CSCI 2011: Integers and Modular Arithmetic

Chris Kauffman

Last Updated:
Thu Jul 5 13:42:49 CDT 2018

Logistics

Reading: Rosen

- Now: 4.1 - 4.6
- Next: 5.1 - 5.3

Assignments

- A04: Due Tonight
- A05: Post Thursday
- Due Tuesday

Holiday Week

- No Discussion Wed
- No Quiz Thursday

Goals

- Finish Algorithms/Big-O
- Numbers, Bits, Encryption

Number Systems

- Must recall how decimal numbers "work"
- Digits 0-9 denote value, right-to-left placement indicates power of 10: base 10 system
- Works just as well for base 2 with digits 0/1

Decimal: Base 10 Example

Each digit adds on a power 10

$$
\begin{aligned}
80,345 =& 5 \times 10^0 + & \text{5 ones} \\
& 4 \times 10^1 + & \text{40 tens} \\
& 3 \times 10^2 + & \text{300 hundreds} \\
& 0 \times 10^3 + & \text{0 thousands} \\
& 8 \times 10^4 & \text{80,000 ...}
\end{aligned}
$$

So 5 + 40 + 300 + 80,000

Binary: Base 2 Example

Each digit adds on a power 2

$$
\begin{aligned}
11001_2 =& 1 \times 2^0 + & \text{1 ones} \\
& 0 \times 2^1 + & \text{0 twos} \\
& 0 \times 2^2 + & \text{0 fours} \\
& 1 \times 2^3 + & \text{8 eights} \\
& 1 \times 2^4 + & \text{16 sixteens} \\
=& 1 + 8 + 16 & = 25
\end{aligned}
$$

So, $11001_2 = 25_{10}$

Exercise: Convert Binary to Decimal

Base 2 Example:

$$
\begin{aligned}
11001 = & 1 \times 2^0 + & 1 \\
& 0 \times 2^1 + & 0 \\
& 0 \times 2^2 + & 0 \\
& 1 \times 2^3 + & 8 \\
& 1 \times 2^4 + & 16 \\
= & 1 + 8 + 16 & = 25
\end{aligned}
$$

So, $11001_2 = 25_{10}$

Try With a Pal

Convert the following two numbers from base 2 (binary) to base 10 (decimal)

- 111
- 11010
- 01100001

Answers: Convert Binary to Decimal

$$
\begin{aligned}
111_2 =& 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
=& 1 \times 4 + 1 \times 2 + 1 \times 1 \\
=& 7_{10} \\
11010_2 =& 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
=& 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
=& 26_{10} \\
01100001_2 =& 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 \\
& + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
=& 0 \times 128 + \times 64 + 1 \times 32 + 0 \times 16 \\
& + 0 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \\
=& 97_{10}
\end{aligned}
$$

Note: last example ignores leading 0's

The Other Direction: Base 10 to Base 2

Converting a number from base 10 to base 2 is easily done using repeated division by 2; keep track of **remainders**

Convert 124 to base 2:

$$
\begin{aligned}
124 \div 2 &= 62 && \text{rem } 0 \\
62 \div 2 &= 31 && \text{rem } 0 \\
31 \div 2 &= 15 && \text{rem } 1 \\
15 \div 2 &= 7 && \text{rem } 1 \\
7 \div 2 &= 3 && \text{rem } 1 \\
3 \div 2 &= 1 && \text{rem } 1 \\
1 \div 2 &= 0 && \text{rem } 1
\end{aligned}
$$

- Last step got 0 so we're done.
- Binary digits are in **remainders in reverse**
- Answer: `1111100`
- Check:
  $0 + 0 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 4 + 8 + 16 + 32 + 64 = 124$

Exercise: Decimal to Binary

124_{10} to Base 2

$$
\begin{aligned}
124 \div 2 &= 62 \qquad &\text{rem } 0\\
62 \div 2 &= 31 \qquad &\text{rem } 0\\
31 \div 2 &= 15 \qquad &\text{rem } 1\\
15 \div 2 &= 7 \qquad &\text{rem } 1\\
7 \div 2 &= 3 \qquad &\text{rem } 1\\
3 \div 2 &= 1 \qquad &\text{rem } 1\\
1 \div 2 &= 0 \qquad &\text{rem } 1
\end{aligned}
$$

