MTH 2215
APPLIED DISCRETE MATHEMATICS

Chapter 3, Section 3.1
Algorithms

These class notes are based on material from our textbook, **Discrete Mathematics and Its Applications**, 6th ed., by Kenneth H. Rosen, published by McGraw Hill, Boston, MA, 2006. They are intended for classroom use only and are **not** a substitute for reading the textbook.

Algorithms

What is an algorithm?

- An *algorithm* is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the *input* into the *output*. (CLRS, p. 5)

Algorithms

What is an algorithm?

- An *algorithm* is a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship . The algorithm describes a specific computational procedure for achieving that input/output relationship. (CLRS, p. 5)

Algorithms

Can an algorithm be specified in pseudocode?

In English?

In C++?

In the form of a hardware design?

YES to all!

Algorithms

Formal definition of the *sorting problem*:

Input: A sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Algorithms

Instance: The input sequence <14, 2, 9, 6, 3> is an *instance* of the sorting problem.

An *instance* of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Algorithms

Correctness: An algorithm is said to be *correct* if, for every instance, it halts with the correct output. We say that a correct algorithm *solves* the given computational problem.

Algorithms as a Technology

Efficiency: Algorithms that solve the same problem can differ enormously in their efficiency. Generally speaking, we would like to select the most efficient algorithm for solving a given problem.

Algorithms as a Technology

Space efficiency: Space efficiency is usually an all-or-nothing proposition; either we have enough space in our computer's memory to run a program implementing a specific algorithm, or we do not. If we have enough, we're OK; if not, we can't run the program at all. Consequently, analysis of the space requirements of a program tend to be pretty simple and straightforward.

Algorithms as a Technology

Time efficiency: When we talk about the efficiency of an algorithm, we usually mean the *time* requirements of the algorithm: how long would it take a program executing this algorithm to solve the problem? If we could afford to wait around forever (and could rely on the power company not to lose the power), it wouldn't make any difference how efficient our algorithm was in terms of time. But we can't wait forever; we need solutions in a reasonable amount of time.

Algorithms as a Technology

Space efficiency:

Note that space requirements set a minimum lower bound on the time efficiency of the problem.

Suppose that our data structure is a single-dimensioned array with $n = 100$ elements in it. Let's say that the first step in our algorithm is to execute a loop that just copies values into each of the 100 elements. Then our algorithm must take at least 100 iterations of the loop. So the running time of our algorithm is at least $O(n)$, just from setting up (initializing) our data structure!

Algorithms as a Technology

Algorithm analysis almost always involves loops:

The Theorem of Bohm and Giacopini proves that only three types of flow control are needed to execute any computable algorithm: sequential, selection, and repetition.

Sequential and selection statements usually require one step each.

Repetition statements include both explicit iteration (such as *for*, *while*, and *repeat* loops) and recursive function calls. To determine the running time of repetition statements, we have to analyze what is going on in the loops.

Bohm C. Giacopini G.: Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Communications of ACM, Vol. 9, No. 5, May 1966.

Algorithms as a Technology

Time efficiency of two sorts:

Suppose we use insertion sort to sort a list of numbers. Insertion sort has a time efficiency roughly equivalent to $c_1 \bullet n^2$. The value n is the number of items to be sorted. The value c_1 is a constant which represents the overhead involved in running this algorithm; it is independent of n.

Compare this to merge sort. Merge sort has a time efficiency of $c_2 \bullet n \bullet \lg n$ (where $\lg n$ is the same as $\log_2 n$).

Algorithms as a Technology

n	*Insertion sort - $O(n^{2})$*	*Merge sort – O(n lg n)*
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16,394	896
256	65,536	2048
512	262,144	4608
1024	1,048,576	10,240
1,048,576	~1,000,000,000,000	20,971,520

Algorithms as a Technology

Time efficiency of two sorts:

Do the two constants, c_1 and c_2, affect the result?

Yes, but only with low values of n.

Suppose that insertion sort is hand-coded in machine language for optimal performance and its overhead is very low, so that $c_1 = 2$.

Now suppose that merge sort is written in Ada by an average programmer and the compiler doesn't do a good job of optimization, so that $c_2 = 50$.

Algorithms as a Technology

Time efficiency of two sorts:

To make things worse, suppose that insertion sort is run on a machine that executes 1 billion instructions per second, while merge sort is run on a slow machine that executes only 10 million instructions per second.

