Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 11 - Number Theory

Number theory is the study of the integers. The most basic concept in number theory is *divisibility*. We say that b *divides* a (written $b|a$) if $a = bk$ for some integer k. We can also say that a is a *multiple* of b, or that b is a *divisor* of a (if $b \geq 0$). Every positive integer a is divisible by the trivial divisors 1 and a, and the nontrivial divisors of a are called the *factors* of a.

Greatest Common Divisor The greatest common divisor, or GCD, of two integers a and b, is the largest of the common divisors of a and b. For example, the factors of 24 are 2, 3, 4, 6, 8, and 12, while the factors of 30 are 2, 3, 5, 6, 10, and 15, so the greatest common divisor of them (or gcd(24,30)) is 6. This has several uses. More prosaically, we can use this to simplify fractions by removing the common factors: e.g., reducing $\frac{24}{30}$ to $\frac{4}{5}$. Euclid's algorithm for calculating the GCD is still the most widely used and simple to program. It exploits the property that:

If $a = bt + r$ for integers t and r, then $GCD(a, b) = GCD(b, r)$.

Why? Clearly $a = bt + r$ for some t and r - r is the remainder and t the multiple of b. Then $GCD(a, b) = GCD(bt + r, b)$. But any common divisor of b and $bt + r$ must reside entirely in r, as bt must necessarily be divisible by any divisor of b. So $GCD(a, b) = GCD(r, b)$. So we can write a recursive algorithm to find the GCD of any two positive integers:

```
In [1]: def gcd(a, b):
            if b == 0:
                return a
            return gcd(b, a % b)
```

Least Common Multiple The least common multiple (LCM) is a closely related problem, the *smallest* integer which is divisible by both of a pair of integers. We can calculate it easily using the GCD:

$LCM(a, b) = ab/GCD(a, b)$.

To calculate the GCD and LCM of more than two numbers, we can just nest the calls, e.g., $GCD(a, b, c) = GCD(a, GCD(b, c))$.

Primes A prime number is an integer $p > 1$ whose only divisors are 1 and p. Primes have a number of useful properties and are essential in number theory. Any number which is not prime is called *composite*.

Testing primality There are an infinite number of primes, but they are not distributed according to any pattern. There are roughly $x/ln(x)$ primes less than or equal to x, meaning that roughly 1 out of every $ln(x)$ numbers is prime. The most straightforward way to test whether a number is prime is trial division by candidate divisors. If N is prime, then none of the numbers in $[2, N - 1]$ will divide it, so we loop through all of those numbers and test whether any of them are factors of N:

```
In [4]: def is_prime(N):
            for i in xrange(2, N):
                if (N % i) == 0:
                    return False
            return True
```

```
In [9]: [p for p in range(2, 50) if is_prime(p)]

Out[9]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

This is accurate, but we can make a couple of improvements. First, note that any divisor d of N has a paired divisor d/N. At least one of d and d/N must be less than or equal to $\sqrt{N}$. That means we only need to test candidate divisors up to a maximum of $\sqrt{N}$ rather than $N-1$. Secondly, we know that 2 is the only even prime, so we can simply skip all other even candidate divisors:

```
In [18]: def is_prime_faster(N):
             if (N % 2) == 0:
                 return False
             for i in xrange(3, int(sqrt(N)), 2):
                 if (N % i) == 0:
                     return False
             return True

In [19]: %timeit [p for p in range(2, 10000) if is_prime(p)]
         %timeit [p for p in range(2, 10000) if is_prime_faster(p)]

1 loops, best of 3: 827 ms per loop
100 loops, best of 3: 18.9 ms per loop
```

Finally, if we already know the primes less than $\sqrt{N}$, we can restrict the candidate divisors to the primes rather than all odd numbers, giving an even larger speed up.

These methods work OK for relatively small numbers, but for large numbers like those primes used for cryptography, faster methods are required. For these, we exploit *probabilistic* primality testing, which uses a number of trial random numbers to test using methods I won't go into detail about to estimate whether a number is prime. It does have a chance of failure, but that chance can be made arbitrarily low. If you'd like to know more detail, look up the Miller-Rabin primality test algorithm.

Generating Primes If we want to generate a list of primes less than some N, there's a better way than simply running one of the above `is_prime` tests on each (odd) number in the range. The algorithm we'll talk about is called the Sieve of Erisothanes.

Sieve of Erisothanes The Sieve works by starting with the first prime, 2, and "crossing out" all multiples of 2 between 2^2 and N, marking them as composite. Then it takes the next uncrossed number, 3, and repeats, marking multiples of 3 between 3^2 and N as composite, leaving the next available prime as 5. We keep doing this, repeating until we've generated all the primes we need. A well-implemented Sieve can generate all the primes less than ~10 million in only a few seconds (which, of course, you can generate before submitting your code and simply load in the list of primes you need from a file or from code). For primes larger than that, you'll want to use some kind of optimized probability primality test.

