Chapter 3:
The Fundamentals: Algorithms, the Integers, and Matrices

§3.1: Algorithms

Algorithms

- The foundation of computer programming.
- Most generally, an *algorithm* just means a definite procedure for performing some sort of task.
- A computer *program* is simply a description of an algorithm in a language precise enough for a computer to understand, requiring only operations the computer already knows how to do.
- We say that a program *implements* (or "is an implementation of") its algorithm.

Programming Languages

- Some common programming languages:
  - **Newer:** Java, C, C++, Visual Basic, JavaScript, Perl, Tcl, Pascal
  - **Older:** Fortran, Cobol, Lisp, Basic
  - Assembly languages, for low-level coding.
- In this class we will use an informal, Pascal-like "*pseudo-code*" language.
- You should know at least 1 real language!

Algorithm Example (English)

- Task: Given a sequence $\{a_i\}=a_1,\ldots,a_n$, $a_i \in \mathbf{N}$, say what its largest element is.
- Set the value of a *temporary variable* v (largest element seen so far) to a_1's value.
- Look at the next element a_i in the sequence.
- If $a_i > v$, then re-assign v to the number a_i.
- Repeat previous 2 steps until there are no more elements in the sequence, & return v.

Executing an Algorithm

- When you start up a piece of software, we say the program or its algorithm are being *run* or *executed* by the computer.
- Given a description of an algorithm, you can also execute it by hand, by working through all of its steps on paper.
- Before ~WWII, “computer” meant a *person* whose job was to run algorithms!

Executing the Max algorithm

- Let $\{a_i\}$=7,12,3,15,8.   Find its maximum…
- Set $v = a_1 = 7$.
- Look at next element: $a_2 = 12$.
- Is $a_2>v$? Yes, so change v to 12.
- Look at next element: $a_2 = 3$.
- Is 3>12? No, leave v alone….
- Is 15>12? Yes, v=15…

Algorithm Characteristics

Some important features of algorithms:

- *Input*. Information or data that comes in.
- *Output*. Information or data that goes out.
- *Definiteness*. Precisely defined.
- *Correctness*. Outputs correctly relate to inputs.
- *Finiteness*. Won’t take forever to describe or run.
- *Effectiveness*. Individual steps are all do-able.
- *Generality*. Works for many possible inputs.
- *Efficiency*. Takes little time & memory to run.

Our Pseudocode Language: §A2

Declaration

procedure
name(*argument*: *type*)

variable := *expression*

informal statement

begin *statements* **end**

{*comment*}

if *condition* then
statement [else
statement]

STATEMENTS

for *variable* := *initial value* to *final value*
statement

while *condition*
statement

procname(*arguments*)

Not defined in book:

return *expression*

procedure *procname*(*arg*: *type*)

- Declares that the following text defines a procedure named *procname* that takes inputs (*arguments*) named *arg* which are data objects of the type *type*.
  - Example:
    procedure *maximum*(*L*: list of integers)
    [statements defining *maximum*…]

variable := *expression*

- An *assignment* statement evaluates the expression *expression*, then reassigns the variable *variable* to the value that results.
  - Example:
    v := 3x+7 (If x is 2, changes v to 13.)
- In pseudocode (but not real code), the *expression* might be informal:
  - x := the largest integer in the list L

Informal statement

- Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: "swap *x* and *y*"
- Keep in mind that real programming languages never allow this.
- When we ask for an algorithm to do so-and-so, writing "Do so-and-so" isn't enough!
  - Break down algorithm into detailed steps.

begin *statements* **end**

- Groups a sequence of statements together:

```
begin
    statement 1
    statement 2
    …
    statement n
end
```

- Allows sequence to be used like a single statement.
- Might be used:
  - After a **procedure** declaration.
  - In an **if** statement after **then** or **else**.
  - In the body of a **for** or **while** loop.

{*comment*}

- Not executed (does nothing).
- Natural-language text explaining some aspect of the procedure to human readers.
- Also called a *remark* in some real programming languages.
- Example:
  - {Note that v is the largest integer seen so far.}

if *condition* **then** *statement*

- Evaluate the propositional expression *condition*.
- If the resulting truth value is **true**, then execute the statement *statement*; otherwise, just skip on ahead to the next statement.
- Variant: **if** *cond* **then** *stmt1* **else** *stmt2*
  Like before, but iff truth value is **false**, executes *stmt2*.

while *condition* *statement*

- *Evaluate* the propositional expression *condition*.
- If the resulting value is **true**, then execute *statement*.
- Continue repeating the above two actions over and over until finally the *condition* evaluates to **false**; then go on to the next statement.

while *condition* *statement*

- Also equivalent to infinite nested **if**s, like so:

```
if condition
    begin
        statement
        if condition
            begin
                statement
                …(continue infinite nested if’s)
            end
    end
```

for *var* := *initial* to *final* *stmt*

- *Initial* is an integer expression.
- *Final* is another integer expression.
- Repeatedly execute *stmt*, first with variable *var* := *initial*, then with *var* := *initial*+1, then with *var* := *initial*+2, *etc*., then finally with *var* := *final*.
- What happens if *stmt* changes the value that *initial* or *final* evaluates to?

for *var* := *initial* to *final* *stmt*

- **For** can be exactly defined in terms of **while**, like so:

```
begin
    var := initial
    while var ≤ final
        begin
            stmt
            var := var + 1
        end
end
```

procedure(*argument*)

- A *procedure call* statement invokes the named *procedure*, giving it as its input the value of the *argument* expression.
- Various real programming languages refer to procedures as *functions* (since the procedure call notation works similarly to function application $f(x)$), or as *subroutines*, *subprograms*, or *methods*.

Max procedure in pseudocode

```
**procedure** $max(a_1, a_2, \ldots, a_n$: integers)
    $v := a_1$    {largest element so far}
    **for** $i := 2$ **to** $n$    {go thru rest of elems}
        **if** $a_i > v$ **then** $v := a_i$    {found bigger?}
    {at this point $v$'s value is the same as
     the largest integer in the list}
    **return** $v$
```

Another example task

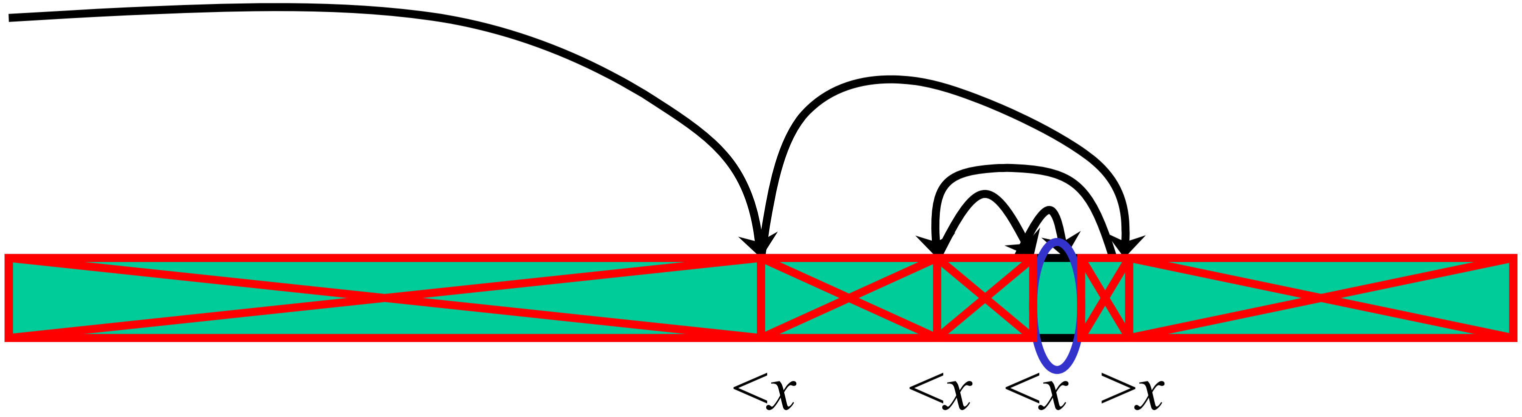
- Problem of *searching an ordered list*.
  - Given a list L of n elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element x,
  - Determine whether x appears in the list,
  - and if so, return its index (position) in the list.
- Problem occurs often in many contexts.
- Let's find an *efficient* algorithm!