- Remainders in reverse
- $124_{10} = 1111100_2$

- Convert 19_{10} to base 2
- Estimate # of steps it takes for the conversion of any number
- Speculate on an algorithm to convert numbers from base 10 to base 7, base 9, and base 13

Answers: Decimal to Binary

19_{10} to Base 2

$$
\begin{aligned}
19 \div 2 &= 9 && \text{rem } 1 \\
9 \div 2 &= 4 && \text{rem } 1 \\
4 \div 2 &= 2 && \text{rem } 0 \\
2 \div 2 &= 1 && \text{rem } 0 \\
1 \div 2 &= 0 && \text{rem } 1
\end{aligned}
$$

- $19_{10} = 10011_2$

- Convert 19_{10} to base 2
  - $19_{10} = 10011_2$
- Estimate # of steps it takes for the conversion of any number
  - Takes $\log_2 N$ steps where N is the number to convert
- Speculate on an algorithm to convert numbers from base 10 to base 7, base 9, and base 13
  - Repeatedly divide by the base, answer is remainder digits in reverse

Decimal, Hexadecimal, Octal, Binary

- Numbers exist independent of any writing system
- Can write the same number in a variety of bases
- Most common in computing are below
- Most programming languages have constant syntax for different bases such as the C examples below
- **Expectation**: Gain familiarity with doing conversions between bases as it will be useful in practice

	Decimal	Binary	Hexadecimal	Octal
Base	10	2	16	8
Mathematical	125	1111101_2	$7D_{16}$	175_8
C Prefix	None	`0b...`	`0x..`	`0...`
C Example	125	`0b1111101`	`0x7D`	`0175`

Octal: Base 8

Octal Basics

- Easy to convert Binary to Octal by grouping bits in groups of 3

  ```
  100 111 101 011
    4   7   5   3
  ```

- So $100111101011_2 = 4753_8$

File Permissions

- Octal commonly used for Unix file permissions
- 3 entities: user, group, other
- 3 permissions: read, write, execute

```
> chmod 665 somefile.txt

RESULT:
binary      octal
110110101 = 665
rw-rw-r-x somefile.txt
U  G  O
S  R  T
E  O  H
R  U  E
   P  R

Readable chmod version:
> chmod u=rw,g=rw,o=rx somefile.txt

Make file read/write/execute
by everyone on the system
> chmod 777 ur_mom.txt
```

Hexadecimal: Base 16

- Hex: compact way to write bit sequences
- One byte is 8 bits
- Hex uses 2 written characters per byte
- Each hex character represents 4 bits

```
|-----------+-----------------+-----|
| Byte      | Hex             | Dec |
|-----------+-----------------+-----|
| 0101 0111 | 57 = 5*16 + 7   |  87 |
| 5    7    |                 |     |
| 0011 1100 | 3C = 3*16 + 12  |  60 |
| 3    C=12 |                 |     |
| 1110 0010 | E2 = 14*16 + 2  | 226 |
| E=14 2    |                 |     |
|-----------+-----------------+-----|
```

Hex to 4 bit equivalence

Dec	Bits	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Binary Integer Addition/Subtraction

Adding/subtracting in binary works the same as with decimal EXCEPT that carries occur on values of 2 rather than 10j

```
ADDITION #1                 SUBTRACTION #1
   1 11     <-carries                ? <-carries
   0100 1010 = 74              0111 1001 = 121
 + 0101 1001 = 89            - 0001 0011 =  19
------------                ------------
   1010 0011 = 163             VVVVVVVVVVVVV
                               VVVVVVVVVVVVV
ADDITION #2                    VVVVVVVVVVVVV
   1111   1 <-carries                x12 <-carries
   0110 1101 = 109             0111 0001 = 119
 + 0111 1001 = 121           - 0001 0011 =  19
------------                ------------
   1110 0110 = 230             0110 0110 = 102
```