Now let's sort 1 million numbers:

insertion sort: $\frac{2 \bullet \left(10^6\right)^2 instructions}{10^9 instructions/\sec} = 2000\sec.$

merge sort: $\frac{50 \bullet 10^6 \lg 10^6 instructions}{10^7 instructions/\sec} \approx 100\sec.$

Algorithms as a Technology

It is easy to see that merge sort wins, even though it has higher overhead expenses.

Why?

Because the merge sort algorithm itself is more is more efficient than the algorithm for insertion sort.

Analysis of algorithms helps us determine which ones are most efficient, and under what circumstances.

MTH 2215
APPLIED DISCRETE MATHEMATICS

Chapter 3, Section 3.2
The Growth of Functions

Asymptotic notations

- Asymptotic efficiency of algorithms
  - How does the running time of an algorithm increase as the input increases in size without bound?
- Asymptotic notations
  - Define sets of functions that satisfy certain criteria and use these to characterize time and space complexity of algorithms

Big-*O*

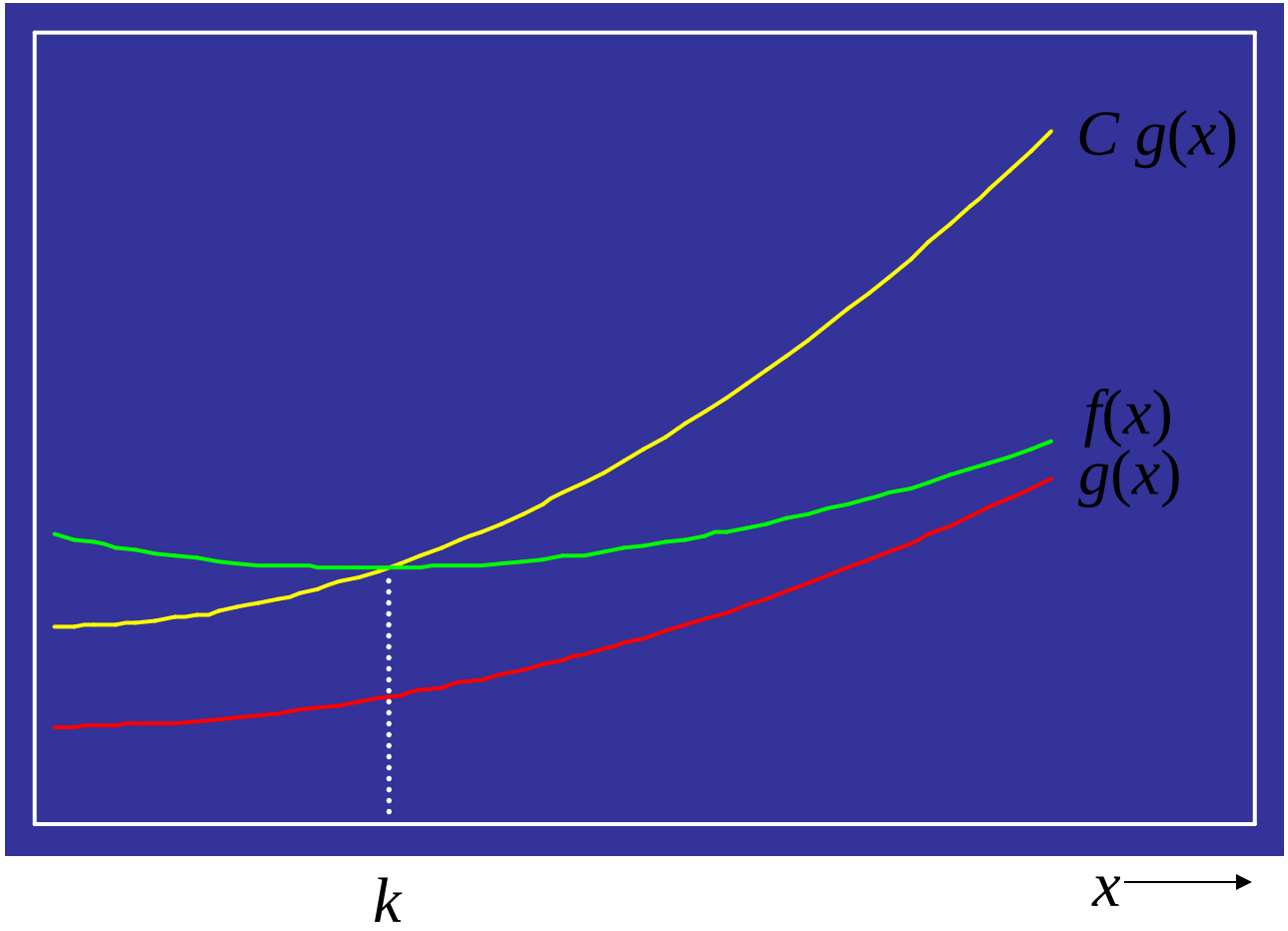
- Let *f* and *g* be functions from **N** or **R** to **R**. We say that "*f*(*x*) is *O*(*g*(*x*))" if there are positive constants *C* and *k* such that

$$|f(x)| \leq C\ |g(x)|$$

  whenever $x > k$.

