CHAPTER 3
THE FUNDAMENTALS: ALGORITHMS, THE INTEGERS AND MATRICES

Outline

- **Content**
  - Algorithms
  - The Growth of Functions
  - Complexity of Algorithms
  - The Integers and Division
  - Primes and Greatest Common Divisors
  - Integers and Algorithms
  - Applications of Number Theory
  - ~~Matrices~~
- **Reading**
  - Chapter 3

Most of the following slides are by courtesy of Prof. J.-D. Huang and Prof. M.P. Frank

Algorithms

- **An algorithm is a finite set of precise instructions for performing a computation or for solving a problem**
- **E.g.,**
  - +, -, *, /
  - cooking recipe

- **A computer program is simply a description of an algorithm in a computer language (e.g., C++)**
  - A program implements (or “is an implementation of”) its algorithm

Finding the Maximum Element–in English

- **Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.**
- **Solution 1: Describe the sequence of steps used.**
  1. Set the temporary maximum equal to the first integer in the sequence
  2. Compare the next integer in the sequence to the temporary maximum, and if it is larger than the temporary maximum, set the temporary maximum equal to this integer.
  3. Repeat the previous step if there are more integers in the sequence
  4. Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence.

Finding the Maximum Element–Pseudocode

- **Pseudocode: an intermediate step between an English language description and an implementation in a programming language**
  - Why not using a computer language directly?
  - Too complicated
  - An algorithm in one language cannot be interpreted in others
- **Solution 2: pseudocode**

procedure *max*(a_1, …, a_n: integers)
max := a_1
for *i* := 2 **to** *n*
    if *max* < a_i **then** *max* := a_i
{*max* is the largest element}

Properties of Algorithms

- **Keep the following in mind if you try to describe an algorithm**

1. **Input: Information or data that comes in**
2. **Output: Information or data that goes out**
3. **Definiteness: Precisely defined**
4. **Correctness: Outputs correctly w.r.t. inputs**
5. **Finiteness: Won't take forever to run**
6. **Effectiveness: Individual steps are all doable**
7. **Generality: Works for many possible inputs**

Searching an number in an Ordered List (1/2)

- **The searching problem: Locate an element *x* in an sorted list of distinct elements $a_1, \ldots, a_n$, or determine that it is not in the list.**

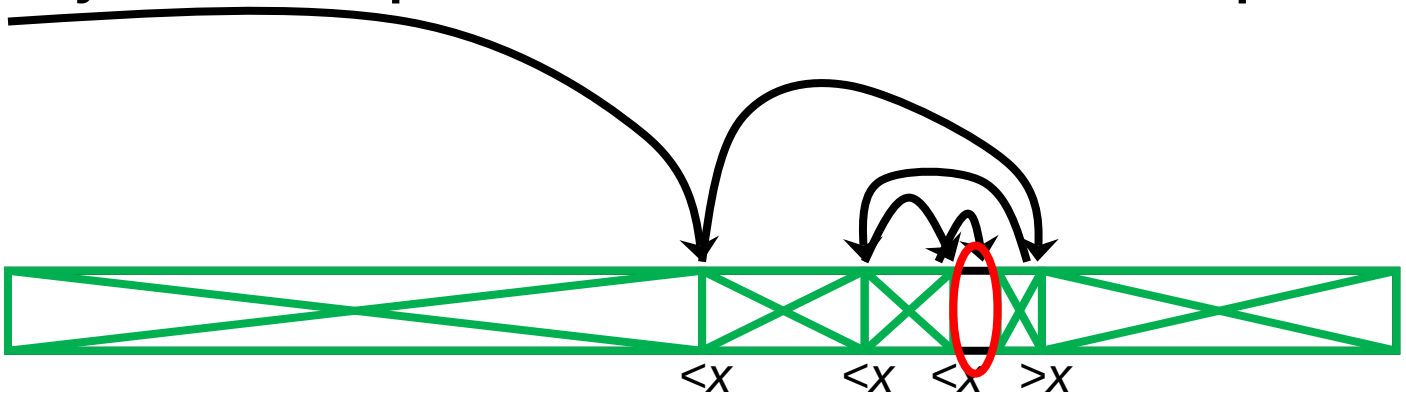
- **Solution 1: Linear search**
  - Compare *x* with each element of the list until it is found.

procedure *linear search*(*x*: integer, $a_1,\ldots, a_n$: distinct integers)
i := 1
while ($i \leq n$ **and** $x \neq a_i$)
  i := *i* + 1
if $i \leq n$ **then** *location* := *i*
else *location* := 0
{*location* is the subscript of the term that equals *x*, or is 0 if *x* is not found}

  - Can we do better?

Searching an number in an Ordered List (2/2)

- **Binary search: Explore the feature of the sorted sequence**



procedure *binary search*(*x*: integer, a_1,…, a_n: distinct integers)
l := 1 {*l* is left endpoint of search interval}
r := *n* {*r* is right endpoint of search interval}
while *l* < *r*
    m := $\lfloor (l+r)/2 \rfloor$ {*m* is the middle point of search interval}
    if *x* > a_m **then** *l* := *m* +1
    else *r* := *m*
if *x* = a_l **then** *location* := *l*
else *location* := 0
{*location* is the subscript of the term that equals *x*, or is 0 if *x* is not found}

Greedy Algorithms

- **Greedy algorithms are algorithms that make what seems to be the best choice at each step**

- **Some greedy algorithms lead to optimal solutions**
  - Few algorithms!
- **However, some do not**
  - Usually, we are in this case.