Search alg. #1: Linear Search

```
procedure linear search
    ($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
    $i := 1$
    while ($i \leq n \wedge x \neq a_i$)
        $i := i + 1$
    if $i \leq n$ then location $:= i$
    else location $:= 0$
    return location {index or 0 if not found}
```

Search alg. #2: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Search alg. #2: Binary Search

```
procedure binary search
    ($x$:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
    $i := 1$   {left endpoint of search interval}
    $j := n$   {right endpoint of search interval}
    while $i<j$ begin    {while interval has >1 item}
        $m := \lfloor (i+j)/2 \rfloor$   {midpoint}
        if $x>a_m$ then $i := m+1$ else $j := m$
    end
    if $x = a_i$ then location $:= i$ else location $:= 0$
    return location
```

Sorting alg. : Bubble Sort

```
procedure bubblesort($a_1, a_2, \ldots, a_n$)

    for $i$ := 1 to $n$-1
        for $j$ := 1 to $n$-$i$
            if $a_j > a_{j+1}$ then interchange $a_j$ and $a_{j+1}$

    {$a_1, a_2, \ldots, a_n$ is in increasing order}
```

Sorting alg. : Insertion Sort

```
procedure insertionsort($a_1, a_2, \ldots, a_n$)
  for $j$ := 2 to $n$
  begin
      $i$ := 1
      while $a_j > a_i$
            $i$ := $i$+1
      $m$ := $a_j$
      for $k$ := 0 to $j$-$i$-1
            $a_{j-k}$ := $a_{j-k-1}$
      $a_i$ := $m$
  end    {$a_1, a_2, \ldots, a_n$ are sorted}
```

Greedy Change-Making Alg.

```
procedure change($c_1, c_2, \ldots, c_r : c_1 > c_2 > \ldots > c_r$ ; $n$)
  for $i$ := 1 to $r$
      while $n \geq c_i$
        begin
            add a coin with value $c_i$ to the change
            $n$ := $n - c_i$
        end
```

Practice exercises

- 3.1.3: Devise an algorithm that finds the sum of all the integers in a list. [2 min]
- **procedure** *sum*($a_1, a_2, \ldots, a_n$: integers)

```
    $s := 0$    {sum of elems so far}
    for $i := 1$ to $n$    {go thru all elems}
        $s := s + a_i$    {add current item}
    {at this point $s$ is the sum of all items}
    return $s$
```

§3.2: The Growth of Functions

Orders of Growth

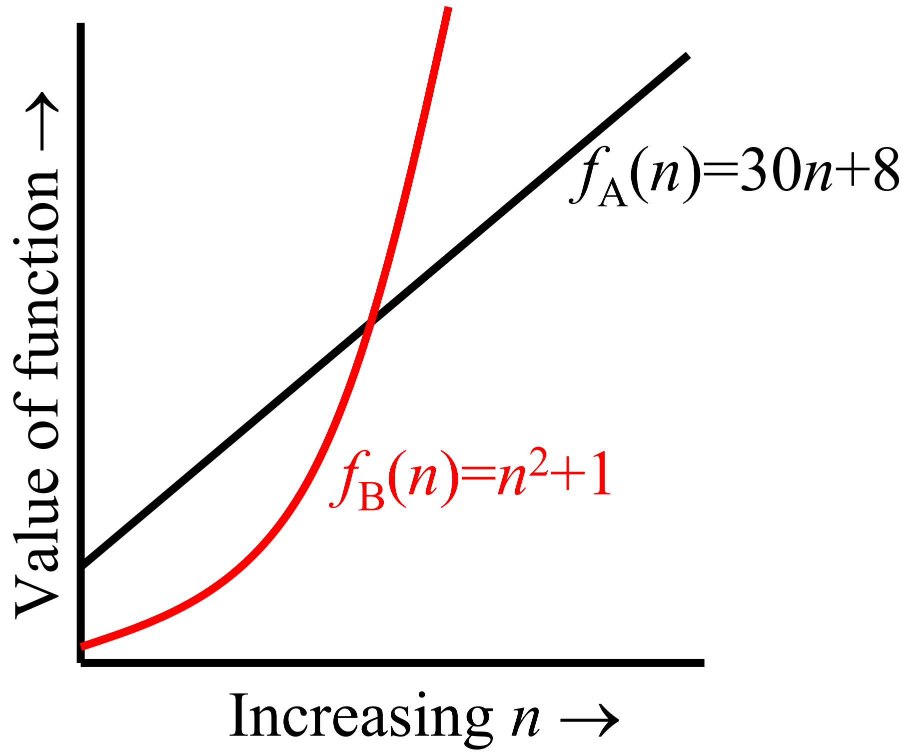
- For functions over numbers, we often need to know a rough measure of *how fast a function grows*.
- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ *in the limit* (for large enough values of x).
- Useful in engineering for showing that one design *scales* better or worse than another.

Orders of Growth - Motivation

- 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?

Visualizing Orders of Growth

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



Concept of order of growth

- We say $f_A(n)=30n+8$ is *order n*, or $O(n)$. It is, at most, 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.
- 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 g be any function $\mathbf{R}\rightarrow\mathbf{R}$.

- Define "*at most order g*", written O(g), to be: $\{f:\mathbf{R}\rightarrow\mathbf{R} \mid \exists c,k: \forall x>k: f(x) \le cg(x)\}$.
  - "Beyond some point k, function f is at most a constant c times g (*i.e.*, proportional to g)."
- "f is *at most order g*", or "f is O(g)", or "f=O(g)" all just mean that $f\in$O(g).
- Sometimes the phrase "at most" is omitted.

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!)

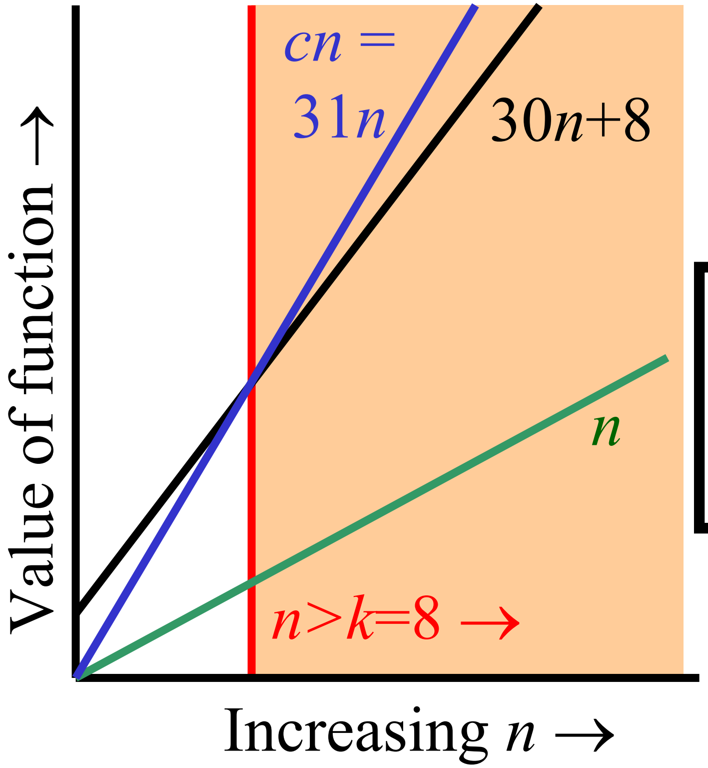
However, you should **prove** that the values you choose do work.

“Big-O” Proof Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, k: \forall n>k: 30n+8 \leq cn$.
- Show that n^2+1 is $O(n^2)$.
  - Show $\exists c, k: \forall n>k: n^2+1 \leq cn^2$.