- Read as: "*f*(*x*) is big-oh of *g*(*x*)"

Big-*O*



Big-*O*

- Big-*O* provides an upper bound on a function (to within a constant factor)
- $O(g(x))$ is a *set* of functions
- Commonly used notation

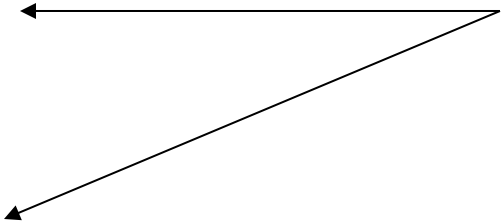
  $f(x) = O(g(x))$

- Correct notation

  $f(x) \in O(g(x))$

- Meaningless statement

  $O(g(x)) = f(x)$



Example

- Show that $x^2 + 2x + 1$ is $O(x^2)$.
- We know that $f(x) = O(g(x))$ if there are positive constants C and k such that $|f(x)| \leq C\ |g(x)|$ whenever $x > k$.
- Can we find such constants C and k?
- Yes: represent $x^2 + 2x + 1$ as $\underline{1} \bullet x^2 + \underline{2} \bullet x + \underline{1}$
- Choose C to be $1 + 2 + 1 = 4$
- Choose $k = 1$
- Is $x^2 + 2x + 1 \leq 4x^2$ whenever $x > 1$? Yes!

Example

- Show that $2x^3 + 3x^2 + 1$ is $O(x^3)$.
- We know that $f(x) = O(g(x))$ if there are positive constants C and k such that $|f(x)| \leq C\ |g(x)|$ whenever $x > k$.
- Can we find such constants C and k?
- Yes: represent $2x^3 + 3x^2 + 1$ as $\underline{2} \bullet x^3 + \underline{3} \bullet x^2 + \underline{1}$
- Choose C to be $2 + 3 + 1 = 6$
- Choose $k = 1$
- Is $2x^3 + 3x^2 + 1 \leq 6x^3$ whenever $x > 1$? Yes!

Example

- Show that $7x^2$ is $O(x^3)$.
- We know that $f(x) = O(g(x))$ if there are positive constants C and k such that $|f(x)| \leq C\ |g(x)|$ whenever $x > k$.
- Can we find such constants C and k?
- Yes: represent $7x^2$ as $7 \cdot x^2$
- Choose C to be 7
- Choose $k = 1$
- Is $7x^2 \leq 7x^3$ whenever $x > 1$? Yes!

Properties of Big-O

- If "$f(x)$ is $O(g(x))$" then $g(x)$ grows at least as fast as $f(x)$
- "$f(x)$ is $O(g(x))$" iff $O(f(x)) \subseteq O(g(x))$
- If "$f(x)$ is $O(g(x))$" and "$g(x)$ is $O(f(x))$" then $O(f(x)) = O(g(x))$
- If "$f(x)$ is $O(g(x))$" and "$h(x)$ is $O(g(x))$" then $(f + h)(x)$ is $O(g(x))$

Properties of Big-O (Cont..)