- **Detailed discussion in the course Algorithms**
  - Welcome to take “Algorithms”!

Change Making

- **Problem: making *n* cents change with quarters, dimes, nickels, and pennies, and using the least total number of coins.**

- **Greedy algorithm: make a locally optimal choice at each step, i.e., choose the largest denomination possible to add to the pile of change without exceeding *n* cents.**

procedure *change*(c_1,…, c_r: values of denominations of coins, where $c_1 > \ldots > c_r$; n: a positive integer)
for i := 1 **to** r
  while $n \geq c_i$
    add a coin with value c_i to the change
    $n = n - c_i$

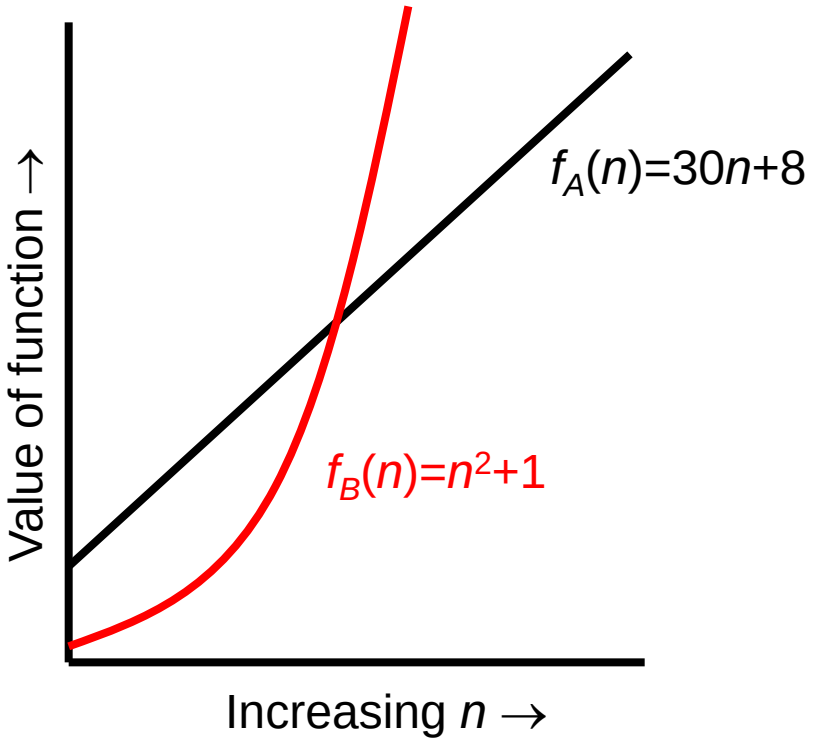
- **Optimal!**
  - Proof by contradiction

Order of Growth (1/3)

- **Suppose you are designing a web site to process user data (e.g., financial records)**
- **Suppose database program *A* takes $f_A(n)=30n+8$ microseconds to process any *n* records, while program *B* takes $f_B(n)=n^2+1$ microseconds to process the *n* records**
- **Which program do you choose, knowing you'll want to support millions of users?**

Order of Growth (2/3)

- **On the chart, as you go to the right, a faster growing function eventually becomes larger...**



Order of Growth (3/3)

- **We say $f_A(n)=30n+8$ is order n, or $O(n)$**
  - it is roughly proportional to n
- **$f_B(n)=n^2+1$ is order n^2, or $O(n^2)$**
  - it is roughly proportional to n^2

We will fix big-O notation later

- **Any $O(n^2)$ function is faster-growing than any $O(n)$ function**
- **For large numbers of user records, the $O(n^2)$ function will always take more time**

Definition: $O(g)$, at most order g

- **Let f and g be any function $\mathbb{R}\to\mathbb{R}$.**
  Define "at most order g", written $O(g)$, to be:
  $$\{f: \mathbb{R}\to\mathbb{R} \mid \exists c, k: \forall x > k: f(x) \le cg(x)\}.$$

$O(g)$: read big-O of g

  - Beyond some point k, function f is at most a constant c times g.
- **"f is *at most order* g", or "f is $O(g)$", or "$f = O(g)$" all just mean that $f \in O(g)$.**
  - $f = O(g)$ is commonly used
- **Sometimes the phrase "at most" is omitted.**
- **Intuitively, big-O gives an upper bound**