Big-O example, graphically

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



$30n+8 \in O(n)$

Useful Facts about Big O

- Big O, as a relation, is transitive:
  $f\in O(g) \wedge g\in O(h) \rightarrow f\in O(h)$
- O with constant multiples, roots, and logs...
  $\forall f$ (in $\omega(1)$) & constants $a,b\in\mathbf{R}$, with $b\geq 0$, af, f^{1-b}, and $(\log_b f)^a$ are all $O(f)$.
- Sums of functions:
  If $g\in O(f)$ and $h\in O(f)$, then $g+h\in O(f)$.

More Big-O facts

- $\forall c>0$, $O(cf)=O(f+c)=O(f-c)=O(f)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \rightarrow$
  - $f_1 f_2 \in O(g_1 g_2)$
  - $f_1+f_2 \in O(g_1+g_2)$

$= O(\max(g_1,g_2))$

$= O(g_1)$ if $g_2 \in O(g_1)$ (Very useful!)

Orders of Growth (§3.2) - So Far

- For any g:**R**→**R**, "*at most order g*",
  $O(g) \equiv \{f:\mathbf{R}\rightarrow\mathbf{R} \mid \exists c,k\ \forall x>k\ |f(x)| \leq |cg(x)|\}$.
  - Often, one deals only with positive functions and can ignore absolute value symbols.
- "$f\in O(g)$" often written "f is $O(g)$" or "$f=O(g)$".
  - The latter form is an instance of a more general convention...

Order-of-Growth Expressions

- “O(f)” when used as a term in an arithmetic expression means: “some function f such that $f \in O(f)$”.
- *E.g.*: “$x^2+O(x)$” means “x^2 plus some function that is $O(x)$”.
- Formally, you can think of any such expression as denoting a set of functions: “$x^2+O(x)$” $:\equiv \{g \mid \exists f \in O(x): g(x)= x^2+f(x)\}$

Order of Growth Equations

- Suppose E_1 and E_2 are order-of-growth expressions corresponding to the sets of functions S and T, respectively.
- Then the "equation" $E_1=E_2$ really means

$$\forall f\in S, \exists g\in T: \quad f=g$$

or simply $S\subseteq T$.

- Example: $x^2 + \mathrm{O}(x) = \mathrm{O}(x^2)$ means

$$\forall f\in\mathrm{O}(x): \exists g\in\mathrm{O}(x^2): \ x^2+f(x)=g(x)$$

Useful Facts about Big O

- $\forall\ f,g$ & constants $a,b\in\mathbf{R}$, with $b\geq 0$,
  - $af = O(f)$; (e.g. $3x^2 = O(x^2)$)
  - $f+O(f) = O(f)$; (e.g. $x^2+x = O(x^2)$)
- Also, if $f=\Omega(x)$ (*at least* order 1), then:
  - $|f|^{1-b} = O(f)$; (e.g. $x^{-1} = O(x)$)
  - $(\log_b |f|)^a = O(f)$. (e.g. $\log x = O(x)$)
  - $g=O(fg)$ (e.g. $x = O(x \log x)$)
  - $fg \neq O(g)$ (e.g. $x \log x \neq O(x)$)
  - $a=O(f)$ (e.g. $3 = O(x)$)

Definition: $\Theta(g)$, *exactly order g*

- If $f \in O(g)$ and $g \in O(f)$ then we say "*g and f are of the same order*" or "*f is (exactly) order g*" and write $f \in \Theta(g)$.
- Another equivalent definition:

  $\Theta(g) \equiv \{f\!:\!\mathbf{R}\rightarrow\mathbf{R} \mid$

  $\exists c_1 c_2 k\ \forall x>k\!: |c_1 g(x)| \leq |f(x)| \leq |c_2 g(x)|\ \}$
- "Everywhere beyond some point k, $f(x)$ lies in between two multiples of $g(x)$."

Rules for Θ

- Mostly like rules for O( ), except:
- $\forall\ f,g>0$ & constants $a,b \in \mathbf{R}$, with $b>0$,
  $af \in \Theta(f)$, but ← Same as with O.
  $f \notin \Theta(fg)$ unless $g=\Theta(1)$ ← Unlike O.
  $|f|^{1-b} \notin \Theta(f)$, and ← Unlike with O.
  $(\log_b |f|)^c \notin \Theta(f)$. ← Unlike with O.
- The functions in the latter two cases we say are *strictly of lower order* than $\Theta(f)$.

Θ example

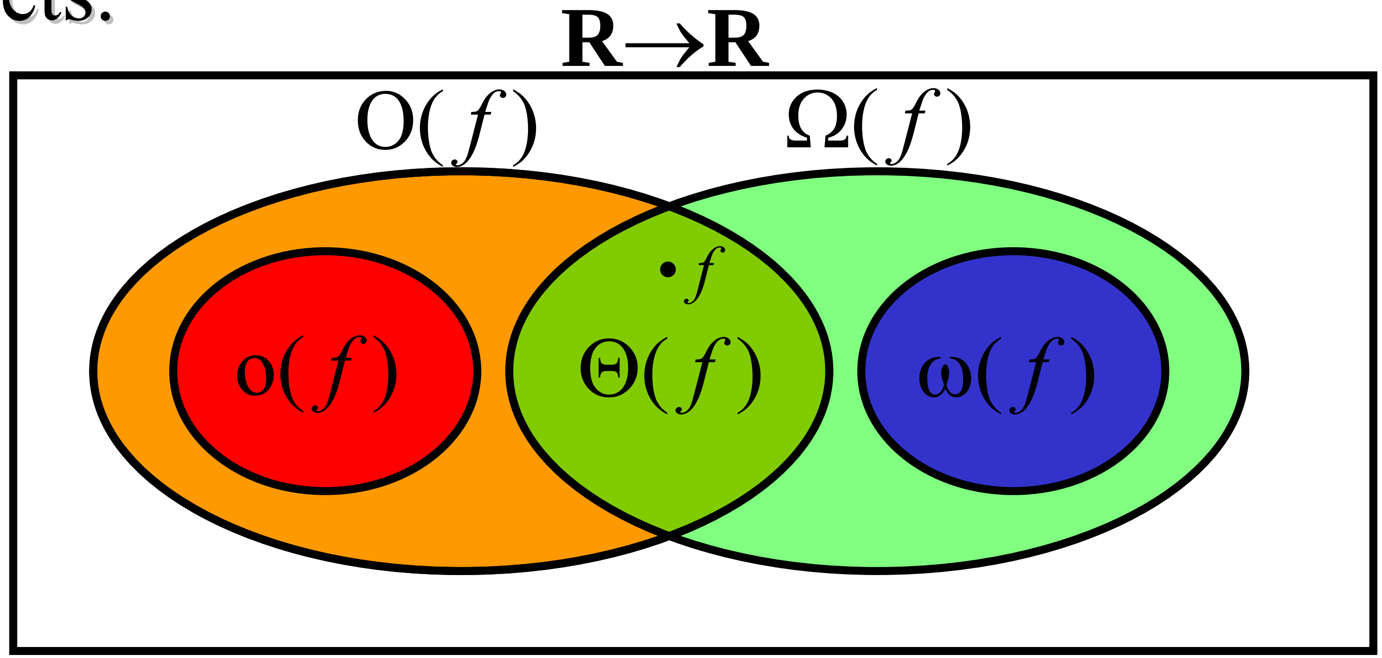
- Determine whether: $\left(\sum_{i=1}^{n} i\right) \overset{?}{\in} \Theta(n^2)$
- Solution:

Other Order-of-Growth Relations

- $\Omega(g) = \{f \mid g \in O(f)\}$
  "The functions that are *at least order g*."
- $o(g) = \{f \mid \forall c>0\ \exists k\ \forall x>k : |f(x)| < |cg(x)|\}$
  "The functions that are *strictly lower order than g*." $o(g) \subset O(g) - \Theta(g)$.
- $\omega(g) = \{f \mid \forall c>0\ \exists k\ \forall x>k : |cg(x)| < |f(x)|\}$
  "The functions that are *strictly higher order than g*." $\omega(g) \subset \Omega(g) - \Theta(g)$.