Exponentiation Algorithms

- Recall discussion of the `pow(x,y)` function for positive integers,
- Computes x^y
- Naive algorithm for this is to multiply in a loop

```
pow_linear(int base, int exp){
  ans = 1
  while(exp > 0){
    ans = ans * base
    exp = exp-1
  }
  return ans
}
```

- Speedier version of this exploits repeated squaring
- Example: compute $7^{14} = (7^6) \cdot (7^8) = (7^6) \cdot ((7^2)^2)^2$
- Latter term is squares 7, then squares result (49^2), then squares again
- **Fast Exponentiation Algorithm** exploits this via add/even powers

Fast Exponentiation Algorithm

```
pow_fast(int base, int exp){
  ans = 1
  power = base
  while(exp > 0){
    if(exp even){  # even
      power = power*power
      exp = exp / 2
    }
    else{          # odd
      ans = ans*power
      power = power*power
      exp = (exp-1) / 2
    }
  }
  return ans
}
```

Variable values at the end of loop iterations

Iter	exp	ans	power
init	14	$1 = 7^0$	$7 = 7^1$
1	7	$1 = 7^0$	$49 = 7^2$
2	3	$49 = 7^2$	$2401 = 7^4$
3	1	$117649 = 7^6$	$\ldots = 7^8$
4	0	$\ldots = 7^{14}$	$\ldots = 7^{16}$

Exponentiation in Binary

- Easy to detect even/odd in binary by examining bits
- Leads to the following adaptation of the algorithm
- What is the runtime complexity of this equivalent version?

```
pow_fast_bin(int base, int exp){
  ans = 1
  power = base
  for(i=0; i < #bits(exp); i++){
    if(exp_bits[i] == 0){# even
      power = power*power
    }
    else{                # odd
      power = power*power
      ans = ans*power
    }
  }
  return ans
}
```

- Variable values at the end of loop iterations
- Binary exp $= 14_{10} = 1110_2$

Iter	bits exp	ans	power
init	14	$1 = 7^0$	$7 = 7^1$
1	0	$1 = 7^0$	$49 = 7^2$
2	1	$49 = 7^2$	$2401 = 7^4$
3	1	$117649 = 7^6$	$\ldots = 7^8$
4	1	$\ldots = 7^{14}$	$\ldots = 7^{16}$

Modular Arithmetic

- Recall Modulo: $30 \bmod 4 = 2 \bmod 4$ as remainder of $30 \div 4 = 7$ rem 2
- Fact: in modulo systems,

$$c = a \cdot b, c \bmod m = (a \bmod m) \cdot (b \bmod m) \bmod m$$

- Example

$$\begin{aligned} 30 &= 2 \cdot 15 \\ 30 \bmod 4 &= (2 \bmod 4) \cdot (15 \bmod 4) \bmod 4 \\ &= 2 \cdot 3 \bmod 4, \text{ because } 15 \div 4 = 3 \text{ rem } 3 \\ &= 6 \bmod 4 \\ &= 2 \bmod 4 \end{aligned}$$

Fast Modular Exponentiation

- Important Applications in Cryptography
- Will Examine them next week

```
pow_fast_mod(int base, int exp,
             int mod)
{
  ans = 1
  power = base % mod
  for(i=0; i < #bits(exp); i++){
    if(exp_bits[i]==0){ # even
      power = power*power % mod
    }
    else{               # odd
      power = power*power % mod
      ans = ans*power % mod
    }
  }
  return ans
}
```

- Binary exp $= 14_{10} = 1110_2$
- Compute 7^{14} **mod** 9
- `pow_fast_mod(7,14,9)`

Iter	bits exp	ans	power
init	14	$1 = 7^0$ % 9	$7 = 7^1$ % 9
1	0	$1 = 7^0$ % 9	$4 = 7^2$ % 9
2	1	$4 = 7^2$ % 9	$7 = 7^4$ % 9
3	1	$1 = 7^6$ % 9	$4 = 7^8$ % 9
4	1	$4 = 7^{14}$ % 9	$7 = 7^{16}$ % 9