Points about the Definition

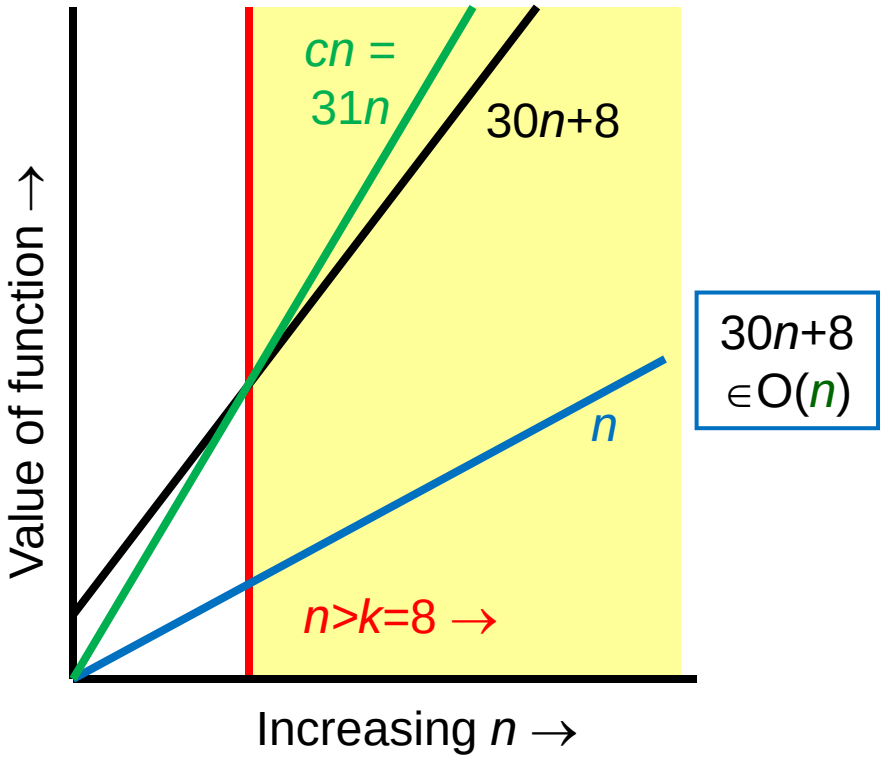
- **Note that *f* is O(*g*) so long as any values of *c* and *k* exist that satisfy the definition.**
- **But: The particular *c*, *k*, values that make the statement true are not unique: Any larger value of *c* and/or *k* will also work.**
- **You are not required to find the smallest *c* and *k* values that work. (Indeed, in some cases, there may be no smallest values!)**
- **However, you should prove that the values you choose do work.**

“Big-O” Proof Examples

- **Show that $30n+8$ is $O(n)$.**
  - Show $\exists c,k: \forall n>k: 30n+8 \leq cn$.
    - Let $c = 31$, $k = 8$.
    - Assume $n > k = 8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.
- **Show that n^2+1 is $O(n^2)$.**
  - Show $\exists c,k: \forall n>k: n^2+1 \leq cn^2$.
    - Let $c = 2$, $k = 1$.
    - Assume $n > 1$. Then $cn^2 = 2n^2 = n^2+n^2 > n^2+1$, or $n^2+1< cn^2$.

Big-O Example, Graphically

- **Note 30*n*+8 isn't less than *n* anywhere (*n*>0).**
- **It isn't even less than 31*n* everywhere.**
- **But it is less than 31*n* everywhere to the right of *n*=8.**



Useful Facts about Big-O

- **Big-O, as a relation, is transitive:**
  $f=O(g) \wedge g=O(h) \rightarrow f=O(h)$

- $\forall c>0$,
  $O(cf)=O(f+c)=O(f-c)=O(f)$

- $f_1=O(g_1) \wedge f_2=O(g_2) \rightarrow$
  $f_1 f_2 = O(g_1 g_2)$
  $f_1+f_2 = O(g_1+g_2)$
  $= O(\max(g_1,g_2))$ (Very useful!)

Common Big-O

- **If $f(x) = a_n x^n + a_{n-1}x^{n-1}+\ldots+a_1 x + a_0$, and $a_n \neq 0$ then $f(x) = O(x^n)$**
  - Omit lower-order terms
  - Delete the ending constant

- **$n! = O(n^n)$**
  - Why?