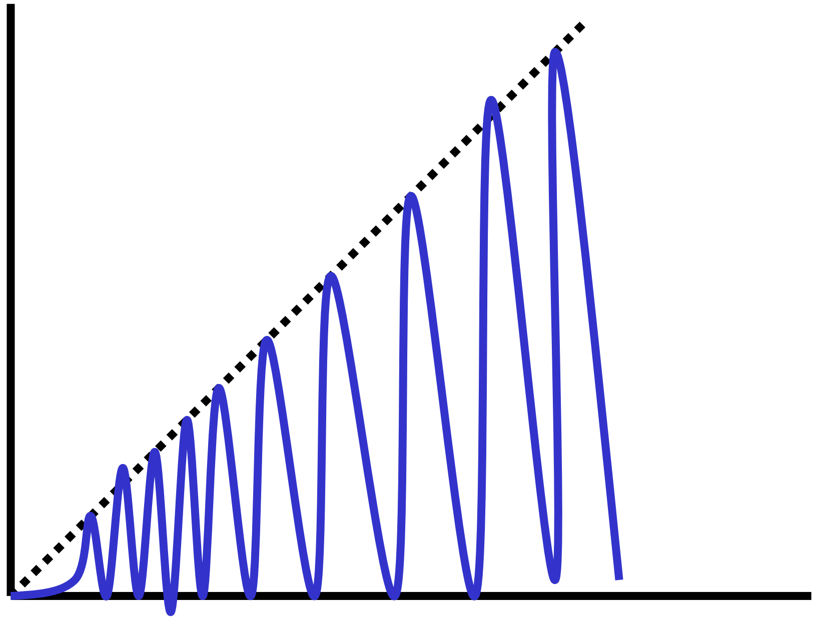
Relations Between the Relations

- Subset relations between order-of-growth sets.



Why o(f)⊂O(x)−Θ(x)

- A function that is O(x), but neither o(x) nor Θ(x):



Strict Ordering of Functions

- Temporarily let's write $f \prec g$ to mean $f \in o(g)$,
  $f \sim g$ to mean $f \in \Theta(g)$
- Note that $$f \prec g \Leftrightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$
- Let $k>1$. Then the following are true:
  $1 \prec \log \log n \prec \log n \sim \log_k n \prec \log^k n$
  $\prec n^{1/k} \prec n \prec n \log n \prec n^k \prec k^n \prec n! \prec n^n$ …

Review: Growth of Functions (§3.2)

Definitions of order-of-growth sets, $\forall g:\mathbf{R}\to\mathbf{R}$

- $O(g) \equiv \{f \mid \exists\, c>0\ \exists k\ \forall x>k\ |f(x)| < |cg(x)|\}$
- $o(g) \equiv \{f \mid \forall c>0\ \exists k\ \forall x>k\ |f(x)| < |cg(x)|\}$
- $\Omega(g) \equiv \{f \mid g\in O(f)\}$
- $\omega(g) \equiv \{f \mid g\in o(f)\}$
- $\Theta(g) \equiv O(g) \cap \Omega(g)$

§3.3: Complexity of Algorithms

What is *complexity*?

- The word *complexity* has a variety of technical meanings in different fields.
- There is a field of *complex systems*, which studies complicated, difficult-to-analyze *non-linear* and *chaotic* natural & artificial systems.
- Another concept: *Informational complexity*: the amount of *information* needed to completely describe an object. (An active research field.)
- We will study *algorithmic complexity*.

Algorithmic Complexity

- The *algorithmic complexity* of a computation is some measure of how *difficult* it is to perform the computation.
- Measures some aspect of *cost* of computation (in a general sense of cost).
- Common complexity measures:
  - "Time" complexity: # of ops or steps required
  - "Space" complexity: # of memory bits req'd

An aside...

- Another, increasingly important measure of complexity for computing is *energy complexity* - How much total energy is used in performing the computation.
- Motivations: Battery life, electricity cost...
- I develop *reversible* circuits & algorithms that recycle energy, trading off energy complexity for spacetime complexity.

Complexity Depends on Input

- Most algorithms have different complexities for inputs of different sizes. (*E*.*g*. searching a long list takes more time than searching a short one.)
- Therefore, complexity is usually expressed as a *function* of input length.
- This function usually gives the complexity for the *worst-case* input of any given length.

Complexity & Orders of Growth

- Suppose algorithm A has worst-case time complexity (w.c.t.c., or just *time*) $f(n)$ for inputs of length n, while algorithm B (for the same task) takes time $g(n)$.
- Suppose that $f\in\omega(g)$, also written $f \succ g$.
- Which algorithm will be *fastest* on all sufficiently-large, worst-case inputs?

Example 1: Max algorithm

- Problem: Find the *simplest form* of the *exact* order of growth (Θ) of the *worst-case* time complexity (w.c.t.c.) of the *max* algorithm, assuming that each line of code takes some constant time every time it is executed (with possibly different times for different lines of code).

Complexity analysis of *max*

```
procedure max($a_1, a_2, \ldots, a_n$: integers)
    $v := a_1$                              $t_1$
    for $i := 2$ to $n$                     $t_2$
        if $a_i > v$ then $v := a_i$        $t_3$
    return $v$                              $t_4$
```

Times for *each* execution of each line.

What's an expression for the *exact* total worst-case time? (Not its order of growth.)

Complexity analysis, cont.

```
**procedure** $max(a_1, a_2, \ldots, a_n$: integers)
    $v := a_1$                              $t_1$
    **for** $i := 2$ **to** $n$                   $t_2$
        **if** $a_i > v$ **then** $v := a_i$           $t_3$
    **return** $v$                            $t_4$
```

Times for *each* execution of each line.

w.c.t.c.:

$$t(n) = t_1 + \left(\sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

Complexity analysis, *cont.*

Now, what is the simplest form of the exact (Θ) order of growth of *t*(*n*)?

$$t(n) = t_1 + \left(\sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

$$= \Theta(1) + \left(\sum_{i=2}^{n} \Theta(1) \right) + \Theta(1) = \Theta(1) + (n-1)\Theta(1)$$

$$= \Theta(1) + \Theta(n)\Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$$

Example 2: Linear Search

```
procedure linear search ($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
    $i := 1$                                      $t_1$
    while ($i \leq n \wedge x \neq a_i$)          $t_2$
        $i := i + 1$                              $t_3$
    if $i \leq n$ then location $:= i$            $t_4$
    else location $:= 0$                          $t_5$
    return location                               $t_6$
```

Linear search analysis

- Worst case time complexity order:

$$t(n) = t_1 + \left(\sum_{i=1}^{n} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

- Best case:

$$t(n) = t_1 + t_2 + t_4 + t_6 = \Theta(1)$$

- Average case, if item is present:

$$t(n) = t_1 + \left(\sum_{i=1}^{n/2} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

Review §3.3: Complexity

- Algorithmic complexity = *cost* of computation.
- Focus on *time* complexity (space & energy are also important.)
- Characterize complexity as a function of input size: Worst-case, best-case, average-case.
- Use orders of growth notation to concisely summarize growth properties of complexity fns.

Example 3: Binary Search

```
procedure binary search ($x$:integer, $a_1, a_2, \ldots, a_n$:
        distinct integers)
    $i := 1$                                             } $\Theta(1)$
    $j := n$                                             }
    while $i<j$ begin
        $m := \lfloor (i+j)/2 \rfloor$                   } $\Theta(1)$
        if $x>a_m$ then $i := m+1$ else $j := m$          }
    end
    if $x = a_i$ then location $:= i$ else location $:= 0$   } $\Theta(1)$
    return location                                      }
```

Key question:

How many loop iterations?

