$\phi(p^r)=(p^r-1)-(p^{r-1}-1)$$

which gives

$$\phi(p^r)=p^r-p^{r-1} .$$

Q.E.D.

We learn how to factor any number n into its prime factors, and the following Theorem will tie all the pieces together. The proof of this theorem is non-trivial, and it will be done in details in any Number Theory course. We only sketch the basic concepts here, and provide references if you wish to explore further.

Theorem 3. For any $n>1$, if we factor $n=p_1^{r_1}\times p_2^{r_2}\times\ldots\times p_m^{r_m}$ where each of the p_i is prime and each of the $r_i>0$. Then,

$$\phi(n)=\phi(p_1^{r_1})\phi(p_2^{r_2})\ldots\phi(p_m^{r_m}) .$$

Proof. Sketch. Let's look at a simpler examle. Suppose $n = p \times q$. Then, any number relatively prime to n is **not** divisible by p and **not** divisible by q . Now take any x relatively prime to n , and we would get the following

$$x \equiv a(mod\ p), x \equiv b(mod\ q)$$

where a, b are not divisible by p. Can we solve these two equations simultaneously? It is proved in Number Theory that this is always possible, and for any given pair (a, b) there is a **unique** solution (mod pq). This means, that each pair (a, b) can be mapped uniquely to one remainder in the range [0, pq-1], and this number is relatively prime to $n = pq$. There are $\phi(p)$ different values for a , and there are $\phi(q)$ values for b , so there are exactly $\phi(p)\phi(q)$ values in [0, pq-1] that are relatively prime to n .

Note: The theorem that these equations can always be solved is called the **Chinese Remainder Theorem**, and is important but difficult to prove. Q.E.D.

Now we have enough tools to compute $\phi(n)$ for any value of n . We start by factoring $n = p_1^{r_1} \times p_2^{r_2} \times ... \times p_m^{r_m}$ with methods described in the previous sections, and use Theorem 3 from above. An easier method is the following

Lemma 4. For any $n > 1$, if we factor $n = p_1^{r_1} \times p_2^{r_2} \times ... \times p_m^{r_m}$ where each of the p_i is prime and each of the $r_i > 0$. Then

$$\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2})...(1 - \frac{1}{p_m}) \quad .$$

Proof. Take the equation from Theorem 3. We can factor each term using Lemma 2, giving

$$\phi(p_i^{r_i}) = p_i^{r_i} - p_i^{r_i - 1} = p_i^{r_i}(1 - \frac{1}{p_i}) \quad ,$$

and when we multiply the terms together, we obtain

$$\phi(n) = p_1^{r_1}(1 - \frac{1}{p_1})\, p_2^{r_2}(1 - \frac{1}{p_2})... p_m^{r_m}(1 - \frac{1}{p_m}) \quad ,$$

$$\phi(n) = p_1^{r_1} p_2^{r_2} ... p_m^{r_m}(1 - \frac{1}{p_1})(1 - \frac{1}{p_2})...(1 - \frac{1}{p_m}) \quad ,$$

$$\phi(n) = n(1 - \frac{1}{p_1})...(1 - \frac{1}{p_m}) \quad .$$

Q.E.D.

This method has the advantage that we do not need to know what powers each of the factors are, just what primes divide n . The powers are magically taken care of by the properties of the phi-function.

References

Niven, Ivan, Herbert S. Zuckerman, and Hugh L. Montgomery. An Introduction to the Theory of Numbers. John Wiley & Sons, 1991.