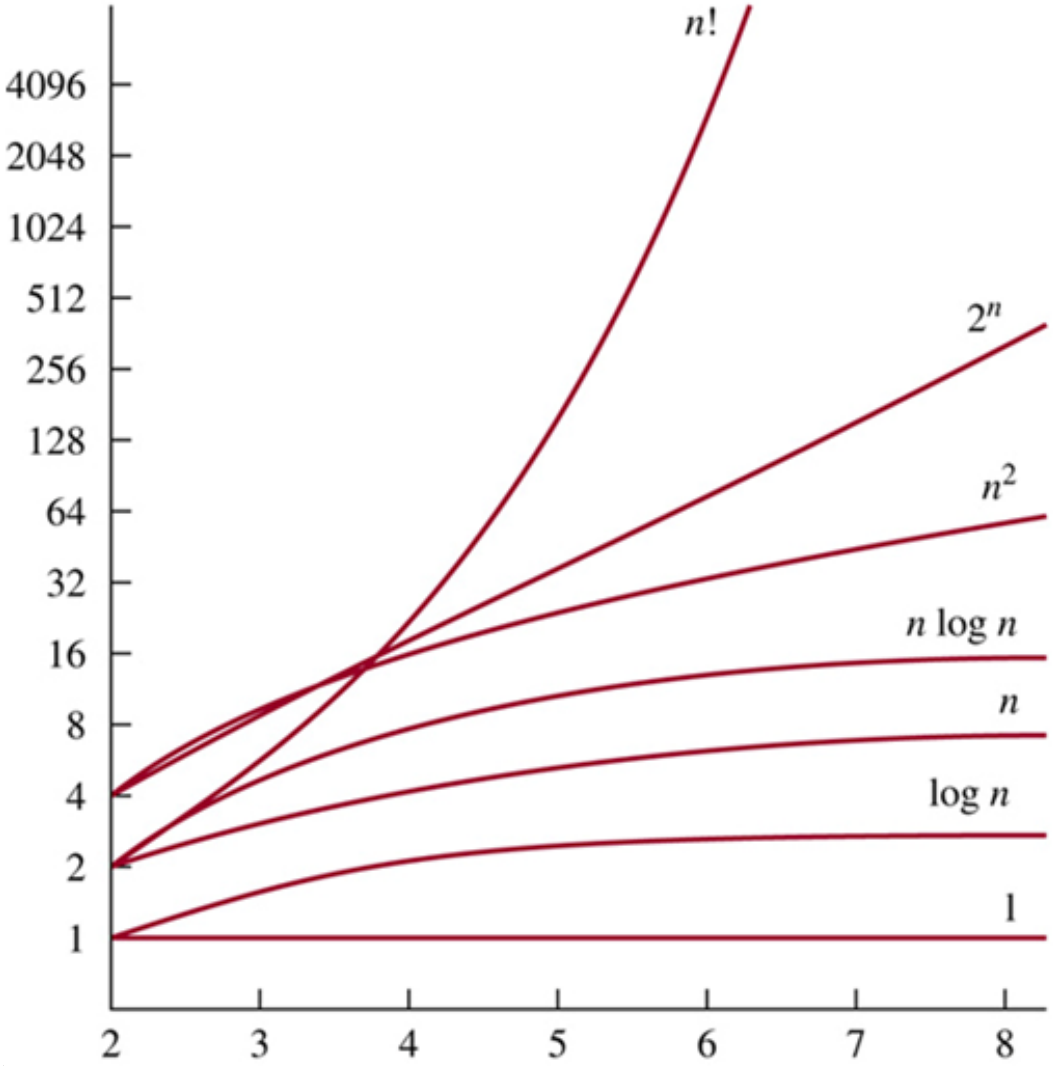
- **$\log n! = O(n \log n)$**
  - Why?

Examples

- **Q: Give a big-O for $f(n) = 3n \log(n!) + (n^2 + 3) \log n$**
- **A:**

- **Q: Give a big-O for $f(x) = (x+1) \log(x^2+1) + 3x^2$**
- **A:**

Big-O Growth Rate



$1 < \log n < n < n \log n < n^2 < 2^n < n!$

Definition: $\Omega(g)$, at Least Order g

- **Let *f* and *g* be any function R→R.**
  Define "at least order *g*", written $\Omega(g)$, to be:
  $\{f: R \to R \mid \exists c, k: \forall x > k: f(x) \geq cg(x)\}$.

$\Omega(g)$: read big-Omega of *g*

  - Beyond some point *k*, function *f* is at least a constant *c* times *g*.

- **Intuitively, big-Omega gives a lower bound**

- ***f*(*x*) is $\Omega(g(x))$ $\leftrightarrow$ *g*(*x*) is O(*f*(*x*))**

Definition: $\Theta(g)$, Exactly Order g

- **Let f and g be any function $R \to R$.**
  Define "is (exactly) order g", written $\Theta(g)$, to be:
  $$\{f: R \to R \mid \exists c_1 c_2 k\ \forall x > k: c_1 g(x) \le f(x) \le c_2 g(x)\}.$$