Binary search analysis

- Suppose $n=2^k$.
- Original range from $i=1$ to $j=n$ contains n elems.
- Each iteration: Size $j-i+1$ of range is cut in half.
- Loop terminates when size of range is $1=2^0$ ($i=j$).
- Therefore, number of iterations is $k = \log_2 n$ $= \Theta(\log_2 n) = \Theta(\log n)$
- Even for $n \neq 2^k$ (not an integral power of 2), time complexity is still $\Theta(\log_2 n) = \Theta(\log n)$.

Names for some orders of growth

• $\Theta(1)$	Constant	
• $\Theta(\log_c n)$	Logarithmic (same order $\forall c$)	
• $\Theta(\log^c n)$	Polylogarithmic	(With c a constant.)
• $\Theta(n)$	Linear	
• $\Theta(n^c)$	Polynomial	
• $\Theta(c^n)$, $c>1$	Exponential	
• $\Theta(n!)$	Factorial	

Problem Complexity

- The complexity of a computational *problem* or *task* is (the order of growth of) the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
- *E.g.* the problem of searching an ordered list has *at most logarithmic* time complexity. (Complexity is O(log *n*).)

Tractable *vs.* intractable

- A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*). **P** is the set of all tractable problems.
- A problem or algorithm that has more than polynomial complexity is considered *intractable* (or *infeasible*).
- Note that $n^{1,000,000}$ is *technically* tractable, but really impossible. $n^{\log \log \log n}$ is *technically* intractable, but easy. Such cases are rare though.

Unsolvable problems

- Turing discovered in the 1930’s that there are problems unsolvable by *any* algorithm.
    - Or equivalently, there are undecidable yes/no questions, and uncomputable functions.
- Example: the *halting problem*.
    - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an “*infinite loop*?”

P *vs.* NP

- **NP** is the set of problems for which there exists a tractable algorithm for *checking solutions* to see if they are correct.
- We know **P⊆NP**, but the most famous unproven conjecture in computer science is that this inclusion is *proper* (*i.e.*, that **P⊂NP** rather than **P=NP**).
- Whoever first proves it will be famous!

Computer Time Examples

$\#ops(n)$	(1.25 bytes) $n=10$	(125 kB) $n=10^6$
$\log_2 n$	3.3 ns	19.9 ns
n	10 ns	1 ms
$n \log_2 n$	33 ns	19.9 ms
n^2	100 ns	16 m 40 s
2^n	1.024 μs	$10^{301,004.5}$ Gyr
$n!$	3.63 ms	Ouch!

Assume time = 1 ns (10^{-9} second) per op, problem size = n bits, #ops a function of n as shown.

Things to Know

- Definitions of algorithmic complexity, time complexity, worst-case complexity; names of orders of growth of complexity.
- How to analyze the worst case, best case, or average case order of growth of time complexity for simple algorithms.

§3.4: The Integers and Division

The Integers and Division

- 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).

Divides, Factor, Multiple

- Let $a,b \in \mathbf{Z}$ with $a \neq 0$.
- $a|b \equiv$ "*a divides b*" $:\equiv$ "$\exists c \in \mathbf{Z}: b=ac$"
  "There is an integer c such that c times a equals b."
  - Example: $3|-12 \Leftrightarrow$ **True**, but $3|7 \Leftrightarrow$ **False**.
- Iff 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" $:\equiv 2|b$. Is 0 even? Is -4?

Facts re: the Divides Relation

- $\forall a,b,c \in \mathbf{Z}$:
  1. $a|0$
  2. $(a|b \wedge a|c) \rightarrow a \mid (b+c)$
  3. $a|b \rightarrow a|bc$
  4. $(a|b \wedge b|c) \rightarrow a|c$
- **Proof** of (2): $a|b$ means there is an s such that $b=as$, and $a|c$ means that there is a t such that $c=at$, so $b+c = as+at = a(s+t)$, so $a|(b+c)$ also.■

More Detailed Version of Proof

- Show $\forall a,b,c \in \mathbf{Z}$: $(a|b \land a|c) \rightarrow a \mid (b + c)$.
- Let a, b, c be any integers such that $a|b$ and $a|c$, and show that $a \mid (b + c)$.
- By defn. 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 $a|(b+c)$.

The Division "Algorithm"

- Really just a *theorem*, not an algorithm…
  - The name is used here for historical reasons.
- For any integer *dividend* a and *divisor* $d \neq 0$, there is a unique integer *quotient* q and *remainder* $r \in \mathbf{N} \ni a = dq + r$ and $0 \leq r < |d|$.

(such that)

- $\forall a,d \in \mathbf{Z}, d>0: \exists! q,r \in \mathbf{Z}: 0 \leq r < |d|, a=dq+r$.
- We can find q and r by: $q = \lfloor a/d \rfloor$, $r = a - qd$.

The **mod** operator

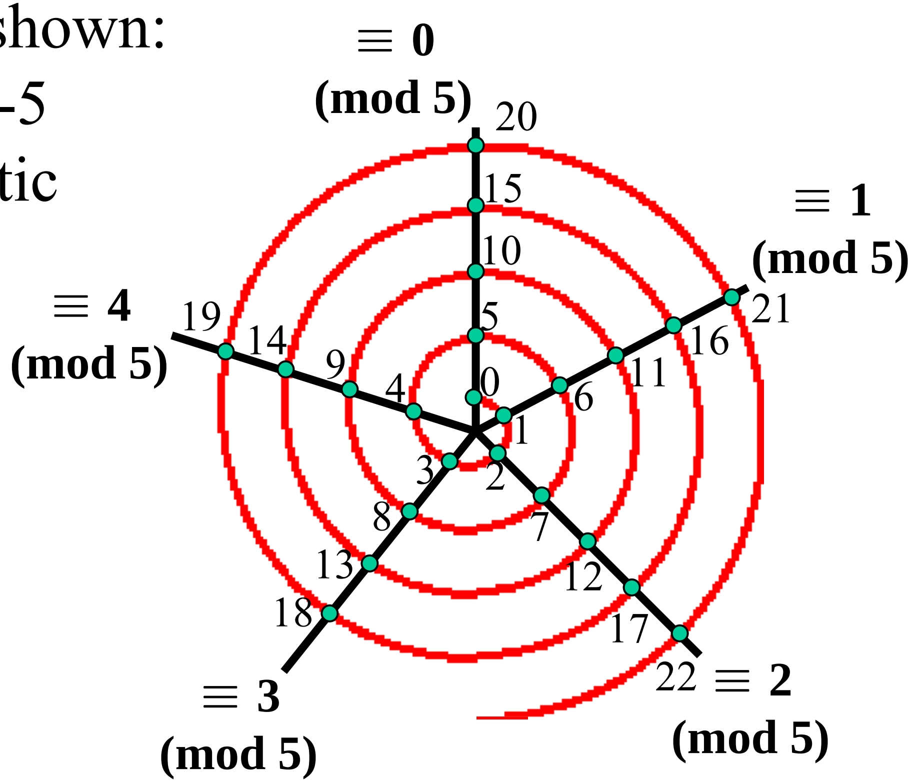
- An integer "division remainder" operator.
- Let $a,d \in \mathbf{Z}$ with $d>1$. Then a **mod** d denotes the remainder r from the division "algorithm" with dividend a and divisor d; *i.e.* the remainder when a is divided by d. (Using *e.g.* long division.)
- We can compute (a **mod** d) by: $a - d\cdot\lfloor a/d \rfloor$.
- In C programming language, "%" = mod.

Modular Congruence

- Let $\mathbf{Z}^+=\{n\in\mathbf{Z} \mid n>0\}$, the positive integers.
- Let $a,b\in\mathbf{Z}$, $m\in\mathbf{Z}^+$.
- Then *a is congruent to b modulo m*, written "$a\equiv b \pmod{m}$", iff $m \mid a-b$.
- Also equivalent to: $(a-b) \bmod m = 0$.
- (Note: this is a different use of "≡" than the meaning "is defined as" I've used before.)

