Number Theory

Number Theory is the study of integers and their resulting *structures*.

Why study it?

1. History: the first true algorithms were number-theoretic.
2. Analysis: We'll learn about new kinds of running times and analyses.
3. Cryptography! Modern cryptosystems rely heavily on this stuff.
4. Computers are always dealing with integers anyway!

How big is an integer?

The **measure of difficulty** for array-based problems was always the size of the array.

What should it be for an algorithm that takes an integer n?

Factorization

Classic number theory question: What is the **prime factorization** of an integer n?

Recall:

- A prime number is divisible only by 1 and itself.
- Every integer > 1 is either prime or composite.
- Every integer has a unique prime factorization.

It suffices to compute a *single* prime factor of n.

leastPrimeFactor

Input: Positive integer `n`
Output: The smallest prime `p` that divides `n`

```
def leastPrimeFactor(n):
    i = 2
    while i * i <= n:
        if n % i == 0:
            return i
        i = i + 1
    return n
```

Running time:
Is this fast??

Polynomial Time

The actual running time, in terms of the size $s \in \Theta(\log n)$ of n, is $\Theta(2^{s/2})$.

Definition

An algorithm runs in **polynomial time** if its worst-case cost is $O(n^c)$ for some constant c.

Why do we care? The following is sort of an algorithmic "Moore's Law":

Cobham-Edmonds Thesis

An algorithm for a computational problem can be feasibly solved on a computer only if it is polynomial time.

So our integer factorization algorithm is actually really slow!

Modular Arithmetic

Division with Remainder

For any integers a and m with $m > 0$, there exist integers q and r with $0 \le r < m$ such that

$$a = qm + r.$$

We write $a \bmod m = r$.
Modular arithmetic means doing all computations "mod m".

Addition mod 15

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7
9	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8
10	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9
11	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10
12	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11
13	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12
14	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Modular Addition

This theorem is the key for efficient computation:

Theorem

For any integers a, b, m with $m > 0$,
$(a + b) \bmod m = (a \bmod m) + (b \bmod m) \bmod m$

Subtraction can be defined in terms of addition:

- $a - b$ is just $a + (-b)$
- $-b$ is the number that adds to b to give 0 mod m
- For $0 < b < m$, $-b \bmod m = m - b$

Multiplication mod 15

×	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13
3	0	3	6	9	12	0	3	6	9	12	0	3	6	9	12
4	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11
5	0	5	10	0	5	10	0	5	10	0	5	10	0	5	10
6	0	6	12	3	9	0	6	12	3	9	0	6	12	3	9
7	0	7	14	6	13	5	12	4	11	3	10	2	9	1	8
8	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7
9	0	9	3	12	6	0	9	3	12	6	0	9	3	12	6
10	0	10	5	0	10	5	0	10	5	0	10	5	0	10	5
11	0	11	7	3	14	10	6	2	13	9	5	1	12	8	4
12	0	12	9	6	3	0	12	9	6	3	0	12	9	6	3
13	0	13	11	9	7	5	3	1	14	12	10	8	6	4	2
14	0	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Modular Multiplication

There's a similar (and similarly useful!) theorem to addition:

Theorem

For any integers a, b, m with $m > 0$,
$(ab) \bmod m = (a \bmod m)(b \bmod m) \bmod m$

What about **modular division**?

- We can view division as multiplication: $a/b = a \cdot b^{-1}$.
- b^{-1} is the number that multiplies with b to give 1 mod m
- Does the reciprocal (multiplicative inverse) always exist?

Modular Inverses

Look back at the table for multiplication mod 15.
A number has an inverse if there is a 1 in its row or column.

Multiplication mod 13

$\times$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10	11	12
2	0	2	4	6	8	10	12	1	3	5	7	9	11
3	0	3	6	9	12	2	5	8	11	1	4	7	10
4	0	4	8	12	3	7	11	2	6	10	1	5	9
5	0	5	10	2	7	12	4	9	1	6	11	3	8
6	0	6	12	5	11	4	10	3	9	2	8	1	7
7	0	7	1	8	2	9	3	10	4	11	5	12	6
8	0	8	3	11	6	1	9	4	12	7	2	10	5
9	0	9	5	1	10	6	2	11	7	3	12	8	4
10	0	10	7	4	1	11	8	5	2	12	9	6	3
11	0	11	9	7	5	3	1	12	10	8	6	4	2
12	0	12	11	10	9	8	7	6	5	4	3	2	1

See all the inverses?

Totient function

This function has a first name; it's Euler.

Definition

The **Euler totient function**, written $\varphi(n)$, is the number of integers less than n that don't have any common factors with n.

Of course, this is also the number of invertible integers mod n.

When n is prime, $\varphi(n) = n - 1$. What about $\varphi(15)$?

Modular Exponentiation

This is the most important operation for cryptography!

Example: Compute $3^{2013} \bmod 5$.

Computing GCD's

The **greatest common divisor** (GCD) of two integers is the largest number which divides them both evenly.

Euclid's algorithm (c. 300 B.C.!) finds it:

GCD (Euclidean algorithm)

Input: Integers `a` and `b`
Output: `g`, the gcd of `a` and `b`

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Correctness relies on two facts:

- $\gcd(a, 0) = a$
- $\gcd(a, b) = \gcd(b, a \bmod b)$

Analysis of Euclidean Algorithm

Worst-case of Euclidean Algorithm

Definition

The **Fibonacci numbers** are defined recursively by:

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-2} + f_{n-1}$ for $n \geq 2$

The worst-case of Euclid's algorithm is computing $\gcd(f_n, f_{n-1})$.