Modulo and Negative Numbers

Definition: $a \div b = q \textbf{ rem } r \leftrightarrow a = q \times b + r$

- q is the quotient
- r is the remainder

Differences arise in practice

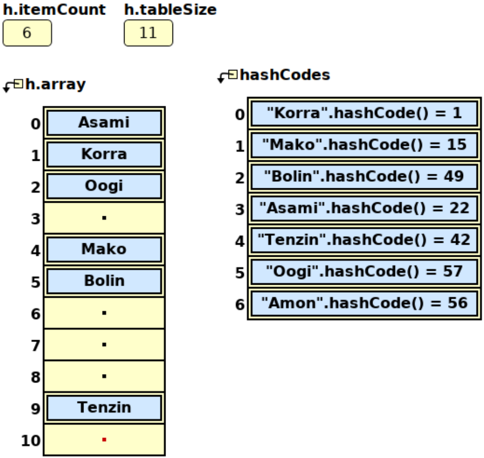
Mathematical Modulus Defined to have only positive remainders in most mathematical contexts

Programming Modulus `a % b` may produce negatives, properly the "remainder operator", truncates division towards 0

Math	Quot	Rem	Programming	Quot	Rem
$-11 \div 3$	-4	$+1$	`q=-11/3; r=-11%3;`	-3	-2
$-10 \div 4$	-3	$+2$	`q=-10/4; r=-10%4;`	-2	-2
$-20 \div 11$	-2	$+2$	`q=-20/11; r=-20%11;`	-1	-9

Hash Tables use Modulus

- A **Hash Table** is an array, usually with Prime Number length
- Objects entering table have associated key, their **Hash Code**
- Hash Codes may be large, modulus used to locate their slot in table, must resolve collisions
  - Why is Mako at slot 4?
  - Tenzin at slot 9?
  - Where will Amon go?
  - Algorithm to compute codes?



Modulus to Generate Pseudorandom Numbers

A classic random number generator from *The C Programming Language* by Kernighan and Ritchie

- Example of a **linear congruential method**: modulo used to restrict range to `int`
- Is there anything random about it? How could one introduce randomness?

```
/* Tracks state of random number generator */
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand() {
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed) {
  next = seed;
}
```

Encryption

- Modular Arithmetic has big play in cryptography
- Obscure information except for the intended recipient
- Usually involves a shared secret or **private key**
- Caesar Cipher
  - Constant shift of characters
  - Secret key is the `shift` amount
  - Not very strong encryption
- Vigenere Cipher
  - Variable shift of characters
  - Secret key is the pass phrase
- A good video on Caesar and Vigenere Ciphers for beginners

Exercise: Caesar Cipher Example

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L	M
13	14	15	16	17	18	19	20	21	22	23	24	25
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Example 1

Secret Key $+4$

Plain Text `MARIO`

Encrypted `QEVMS`

Work It

Secret Key $+9$

Encrypted `CXJM`

Plain Text `????`

Example 2

Secret Key $+7$

Plain Text `LUIGI`

Encrypted `SBPNP`

Notice the wrapping of `U`

- U$\rightarrow$ 21;
- (20+7) % 26 = 27 % 26 = 1
- 1 $\rightarrow$ `B`

Vigenere Cipher

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L	M
13	14	15	16	17	18	19	20	21	22	23	24	25
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Don't use a single key, use a passphrase

Secret Key `TOAD` → `[19, 14, 0, 3]`

Plain Text `PRINCESS`

Encrypted `IFJQVSSV`

Original	P	R	I	N	C	E	S	S
Numbers	15	17	8	13	2	4	18	18
Secret Key	T	O	A	D	T	O	A	D
Numbers	19	14	0	3	19	14	0	3
Sums	34	31	8	16	21	18	18	21
modulo 26	8	5	I	16	21	18	18	21
Encrypted	I	F	J	Q	V	S	S	V

Limits of Classical Cryptography

- Caesar and Vigenere are weak ciphers: how would one crack them?
- Improved versions are extremely strong: takes millenia to crack
- Drawback: **shared secrets** like single private key limits applicability - WHY?
- Modern commerce requires no a priori shared secret, must be able to develop a shared secret as part of the system
- **Public Key Cryptography** solves this