Spiral Visualization of **mod**

Example shown:
modulo-5
arithmetic



Useful Congruence Theorems

- Let $a,b\in\mathbf{Z}$, $m\in\mathbf{Z}^+$. Then:

  $$a\equiv b \pmod{m} \Leftrightarrow \exists k\in\mathbf{Z}\ a=b+km.$$

- Let $a,b,c,d\in\mathbf{Z}$, $m\in\mathbf{Z}^+$. Then if $a\equiv b \pmod{m}$ and $c\equiv d \pmod{m}$, then:
  - $a+c \equiv b+d \pmod{m}$, and
  - $ac \equiv bd \pmod{m}$
- Hash Function： $h(k)=k \mathbf{\ mod\ } m$

§3.5: Primes and Greatest Common Divisors

Prime Numbers

- An integer $p>1$ is *prime* iff it is not the product of any two integers greater than 1:

  $$p>1 \wedge \neg\exists a,b\in\mathbf{N}\text{:}\ a>1,\ b>1,\ ab=p.$$
- The only positive factors of a prime p are 1 and p itself. 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

Its "Prime Factorization"

- Every positive integer has a unique representation as the product of a non-decreasing series of zero or more primes.
  - 1 = (product of empty series) = 1
  - $2 = 2$ (product of series with one element 2)
  - $4 = 2\cdot 2$ (product of series 2,2)
  - $2000 = 2\cdot 2\cdot 2\cdot 2\cdot 5\cdot 5\cdot 5$; $2001 = 3\cdot 23\cdot 29$;
    $2002 = 2\cdot 7\cdot 11\cdot 13$; $2003 = 2003$

An Application of Primes

- When you visit a secure web site (https:… address, indicated by padlock icon in IE, key icon in Netscape), 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.

Greatest Common Divisor

- 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.

  $d = \gcd(a,b) = \max(d\text{: } d|a \wedge d|b) \Leftrightarrow$
  $d|a \wedge d|b \wedge \forall e \in \mathbf{Z}, (e|a \wedge e|b) \rightarrow d \geq e$

- Example: gcd(24,36)=?
  Positive common divisors: 1,2,3,4,6,12…
  Greatest is 12.

GCD shortcut

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

- Example:
  - 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) =

Relative Primality

- Integers *a* and *b* are called *relatively prime* or *coprime* iff their gcd = 1.
  - Example: Neither 21 and 10 are prime, but they are *coprime*. 21=3·7 and 10=2·5, so they have no common factors > 1, so their gcd = 1.
- A *set* of integers $\{a_1,a_2,\ldots\}$ is *(pairwise) relatively prime* if all pairs a_i, a_j, $i\neq j$, are relatively prime.

Least Common Multiple

- lcm(a,b) of positive integers a, b, is the smallest positive integer that is a multiple both of a and of b. *E.g.* lcm(6,10)=30

  m = lcm(a,b) = min(m: $a|m \land b|m$) $\Leftrightarrow$
  $a|m \land b|m \land \forall n \in \mathbf{Z}$: $(a|n \land b|n) \rightarrow (m \leq n)$

- 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 LCM is given by

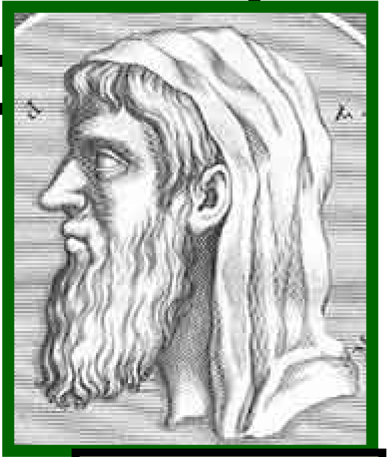
  $$\text{lcm}(a,b) = p_1^{\max(a_1,b_1)} p_2^{\max(a_2,b_2)} \ldots p_n^{\max(a_n,b_n)}.$$

§3.6: Integers and Algorithms

Integers & Algorithms

- Topics:
  - – Euclidean algorithm for finding GCD's.
  - – Base-*b* representations of integers.
    - Especially: binary, hexadecimal, octal.
    - Also: Two's complement representation of negative numbers.
  - – Algorithms for computer arithmetic:
    - Binary addition, multiplication, division.

Euclid’s Algorithm for GCD



Euclid of Alexandria
325-265 B.C.

- Finding GCDs by comparing prime factorizations can be difficult if the prime factors are unknown.
- Euclid discovered: For all integers a, b,
  $\gcd(a, b) = \gcd((a \bmod b), b)$.
- Sort a,b so that $a>b$, and then (given $b>1$) (a mod b) < a, so problem is simplified.

Euclid’s Algorithm Example

- gcd(372,164) = gcd(372 mod 164, 164).
  - 372 mod 164 = $372-164\lfloor 372/164\rfloor = 372-164\cdot 2 = 372-328 = 44$.
- gcd(164,44) = gcd(164 mod 44, 44).
  - 164 mod 44 = $164-44\lfloor 164/44\rfloor = 164-44\cdot 3 = 164-132 = 32$.
- 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 $b \neq 0$
        $r := a$ mod $b$;   $a := b$;   $b := r$
    return $a$
```

Sorting inputs not needed b/c order will be reversed each iteration.

Fast! Number of while loop iterations turns out to be O(log(max(a,b))).

Base-*b* number systems

- Ordinarily we write *base*-10 representations of numbers (using digits 0-9).
- 10 isn't special; any base $b>1$ will work.
- For any positive integers n,b there is a unique sequence $\underbrace{a_k\, a_{k-1} \ldots\, a_1 a_0}$ of *digits* $a_i<b$ such that

$$n=\sum_{i=0}^{k} a_i b^i$$

The "*base b expansion of n*"

See module #12 for summation notation.

Particular Bases of Interest

- Base *b*=10 (decimal):
  10 digits: 0,1,2,3,4,5,6,7,8,9.

Used only because we have 10 fingers

- Base *b*=2 (binary):
  2 digits: 0,1. ("Bits"="binary digits.")

Used internally in all modern computers

- Base *b*=8 (octal):
  8 digits: 0,1,2,3,4,5,6,7.

Octal digits correspond to groups of 3 bits

- Base *b*=16 (hexadecimal):
  16 digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Hex digits give groups of 4 bits

Converting to Base *b*

(Algorithm, informally stated)

- To convert any integer *n* to any base $b>1$:
- To find the value of the *rightmost* (lowest-order) digit, simply compute *n* mod *b*.
- Now replace *n* with the quotient $\lfloor n/b \rfloor$.
- Repeat above two steps to find subsequent digits, until *n* is gone (=0).

Exercise for student: Write this out in pseudocode…

Addition of Binary Numbers

```
procedure add($a_{n-1}\ldots a_0$, $b_{n-1}\ldots b_0$: binary
    representations of non-negative integers $a$,$b$)
    $carry$ := 0
    for $bitIndex$ := 0 to $n-1$                {go through bits}
        $bitSum$ := $a_{bitIndex}+b_{bitIndex}+carry$   {2-bit sum}
        $s_{bitIndex}$ := $bitSum$ mod 2          {low bit of sum}
        $carry$ := $\lfloor bitSum / 2 \rfloor$             {high bit of sum}
    $s_n$ := $carry$
    return $s_n\ldots s_0$: binary representation of integer $s$
```

Two’s Complement

- In binary, negative numbers can be conveniently represented using *two’s complement notation*.
- In this scheme, a string of *n* bits can represent any integer *i* such that $-2^{n-1} \leq i < 2^{n-1}$.
- The bit in the highest-order bit-position ($n-1$) represents a coefficient multiplying -2^{n-1};
  - The other positions $i < n-1$ just represent 2^i, as before.
- The negation of any *n*-bit two’s complement number $a = a_{n-1}\ldots a_0$ is given by $\overline{a_{n-1}\ldots a_0} + 1$.

The bitwise logical complement of the *n*-bit string $a_{n-1}\ldots a_0$.

Correctness of Negation Algorithm

- **Theorem:** For an integer *a* represented in two's complement notation, $-a = \overline{a} + 1$.
- **Proof:**

Subtraction of Binary Numbers

procedure *subtract*($a_{n-1}\ldots a_0$, $b_{n-1}\ldots b_0$: binary two's complement representations of integers a,b)

return *add*(a, *add*($\overline{b}$,1)) { $a + (-b)$ }

This fails if either of the adds causes a carry into or out of the $n-1$ position, since $2^{n-2}+2^{n-2} \neq -2^{n-1}$, and $-2^{n-1} + (-2^{n-1}) = -2^n$ isn't representable!

Multiplication of Binary Numbers