$\Theta(g)$: read big-Theta of g

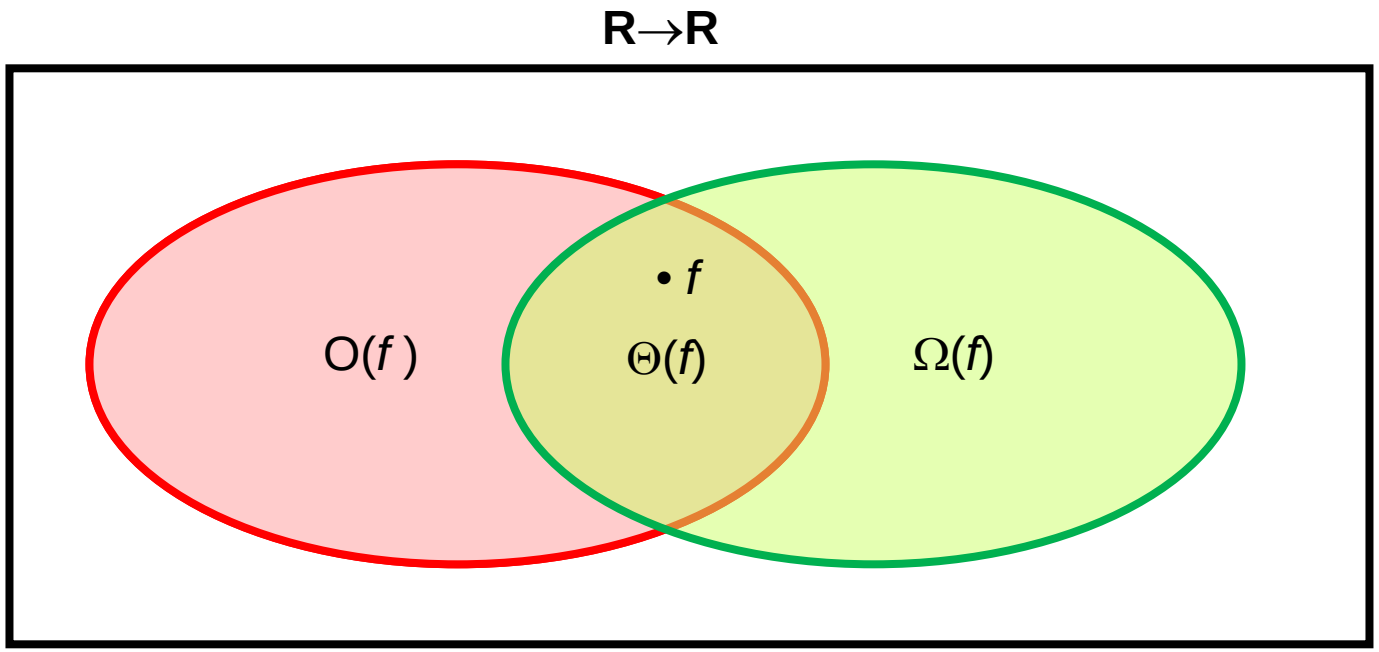
  - "Everywhere beyond some point k, $f(x)$ lies in between two multiples of $g(x)$."

- **If $f=O(g)$ and $g=O(f)$ then we say "g and f are of the same order" or "f is (exactly) order g" and write $f=\Theta(g)$.**
- **$f(x)$ is $\Theta(g(x))$ $\leftrightarrow$ $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$**

- **Intuitively, big-Theta gives a tight bound**

Relations Between the Relations

- **Subset relations between order-of-growth sets.**



Complexity of Algorithms (1/2)

- **An analysis of the time and memory required to solve a problem of a particular size**
  - time complexity: running time
  - space complexity: memory requirement
- **E.g.,**
  - Find the maximum
  - Linear search

Complexity of Algorithms (2/2)

- **Complexity analysis in 3 cases**
  - worst case: usually used
  - average (typical) case
  - best case

- **Detailed discussions in Data Structures and Algorithms**

Integers and Divisions

Integers and Divisions

- **Of course you already know what the integers are, and what division is**
- **But, there are some specific notations, terminology, and theorems associated with these concepts which you may not know**
- **These form the basics of number theory**
  - Vital in many important algorithms today
    - Hash functions
    - Cryptography
    - Digital signatures

Divide, Factor, Multiple

- **Let $a,b \in \mathbb{Z}$ with $a \neq 0$.**
- **$a|b$ ≡ "a divides b" :≡ "$\exists c \in \mathbb{Z}$: $b=ac$"**
  - "There is an integer c such that c times a equals b."
  - E.g., $3|-12 \Leftrightarrow$ **True**, but $3|7 \Leftrightarrow$ **False**.
- **If a divides b, then we say a is a factor or a divisor of b, and b is a multiple of a.**

- **"b is even" :≡ $2|b$.**
- **Q: Is 0 even? Is −4?**

Properties of Divisibility

- $\forall a,b,c \in \mathbf{Z}$:
  1. $a|0$
  2. $(a|b \wedge a|c) \rightarrow a \mid (b + c)$
  3. $a|b \rightarrow a|bc$
  4. $(a|b \wedge b|c) \rightarrow a|c$

- **Pf of 2:**
  - By definition of |, we know $\exists s$: $b=as$, and $\exists t$: $c=at$.
    Let s, t, be such integers.
  - Then $b+c = as + at = a(s+t)$, so
    $\exists u$: $b+c=au$, namely $u=s+t$. Thus $a|(b+c)$.

Prime Numbers

- **An integer $p>1$ is prime iff it is not the product of any two integers greater than 1:**
  $$p>1 \wedge \neg\exists a, b \in \mathbf{N}: a>1, b>1, ab=p.$$

- **The only positive factors of a prime *p* are 1 and *p* itself.**
  - E.g., some primes: 2,3,5,7,11,13...

- **Non-prime integers greater than 1 are called composite, because they can be composed by multiplying two integers greater than 1.**

Fundamental Theorem of Arithmetic

- **Every positive integer > 1 has a unique representation as the product of a non-decreasing series of 1 or more primes**
  - 2 = 2 (product of series with one element 2)
  - 4 = 2·2 (product of series 2,2)
  - 2000 = 2·2·2·2·5·5·5; 2001 = 3·23·29;
    2002 = 2·7·11·13; 2003 = 2003
- **This process is called prime factorization**
- **If *n* is a composite integer, then *n* has a prime divisor less than or equal to $\sqrt{n}$**
  - Usage: how to test whether a given number is a prime?