Extended Euclidean Algorithm

Computing $\gcd(a, m)$ tells us whether $a^{-1} \bmod m$ exists.
This algorithm computes it:

Input: Integers `a` and `b`
Output: Integers `g`, `s`, and `t` such that `g = GCD(a,b)` and $as + bt = g$.

```
def xgcd(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        q, r = divmod(a, b)
        (g, s0, t0) = xgcd(b, r)
        return (g, t0, s0 - t0*q)
```

Notice: $bt = g \bmod a$. So if the gcd is 1, this finds the multiplicative inverse!

Cryptography

Basic setup:

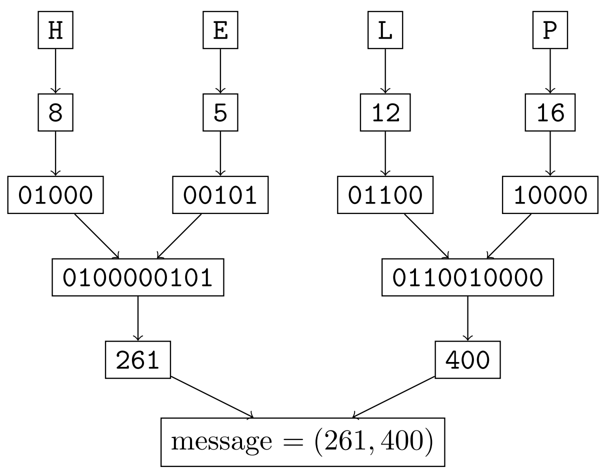
1. Alice has a message *M* that she wants to send to Bob.
2. She **encrypts** *M* into another message *E* which is gibberish to anyone except Bob, and sends *E* to Bob.
3. Bob **decrypts** *E* to get back the original message *M* from Alice.

Generally, *M* and *E* are just big numbers of a *fixed size*.
So the full message must be encoded into bits, then split into *blocks* which are encrypted separately.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Example of blocking



Public Key Encryption

Traditionally, cryptography required Alice and Bob to have a **pre-shared key**, secret to only them.

Along came the internet, and suddenly we want to communicate with people/businesses/sites we haven't met before.

The solution is **public-key cryptography**:

1. Bob has two keys: a public key and a private key
2. The public key is used for encryption and is published publicly
3. The private key is used for decryption and is a secret only Bob knows.

RSA

- RSA public key: A pair of integers (e, n)
- RSA private key: A pair of integers (d, n)
- *The n's are the same!*

RSA Encryption

The message M should satisfy $2 \leq M < n$
$E = M^e \bmod n$

RSA Decryption

$M = E^d \bmod n$

RSA Example

Alice wants to send the message "HELP" to Bob.

- Bob's public key: $(e, n) = (37, 8633)$
- Bob's private key: $(d, n) = (685, 8633)$

Encryption

"HELP" $\rightarrow$ (261, 400) $\rightarrow (261^e \bmod n, 400^e \bmod n) \rightarrow$ (5096, 1385)

Decryption

(5096, 1385) $\rightarrow (5096^d \bmod n, 1385^d \bmod n) \rightarrow$ (261, 400) $\rightarrow$ "HELP"

RSA Key Generation

We need d, e, n to satisfy $(M^d)^e = M \bmod n$ for any M.

Solution:

1. Choose 2 big primes p and q such that $n = pq$ has more than k bits (to encrypt k-bit messages).
2. Choose e such that $2 \leq e < (p-1)(q-1)$ and $\gcd((p-1)(q-1), e) = 1$.
3. Compute $d = e^{-1} \bmod \varphi(n)$ with the Extended GCD algorithm

RSA Analysis

We want to know how much the following cost:

- Generating a public/private key pair
- Encrypting or decrypting with the proper keys
- Decrypting *without* the private key

What would it take for this to be a secure cryptosystem?

Primality Testing

RSA key generation requires computing random primes.

- **Good news**: Primes are everywhere! In particular, about 1 in every k integers with k bits is prime.
- **Bad news**: Testing for primality seems difficult.
  We *need* to be able to do this faster than factorization!

Miller-Rabin Test

Input: Positive integer `n`
Output: `True` if `n` is prime, otherwise `False` (**probably**)

```
def probably_prime(n):
    a = random.randrange(2, n-1)
    d = n-1
    k = 0
    while d % 2 == 0:
        d = d // 2
        k = k + 1
    x = a**d % n
    if x**2 % n == 1: return True
    for r in range(1, k):
        x = x**2 % n
        if x == 1: return False
        if x == n-1: return True
    return False
```

Cost analysis for k-bit encryption

The main capabilities we need are:

- Generating random primes
- Computing XGCDs
- Modular exponentiation

The cost of **key generation** is $O(k^4)$

The cost of **encryption** and **decryption** are $O(k^3)$.

Security of RSA

We need to assert, **without proof**, that:

1. The only way to decrypt a message is to have the private key (d, n).
2. The only way to get the private key is to first compute $\varphi(n)$.
3. The only way to compute $\varphi(n)$ is to factor n.
4. There is no algorithm for factoring a number that is the product of two large primes in polynomial-time.

If all this is true, then as the key length k grows, the cost of factoring will always outpace the cost of encrypting/decrypting with the proper keys.

Summary

We acquired the following number-theoretic tools:

- Modular arithmetic (addition, multiplication, division, powering)
- GCDs and XGCDs with the Euclidean algorithm
- Primality testing (fast) and factorization (slow)

All these pieces are used in implementing and analyzing RSA.