```
procedure multiply($a_{n-1}\ldots a_0$, $b_{n-1}\ldots b_0$:
    binary representations of $a,b\in\mathbf{N}$)
    $product$ := 0
    for $i$ := 0 to $n-1$
        if $b_i = 1$ then
            $product$ := add($a_{n-1}\ldots a_0 0^i$, $product$)
    return $product$
```

i extra 0-bits appended after the digits of *a*

Binary Division with Remainder

```
procedure div-mod($a,d \in \mathbf{Z}^+$)  {Quotient & rem. of $a/d$.}
    $n$ := max(length of $a$ in bits, length of $d$ in bits)
    for $i := n-1$ downto 0
        if $a \geq d0^i$ then      {Can we subtract at this position?}
            $q_i := 1$             {This bit of quotient is 1.}
            $a := a - d0^i$        {Subtract to get remainder.}
        else
            $q_i := 0$             {This bit of quotient is 0.}
    $r := a$
    return $q,r$           {$q$ = quotient, $r$ = remainder}
```

§3.7: Applications of Number Theory

Some Useful Results

Theorem 1: If a and b are positive integers, then there exist integers s and t such that $\gcd(a, b)=sa+tb$.

Example: Express gcd(252, 198) as a linear combination of 252 and 198.

Some Useful Results

Lemma 1: If a, b and c are positive integers such that gcd(a, b)=1 and $a|bc$, then $a|c$.

Pf: by Theorem 1, $1=sa+tb$, $\Rightarrow c=sac+tbc$

Lemma 2: If p is a prime and $p|a_1a_2\ldots a_n$, where each a_i is an integer, then $p|a_i$ for some i.

Some Useful Results

Theorem 2: Let m be a positive integer and let a, b and c be integers. If $ac \equiv bc \pmod{m}$ and $\gcd(c, m)=1$, then $a \equiv b \pmod{m}$.

Pf:

Example: $14 \equiv 8 \pmod{6}$, then $7 \xcancel{\equiv} 4 \pmod{6}$?

Linear Congruences

Theorem 3: If a and m are relatively prime integers and $m > 1$, then an inverse of a modulo m exists. Furthermore, this inverse is unique modulo m.

Pf:

Linear Congruences

Example: Find an inverse of 3 modulo 7.

Example: Solve $3x \equiv 4 \pmod{7}$.

Sol:

Extended Euclid’s Algorithm

```
EXTENDED EUCLID($m$, $b$)
{  ($A_1$, $A_2$, $A_3$)=(1, 0, $m$);     ($B_1$, $B_2$, $B_3$)=(0, 1, $b$) ;
   while ( ($B_3$!=0) && ($B_3$!=1))
           {  Q = $A_3$ div $B_3$ ;
              ($T_1$, $T_2$, $T_3$)=($A_1$–Q*$B_1$, $A_2$–Q*$B_2$, $A_3$–Q*$B_3$) ;
              ($A_1$, $A_2$, $A_3$)=($B_1$, $B_2$, $B_3$) ;
              ($B_1$, $B_2$, $B_3$)=($T_1$, $T_2$, $T_3$) ; }
    if ($B_3$ = 0)    return  gcd($m$, $b$) = $A_3$; no inverse ;
    if ($B_3$ = 1)    return  gcd($m$, $b$)=1;  $b^{-1}$ mod $m$ = $B_2$ ;
}
```

Extended Euclid's Algorithm

Example: **Find the inverse of 550 mod 1759.**

Q	A_1	A_2	A_3	B_1	B_2	B_3
—	1	0	1759	0	1	550
3	0	1	550	1	–3	109
5	1	–3	109	–5	16	5
21	–5	16	5	106	–339	4
1	106	–339	4	–111	355	1

Chinese Remainder Theorem

Theorem 4: Let $m_1, m_2, \ldots, m_n$ be pairwise relatively prime positive integers and $a_1, a_2, \ldots, a_n$ arbitrary integers. Then the system

$$x \equiv a_1 \pmod{m_1},$$
$$x \equiv a_2 \pmod{m_2},$$
$$\vdots$$
$$x \equiv a_n \pmod{m_n},$$

has a unique solution modulo $m = m_1 m_2 \ldots m_n$.

Fermat's Little Theorem

Theorem 5: If p is a prime positive integers and a is an integer not divisible by p, then

$$a^{p-1} \equiv 1 \,(\text{mod } p).$$

Furthermore, for every integer a we have

$$a^{p} \equiv a \,(\text{mod } p).$$

Example: The integer 341 is a *pseudoprime* to the base 2 because it is composite (11·31) and $2^{340} \equiv 1$ (mod 341).

Public Key Cryptography

RSA Cryptosystem: There are a private key and a public key. It is an exponentiation algorithm. (Also known as MIT algorithm)

RSA Encryption: $C = M^e \bmod n$.

RSA Decryption: $M = C^d \bmod n$.

where $n = pq$, p and q are two *large primes*, and

$ed \; mod \; \phi(n) = 1$ with $\phi(n) = (p\text{-}1)(q\text{-}1)$.

Public Key Cryptography

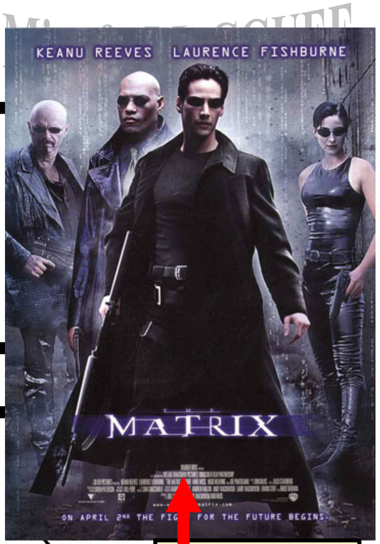
Example: Let $n = 15$, $e = 3$, encrypt $M = 7$.

Example: Encrypt the message STOP with $p = 43$, $q = 59$, with $e = 13$..

Sol:

§3.8: Matrices

Matrices



Not our meaning!

- A *matrix* (say MAY-trix) is a rectangular array of objects (usually numbers).
- An *m*×*n* ("*m* by *n*") matrix has exactly *m* horizontal rows, and *n* vertical columns.
- Plural of matrix = *matrices* (say MAY-trih-sees)
- An *n*×*n* matrix is called a *square* matrix, whose *order is n*.

$$\begin{bmatrix} 2 & 3 \\ 5 & -1 \\ 7 & 0 \end{bmatrix}$$

a 3×2 matrix

Note: The singular form of "matrices" is "*matrix*," **not** "MAY-trih-see"!

Applications of Matrices

Tons of applications, including:

- Solving systems of linear equations
- Computer Graphics, Image Processing
- Models within Computational Science & Engineering
- Quantum Mechanics, Quantum Computing
- Many, many more…

Matrix Equality

- Two matrices **A** and **B** are equal iff they have the same number of rows, the same number of columns, and all corresponding elements are equal.

$$\begin{bmatrix} 3 & 2 \\ -1 & 6 \end{bmatrix} \neq \begin{bmatrix} 3 & 2 & 0 \\ -1 & 6 & 0 \end{bmatrix}$$

Row and Column Order

- The rows in a matrix are usually indexed 1 to *m* from top to bottom. The columns are usually indexed 1 to *n* from left to right. Elements are indexed by row, then column.

$$\mathbf{A} = [a_{i,j}] = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Matrices as Functions

- An $m\times n$ matrix $\mathbf{A} = [a_{i,j}]$ of members of a set S can be encoded as a partial function
$$f_{\mathbf{A}}\colon \mathbb{N}\times\mathbb{N}\rightarrow S,$$
such that for $i<m$, $j<n$, $f_{\mathbf{A}}(i, j) = a_{i,j}$.
- By extending the domain over which $f_{\mathbf{A}}$ is defined, various types of infinite and/or multidimensional matrices can be obtained.

Matrix Sums

- The *sum* **A**+**B** of two matrices **A**, **B** (which **must** have the same number of rows, and the same number of columns) is the matrix (also with the same shape) given by adding corresponding elements.
- **A**+**B** = $[a_{i,j}+b_{i,j}]$

$$\begin{bmatrix} 2 & 6 \\ 0 & -8 \end{bmatrix} + \begin{bmatrix} 9 & 3 \\ -11 & 3 \end{bmatrix} = \begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix}$$

Matrix Products

- For an $m \times k$ matrix **A** and a $k \times n$ matrix **B**, the *product* **AB** is the $m \times n$ matrix:

$$\mathbf{AB} = \mathbf{C} = [c_{i,j}] \equiv \left[\sum_{\ell=1}^{k} a_{i,\ell} b_{\ell,j} \right]$$

- *I.e.*, element (i,j) of **AB** is given by the vector *dot product* of the <u>ith row of **A**</u> and the <u>jth column of **B**</u> (considered as vectors).
- Note: Matrix multiplication is **not** commutative!

Matrix Product Example

- An example matrix multiplication to practice in class:

$$\begin{bmatrix} 0 & 1 & -1 \\ 2 & 0 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & -1 & 1 & 0 \\ 2 & 0 & -2 & 0 \\ 1 & 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} \qquad\qquad\qquad \end{bmatrix}$$

Identity Matrices

- The *identity matrix of order n,* $\mathbf{I}_n$, is the order-n matrix with 1's along the upper-left to lower-right diagonal and 0's everywhere else.

$$\boldsymbol{I}_n = \left[\begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}\right] = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Review: Matrices, so far

Matrix sums and products:

$$\mathbf{A}+\mathbf{B} = [a_{i,j}+b_{i,j}]$$

$$\mathbf{AB} = \mathbf{C} = [c_{i,j}] \equiv \left[\sum_{\ell=1}^{k} a_{i,\ell} b_{\ell,j} \right]$$

Identity matrix of order n:

$\mathbf{I}_n = [\delta_{ij}]$, where $\delta_{ij}=1$ if $i=j$ and $\delta_{ij}=0$ if $i\neq j$.

Matrix Inverses

- For some (but not all) square matrices **A**, there exists a unique multiplicative *inverse* $\mathbf{A}^{-1}$ of **A**, a matrix such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$.
- If the inverse exists, it is unique, and $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1}$.
- We won't go into the algorithms for matrix inversion...

Matrix Multiplication Algorithm