An Application of Primes

- **When you visit a secure web site (https:…), the browser and web site may be using a technology called RSA encryption.**
- **This public-key cryptography scheme involves exchanging public keys containing the product *pq* of two random large primes *p* and *q* (a private key) which must be kept secret by a given party.**
- **So, the security of your day-to-day web transactions depends critically on the fact that all known factoring algorithms are intractable!**
- **Note: There is a tractable quantum algorithm for factoring; so if we can ever build big quantum computers, RSA will be insecure**

The Division Algorithm (Theorem)

- **Really just a theorem, not an algorithm…**
  - The name is used here for historical reasons.

- **For any integer dividend *a* and divisor *d*≠0, there is a unique integer quotient *q* and remainder *r*∈N such that *a* = *dq* + *r* and 0 ≤ *r* < |*d*|.**
  - $q = a \text{ div } d$
  - $r = a \bmod d$

- **We can find *q* and *r* by: $q=\lfloor a/d \rfloor$, $r=a-qd$.**

Greatest Common Divisor (GCD)

- **The greatest common divisor gcd(*a*, *b*) of integers *a*, *b* (not both 0) is the largest (most positive) integer *d* that is a divisor both of *a* and of *b***
  - $d = \gcd(a, b) = \max\{d \mid d|a \wedge d|b\} \Leftrightarrow$
    $d|a \wedge d|b \wedge \forall e \in \mathbf{Z}, (e|a \wedge e|b) \rightarrow d \geq e$

- **E.g., gcd(24, 36)=?**
- **A: Positive common divisors: 1, 2, 3, 4, 6, 12.**
  Greatest is 12

GCD Shortcut

- **If the prime factorizations are written as**
  $a = p_1^{a_1} p_2^{a_2} \ldots p_n^{a_n}$ **and** $b = p_1^{b_1} p_2^{b_2} \ldots p_n^{b_n}$
- **then the GCD is given by:**
  $\gcd(a,b) = p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \ldots p_n^{\min(a_n,b_n)}$.

- **E.g.,**
  - a=84=2·2·3·7 = $2^2 \cdot 3^1 \cdot 7^1$
  - b=96=2·2·2·2·2·3 = $2^5 \cdot 3^1 \cdot 7^0$
  - gcd(84,96) = $2^2 \cdot 3^1 \cdot 7^0 = 4 \cdot 3 \cdot 1 = 12$

Relatively Primality

- **Integers *a* and *b* are relatively prime if their gcd is 1**
- **E.g., 8 and 9**
  - Neither 8 nor 9 is prime
  - However, 8 and 9 are relatively prime
    - gcd(8, 9) = 1

- **A set of integers {a_1, a_2, …} is (pairwise) relatively prime if all pairs a_i, a_j, $i \neq j$, are relatively prime**

Least Common Multiple (LCM)

- **lcm(*a*, *b*) of positive integers *a* and *b* is the smallest positive integer that is a multiple both of *a* and of *b***
  - $m = \text{lcm}(a, b) = \min\{m \mid a|m \wedge b|m\} \Leftrightarrow$
    $a|m \wedge b|m \wedge \forall n \in \mathbf{Z}: (a|n \wedge b|n) \rightarrow (m \leq n)$