The RSA System

Based on the following scheme

$$(m^e)^d \equiv m \bmod n$$

- m is a message, numeric form, to be encrypted/decrypted
- n is very large number, product of two primes
- e is a **public key**, used for encryption, **shared freely**
- d is a **private key**, used for decryption, **kept secret**
- e, d are chosen as **inverses** of one another under modulo n

Notice the following

- Need for fast modular exponentiation
- Need to know how to find d, e efficiently

Encryption in RSA

- Alice wants to send Bob a message

```
m = HELP = H E  L P
m =        0704 1115
block#     1    2
```

- Requests Bob's
  - Public Encryption Key $e = 13$
  - Modulo base $n = 2537$
- Alice computes **ciphered** message in two blocks

$$c = m^e \bmod n$$
$$c_1 = 0704^{13} \bmod 2537 = 0981$$
$$c_2 = 1115^{13} \bmod 2537 = 0461$$

Security

- Alice sends Bob message 0981 0461 over an open channel like the internet
- Evil Eve intercepts the messages so has e, n, c
- Eve tries decrypting with e which yields gibberish

$$0981^{13} \bmod 2537 = 1607$$
$$0461^{13} \bmod 2537 = 1244$$
$$1607\ 1244 = \text{QHMS ???}$$

- Eve cannot determine m as she lacks the decryption key d Bob has kept secret

Decryption in RSA

- Bob receives the message 0981 0461 from Alice
- Bob decrypts it using his secret decryption key $d = 937$

$$m = c^d \textbf{ mod } n = (m^e)^d \textbf{ mod } n$$
$$m_1 = 0981^{937} \textbf{ mod } 2537 = 0704$$
$$m_2 = 0461^{937} \textbf{ mod } 2537 = 1115$$
$$0704\ 1115 = \text{HELP}$$

- If Bob wants to send a safe return message, requests from Alice
  - Alice's public encryption key
  - Alice's modulo base
- Bob encrypts with Alice's key and replies

Fast Modular Exponentiation

- Encryption and Decryption use fast modular exponentiation
- Actual keys/modulo base are not 4 digits but 100-300 digits long or 1024-4096 bits long

Aspects of RSA

Key Generation

- RSAs successful because trios e, d, n can be **efficiently** produced such that for all m

$$(m^e)^d \textbf{ mod } n = m \textbf{ mod } n$$

- Involves picking two large prime numbers p, q

Key Generation: Finding Encryption/Decryption Pairs

Part of RSAs success that trios e, d, n can be **efficiently** produced such that for all m: $(m^e)^d \textbf{ mod } n = m \textbf{ mod } n$

Key Generation involves the following steps

1. Pick two large prime numbers p, q: use primality tests such as the **Sieve of Eranthoses** (super-linear time)
2. Find the least common multiple $t = \text{lcm}(p-1, q-1)$, called **totient** (log time)
3. Pick encryption key $e < t$ such that e is relatively prime to t (no common factors, log time)
4. Find d so that $e \cdot d \textbf{ mod } t = 1$
   - d, e are **modular multiplicative inverses**
   - Use the **Extended Euclidean Algorithm** (log time)
5. Publish public modular base $n = p \cdot q$ and public encryption key e
6. Keep private decryption key d secret

Fermat's Little Theorem used to show e, d, n have desired properties if chose by the above scheme

Strength of RSA

- Strength of RSA is based on the difficulty of **factoring** large integers: determine $n = f_1 \cdot f_2 \cdot f_3 \cdots$ with f_i prime
- During key selection, picked primes p, q, published base $n = p \cdot q$
- During key selection used secret knowledge of p, q to generate encryption/decryption pair
- Attackers only know n: may try to factor n to determine p, q but **no polynomial time algorithm exists** for integer factorization on deterministic CPUs
- Largest published cracked RSA key is 768 bits, took 1500 CPU **years** (2 years with many parallel computers)
- Practical keys/modulo base are 100-300 digits / 1024-4096 bits long making them relatively secure
- On the horizon: Shor's Algorithm factors numbers in polynomial time on non-deterministic machines like Quantum computers, may someday practically impact your Amazon orders

Take-Home

- Number theory was once thought the last bastion of truly pure mathematics
- Many interesting algorithms exist within it
- With the advent of computing and communication networks, number theory has **many** practical applications these days