```
procedure matmul(matrices A: $m×k$, B: $k×n$)
for $i := 1$ to $m$                                  $\Theta(m)\cdot$
    for $j := 1$ to $n$ begin                        $\Theta(n)\cdot($
        $c_{ij} := 0$                                $\Theta(1)+$
        for $q := 1$ to $k$                          $\Theta(k)\cdot$
            $c_{ij} := c_{ij} + a_{iq}b_{qj}$        $\Theta(1))$
    end {$\mathbf{C}=[c_{ij}]$ is the product of A and B}
```

What's the Θ of its time complexity?

Answer: $\Theta(mnk)$

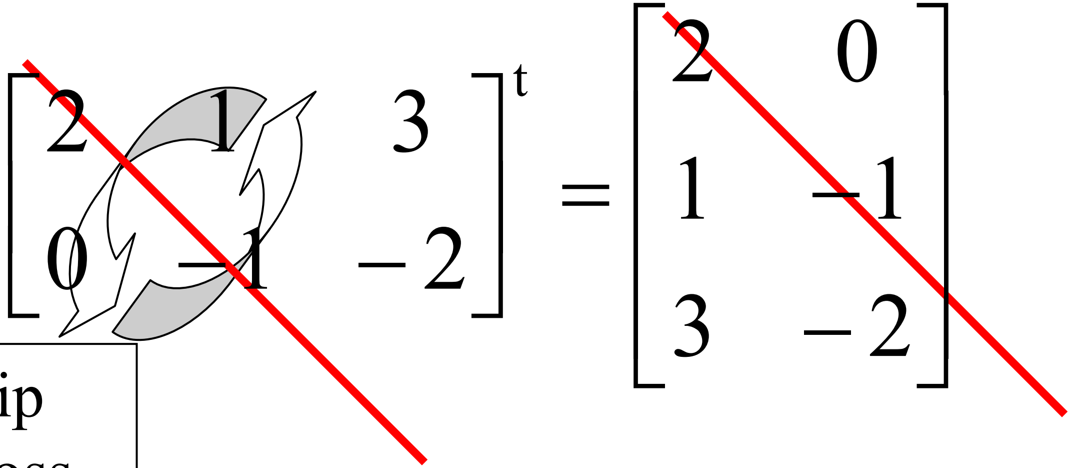
Powers of Matrices

If **A** is an $n \times n$ square matrix and $p \geq 0$, then:

- $\mathbf{A}^p \equiv \underbrace{\mathbf{AAA}\cdots\mathbf{A}}_{p \text{ times}} \quad (\mathbf{A}^0 \equiv \mathbf{I}_n)$
- Example: $\begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix}^3 = \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -1 & 0 \end{bmatrix}$

Matrix Transposition

- If **A**=[a_{ij}] is an *m*×*n* matrix, the *transpose* of **A** (often written $\mathbf{A}^t$ or $\mathbf{A}^T$) is the *n*×*m* matrix given by $\mathbf{A}^t = \mathbf{B} = [b_{ij}] = [a_{ji}]$ $(1 \le i \le n, 1 \le j \le m)$



$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & -1 & -2 \end{bmatrix}^t = \begin{bmatrix} 2 & 0 \\ 1 & -1 \\ 3 & -2 \end{bmatrix}$$

Flip across diagonal

Symmetric Matrices

- A square matrix **A** is *symmetric* iff **A=A**^{t}. *I.e.*, $\forall i,j \le n$: $a_{ij} = a_{ji}$.
- Which is symmetric?

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -2 & 1 & 3 \\ 1 & 0 & -1 \\ 3 & -1 & 2 \end{bmatrix} \begin{bmatrix} 3 & 0 & 1 \\ 0 & 2 & -1 \\ 1 & 1 & -2 \end{bmatrix}$$

Zero-One Matrices

- Useful for representing other structures.
  - *E.g.*, relations, directed graphs (later in course)
- All elements of a *zero-one* matrix are 0 or 1
  - Representing **False** & **True** respectively.
- The *meet* of **A, B** (both $m \times n$ zero-one matrices):
  - $\mathbf{A} \wedge \mathbf{B} :\equiv [a_{ij} \wedge b_{ij}] = [a_{ij}\, b_{ij}]$
- The *join* of **A, B**:
  - $\mathbf{A} \vee \mathbf{B} :\equiv [a_{ij} \vee b_{ij}]$

Boolean Products

- Let **A**=$[a_{ij}]$ be an $m \times k$ zero-one matrix, & let **B**=$[b_{ij}]$ be a $k \times n$ zero-one matrix,
- The *boolean product* of **A** and **B** is like normal matrix ×, but using ∨ instead + in the row-column "vector dot product."

$$\mathbf{A} \odot \mathbf{B} = \mathbf{C} = [c_{ij}] = \left[\bigvee_{\ell=1}^{k} a_{i\ell} \wedge b_{\ell j} \right]$$

Boolean Powers

- For a square zero-one matrix **A**, and any $k \geq 0$, the *kth Boolean power of* ***A*** is simply the Boolean product of *k* copies of **A**.
- $\mathbf{A}^{[k]} \equiv \underbrace{\mathbf{A} \odot \mathbf{A} \odot \ldots \odot \mathbf{A}}_{k \text{ times}}$