- **E.g., lcm(6, 10) = 30**

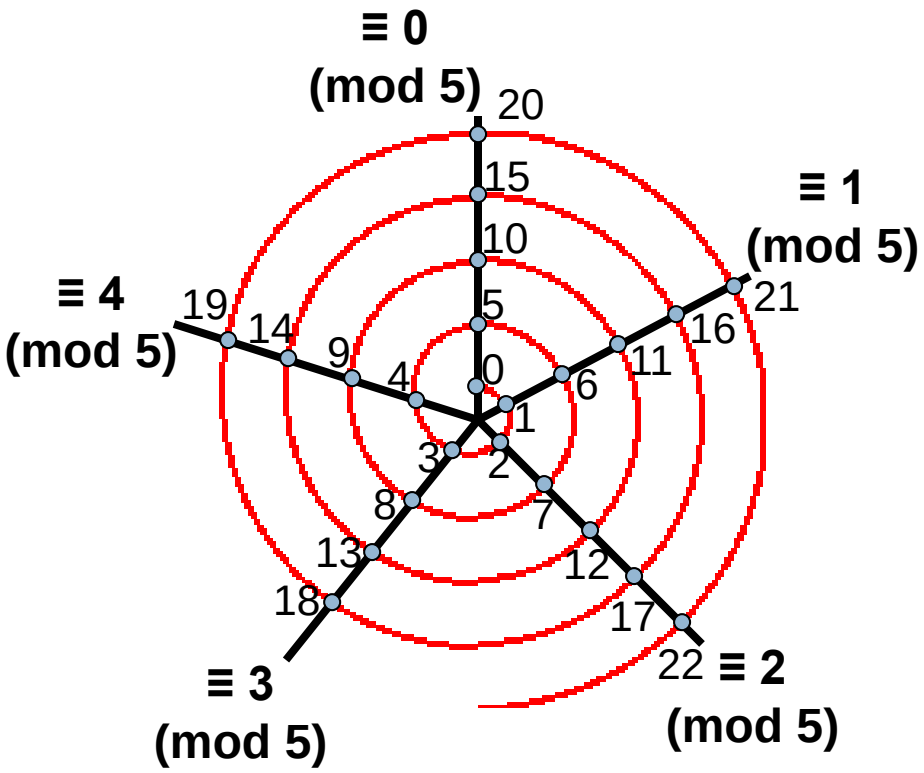
Modulo and Congruence

- **Let *a*, *b*∈Z, *m*∈Z⁺.**
  Then *a* is congruent to *b* modulo *m*, written "*a*≡*b* (mod *m*)",
  iff *m* | *a*-*b*
  - i.e., (*a*–*b*) mod *m* = 0.

- **(Note: this is a different use of "≡" than the meaning is defined as we've used before.)**

Spiral Visualization of mod

- **Example shown: modulo-5 arithmetic**



Useful Congruence Theorems

$a \equiv b \pmod m$: $m \mid a-b$

- **Let $a, b \in \mathbb{Z}, m \in \mathbb{Z}^+$. Then**
  $a \equiv b \pmod m \Leftrightarrow \exists k \in \mathbb{Z}\ a = b + km$.

- **Let $a, b, c, d \in \mathbb{Z}, m \in \mathbb{Z}^+$. Then**
- **if $a \equiv b \pmod m$ and $c \equiv d \pmod m$, then:**
  - $a+c \equiv b+d \pmod m$, and
  - $ac \equiv bd \pmod m$

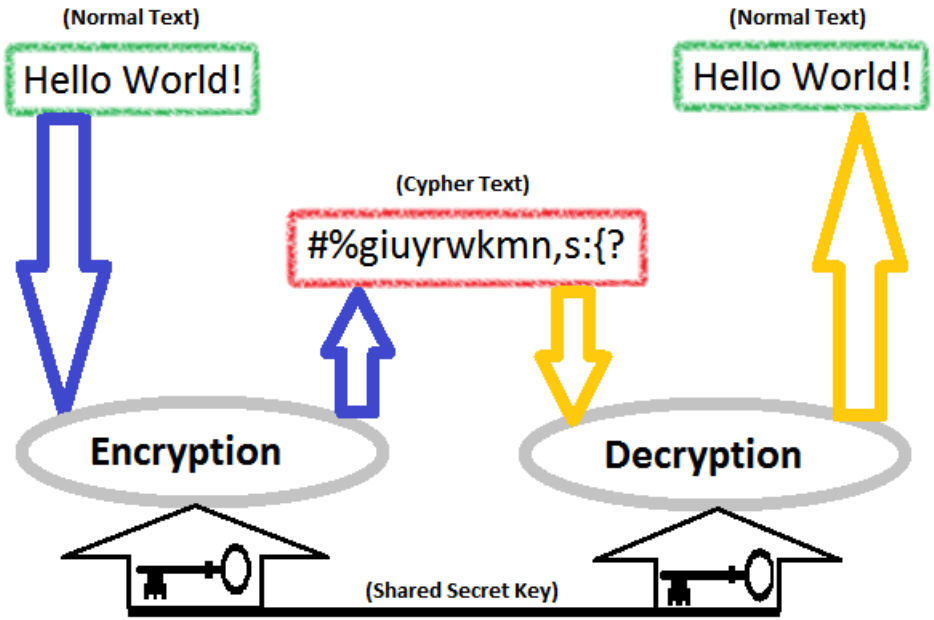
Applications of Congruence

- **Hashing functions**
  - Learn more in Data Structures

- **Pseudorandom number generation**
  - $x_{n+1}=(a \cdot x_n+c) \bmod m$, where
  - a (multiplier): $2 \leq a < m$,
  - c (increment): $\mathbf{0} \leq c < m$,
  - x_0 (seed): $0 \leq x_0 < m$,

Cryptology Applications (1/2)

- **Monographic substitution**
  - Is enciphering based on a permutation of an alphabet $\pi : A \rightarrow A$.
  - ciphertext is obtained from plaintext by replacing each occurrence of each letter by its substitute
    - `letter A B C D E F · · · X Y Z`
    - `subst  Q W E R T Y · · · B N M`



Cryptology Applications (2/2)

- **A cyclic monographic substitution cipher**
  - if the letters of the alphabet are represented by numbers 0, 1, ..., 25 and there is a number m such that $\pi(n) = (m + n) \bmod 26$
- **Julius Caesar encrypted military messages by cyclic monographic substitution**
  - what is “HW WX EUXWH” supposed to mean?