```
In [29]: #Dynamic prime generator using the Sieve
         def primes(max=None):
             composites = {}
             #Yield 2 first, then only loop through odd numbers
             yield 2
             q = 3
             while max is None or q < max:
                 if q not in composites:
                     yield q
                     composites[q * q] = [q]
                 else:
                     for p in composites[q]:
```

```
                    try:
                        composites[p+q].append(p)
                    except KeyError:
                        composites[p+q] = [p]
                del composites[q]
            q += 2
```

```
In [32]: %timeit [p for p in primes(10000000)]

1 loops, best of 3: 7.82 s per loop
```

Prime Factorization The Fundamental Theorem of Arithmetic states that every integer has a unique representation as a multiplication of its prime factors. That is, the prime numbers are the multiplicative building blocks of integers. For example, 1200 can be factored into $2^4 \times 3 \times 5^2$.

A naive algorithm for finding the prime factorization of an integer takes a list of primes (e.g., from a sieve) and simply checks each of them to see which divides the integer. We can do better by dividing the intial integer by each prime factor we find:

```
In [40]: def factor(N):
             f = {}
             for p in primes(sqrt(N)):
                 if p > N:
                     break
                 while (N % p) == 0:
                     try:
                         f[p] += 1
                     except KeyError:
                         f[p] = 1
                     N = N // p
             if N > 1:
                 f[N] = 1
             return f
```

```
In [41]: print factor(1200)
         print factor(136117223861)
         print factor(142391208960)

{2: 4, 3: 1, 5: 2}
{104729: 1, 1299709: 1}
{2: 10, 3: 4, 5: 1, 7: 4, 11: 1, 13: 1}
```

The prime representation of numbers is extremely useful for dealing with very large numbers without integer overflow problems. For example, suppose you were asked how many trailing zeros there are in the decimal representation of 100!. This is a *very* large number, much too large to fit into a 32-bit integer. Python and Java can do this with their big integer handling, but there's an easier and faster way using the prime representation of the number. A trailing zero at the end of an integer indicates a factor of 10 - the number is divisible by 10. We wish to know how many times we can divide the number by 10, which is equivalent to asking how many prime factors of 2 and 5 are present in the number and reporting the smaller. The factorial function will always add more factors of 2 than 5 to the final integer, so we need only count the number of 5s in the prime factorization.

```
In [46]: def count_trailing_zeros(n):
             #Count the number of trailing zeros in the factorial of n
             count = 0
             while n >= 5:
```

```
            count += n // 5
            n = n // 5
        return count
```

```
In [47]: print count_trailing_zeros(100)
```

```
24
```

And let's check, just to confirm that our answer is right:

```
In [44]: def fact(n):
            if n == 1:
                return 1
            return n * fact(n-1)
        print fact(100)
```

```
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862530
```

```
In [1]: 365 % 7
```

```
Out[1]: 1
```

Modular Arithmetic Sometimes we don't *want* to work with the exact values of integers - instead we're interested in working with integers *modulus* some other number. That is, we'll work with the remainder (or residue) of the integer remaining after dividing by the modulus. This could be useful if we want to know, say, what day of the week it will be N days in the future (the N mod 7th day after today), or what the last digit of a large integer is (N mod 10). It's also useful for working with large numbers, where we can more easily work with intermediate values modulo some reasonable number. So let's talk about how modular arithmetic works.

In most languages, there will be a primitive operator (usually %) that gives the remainder of an integer modulus some other number. Let's look at some identities on how to operate on numbers $; mod ; n$. First, sums and differences modulo n are identical if we first take the modulus of the initial numbers:

$(x + y) \bmod n = ((x \bmod n) + (y \bmod n)) \bmod n$

$(x - y) \bmod n = ((x \bmod n) - (y \bmod n)) \bmod n$

This suggests we can extend this to multiplication:

$xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$

and exponentiation:

$x^y \bmod n = (x \bmod n)^y \bmod n$

Division is tricky - you *can't* just do this simple method for modular division. In fact, it's complicated enough (and rare enough) that I'm not going to cover it in class, but the short answer is that you need to find the inverse of some d - a number that when multiplied by d is equal to 1 (mod n), and then multiply by that number. But sometimes inverses don't exist, and in general it just gets complicated. Hie thee to a number theory book (e.g. CLRS) if you need more.

Let's look at an example - what is the last digit of 2^{100}? What we really want to know is 2^{100} mod 10. By repeated squaring and taking the remainder mod 10 at each step we can end up with the final result while avoiding ever having to work directly with large numbers:

$2^3 mod 10 = 8$

$2^6 mod 10 = 8 \times 8 mod 10 = 4$

$2^{12} mod 10 = 4 \times 4 mod 10 = 6$

$2^{24} mod 10 = 6 \times 6 mod 10 = 6$

$2^{48} mod 10 = 6 \times 6 mod 10 = 6$

$2^{96} mod 10 = 6 \times 6 mod 10 = 6$

$2^{100} mod 10 = 2^{96} \times 2^3 \times 2 mod 10 = 6 \times 8 \times 2 = 96 mod 10 = 6$

And we can check to make sure that our answer is right:

```
In [48]: print pow(2, 100)
```

```
1267650600228229401496703205376
```

Chinese Remainder Theorem The Chinese Remainder Theorem deals with finding some x where we know the remainders of x divided by different integers. The theorem states that if the moduli are relatively prime (their GCD is 1) then we are guaranteed a solution (otherwise the different remainders could possibly be inconsistent). Let's look at an example. Suppose we know that

$x \ mod \ 3 = 2$ and

$x \ mod \ 5 = 3$.

We could find the answer(s) by simply listing values of x which hold for the first:

$2, 5, 8, 11, 14, 17, 20, 23, 26, 29 \ldots$

and second equations:

$3, 8, 13, 18, 23, 28, 33 \ldots$

and we can see that $x = 8$ and $x = 23$ are both valid solutions. But this method could take quite a while if we have large remainders, moduli, or a large number of moduli.

Lab Section No official homework this week, but take a look at

Timus Online Judge #1049 - Brave Balloonists

Send me a complete and correct solution and I'll give you homework extra credit!