Integer Algorithms

- **Self-study**
- **Representations of integers in different bases**
  - binary, hexadecimal, …
  - base conversions
- **Algorithm for integer operations**
  - addition
  - multiplication
  - division and modulus

Modular Exponentiation

- **In cryptograph, it's important to efficiently find b^n mod m, where b, n, and m are large integers**
  - e.g., 2^{644} mod 645
- **It's impractical to calculate b^n first**

- **Instead,**
  - find $n = (a_{k-1} \cdot \cdot \cdot a_1\ a_0)_2$, (binary representation)
  - find b mod m, b^2 mod m, b^4 mod m, …, $b^{2^{k-1}}$ mod m
  - multiply those terms (b^{2^j} mod m) where $a_j = 1$
  - find the remainder of the above product

Algorithm for Modular Exponentiation

ALGORITHM 5 Modular Exponentiation.

procedure *modular exponentiation*(b: integer, $n = (a_{k-1}a_{k-2}\cdots a_1a_0)_2$, m: positive integers)
$x := 1$
$power := b$ **mod** m
for $i := 0$ **to** $k - 1$
begin
  if $a_i = 1$ **then** $x := (x \cdot power)$ **mod** m
  $power := (power \cdot power)$ **mod** m
end
{x equals b^n **mod** m}

if $a \equiv b \pmod m$ and $c \equiv d \pmod m$, then:
$a+c \equiv b+d \pmod m$, and
$ac \equiv bd \pmod m$

Example for Modular Exponentiation

- **2^{644} mod 645; $2^{644} = 2^4 \times 2^{128} \times 2^{512}$**

0: Because $a_0 = 0$, we have $x = 1$ and $power = 2^2 = 4$ **mod** 645 $= 4$;

1: Because $a_1 = 0$, we have $x = 1$ and $power = 4^2 = 16$ **mod** 645 $= 16$;

2: Because $a_2 = 1$, we have $x = 1 \cdot 16$ **mod** 645 $= 16$ and $power = 16^2 = 256$ **mod** 645 $= 256$;

3: Because $a_3 = 0$, we have $x = 16$ and $power = 256^2 = 65{,}536$ **mod** 645 $= 391$;

4: Because $a_4 = 0$, we have $x = 16$ and $power = 391^2 = 152{,}881$ **mod** 645 $= 16$;

5: Because $a_5 = 0$, we have $x = 16$ and $power = 16^2 = 256$ **mod** 645 $= 256$;

6: Because $a_6 = 0$, we have $x = 16$ and $power = 256^2 = 65{,}536$ **mod** 645 $= 391$;

7: Because $a_7 = 1$, we find that $x = (16 \cdot 391)$ **mod** 645 $= 451$ and $power = 391^2 = 152{,}881$ **mod** 645 $= 16$;

8: Because $a_8 = 0$, we have $x = 451$ and $power = 16^2 = 256$ **mod** 645 $= 256$;

9: Because $a_9 = 1$, we find that $x = (451 \cdot 256)$ **mod** 645 $= 1$.

if $a \equiv b \pmod m$ and $c \equiv d \pmod m$, then:
$a+c \equiv b+d \pmod m$, and
$ac \equiv bd \pmod m$

procedure *modular exponentiation*(b: integer, $n = (a_{k-1}a_{k-2}\cdots a_1a_0)_2$, m: positive integers)
$x := 1$
$power := b$ **mod** m
for $i := 0$ **to** $k - 1$
begin
  if $a_i = 1$ **then** $x := (x \cdot power)$ **mod** m
  $power := (power \cdot power)$ **mod** m
end
{x equals b^n **mod** m}

Euclid's Algorithm for GCD

- **Finding GCDs by comparing prime factorizations is time-consuming**
- **Euclid discovered: For all integers *a*, *b*,**
  gcd(*a*, *b*) = gcd((*a* mod *b*), *b*)
- **Sort *a*, *b* so that *a*>*b*, and then (given *b*>1)**
  (*a* mod *b*) < *b*, so problem is simplified

Proof of Euclid's Algorithm

- **If *d* | *m* and *d* | *n*, then *d* | (*m*+*n*) and *d* | (*m*-*n*)**
  - Already known
- **Show that gcd(*m*, *n*) = gcd(*m*-*n*, *n*)**
  1. gcd(*m*, *n*) can divide *m*-*n* and *n*

  gcd(*m*, *n*) ≤ gcd(*m*-*n*, *n*)
  2. gcd(*m*-*n*, *n*) can divide *m* and *n*

  gcd(*m*-*n*, *n*) ≤ gcd(*m*, *n*)
  - from 1. & 2., gcd(*m*, *n*) = gcd(*m*-*n*, *n*)

- **gcd(*m*, *n*) = gcd(*n*, *m* mod *n*)**

Euclid's Algorithm Example

- **gcd(372, 164) = gcd(372 mod 164, 164)**
  - 372 mod 164 = 44
- **gcd(164, 44) = gcd(164 mod 44, 44)**
  - 164 mod 44 = 32
- **gcd(44, 32) = gcd(44 mod 32, 32) = gcd(12, 32)**
  = gcd(32 mod 12, 12) = gcd(8, 12)
  = gcd(12 mod 8, 8) = gcd(4, 8) = gcd(8 mod 4, 4) = gcd(0, 4) = 4

Euclid's Algorithm Pseudocode


```
procedure gcd(a, b: positive integers)
        while b ≠ 0
                r := a mod b; a := b; b := r
        return a
```

- **Sorting inputs is not needed because order will be reversed each iteration**
- **Fast! Number of while loop iterations turns out to be O(log(max(*a*, *b*)))**