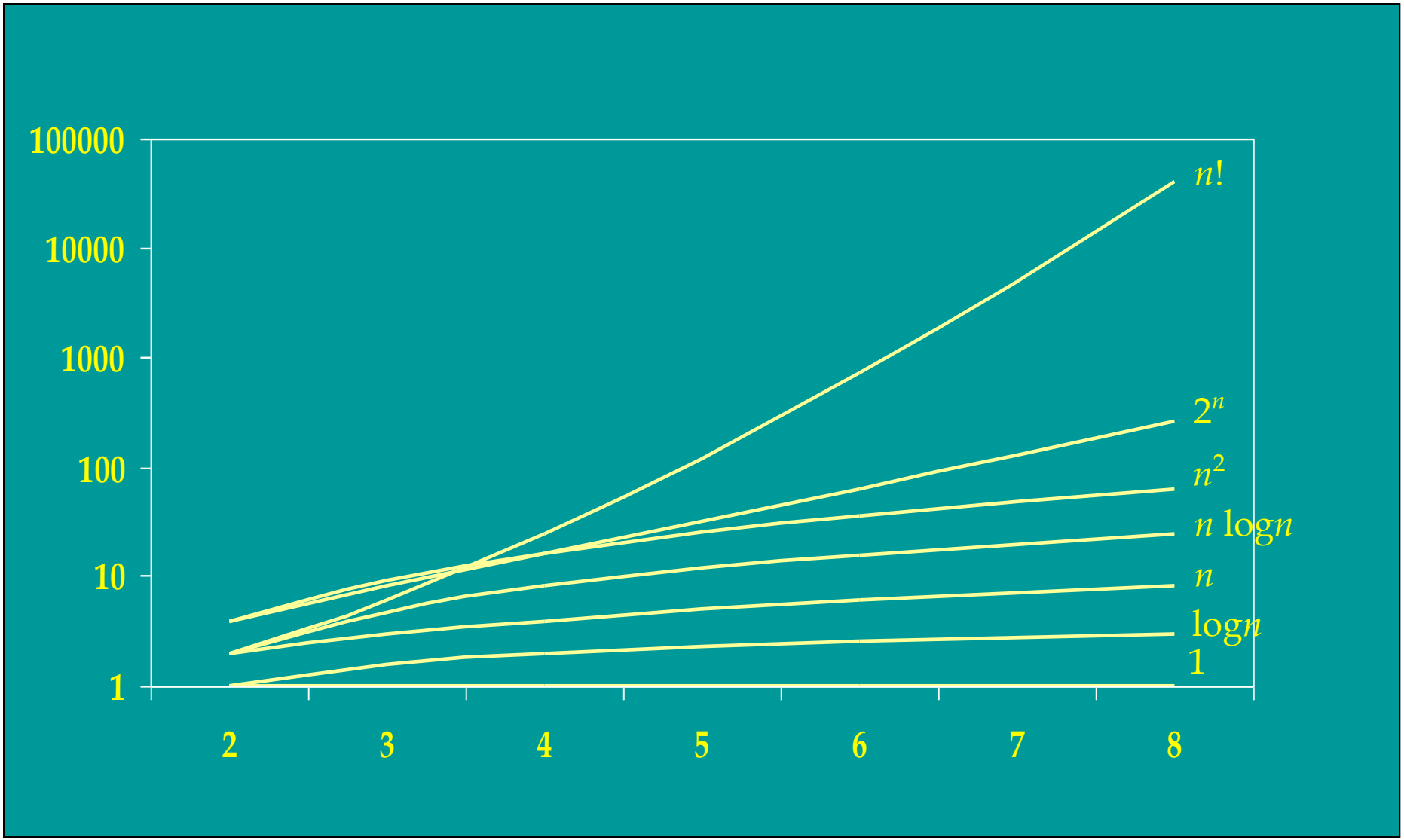
- If a is a scalar and "$f(x)$ is $O(g(x))$", then
  $a*f(x)$ is $O(g(x))$
- If "$f(x)$ is $O(g(x))$" and "$g(x)$ is $O(h(x))$", then
  $f(x)$ is $O(h(x))$
- $f(x) = O(g(x))$

In practice, $g(x)$ is chosen to be as small as possible.

Common Complexity Functions

Complexity	**Term**
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^b)$	polynomial
$O(b^n)$, where $b > 1$	exponential
$O(n!)$	factorial

Growth of Some Common Functions



Important Big-*O* Result

- Let $f(x)=a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers.

- Then $f(x)$ is $O(x^n)$.

Big-*O* Estimates

What is the big-*O* estimate for $n!$?

$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

$n! \leq n \cdot n \cdot n \cdot \ldots \cdot n$

$n! \leq n^n$

So: $n!$ is $O(n^n)$

Big-O Estimates

What is the big-O estimate for $\log(n!)$?

$n! \leq n^n$

$\Rightarrow \log n! \leq \log n^n$

$\Rightarrow \log n! \leq n \log n$

So: $\log n!$ is $O(n \log n)$

Growth of Combinations of Functions

Addition of functions –

Theorem 2:

If $f_1(x)$ is $O(g_1(x))$ and
$f_2(x)$ is $O(g_2(x))$, then
$(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$.

Example: What is the complexity of the function $2n^2 + 3$ n log n ?

Example

What is the complexity of the function $f(n) = 2n^2 + 3\ n \log n$?

We know that $2n^2$ is $O(n^2)$.

We know that $3\ n \log n$ is $O(n \log n)$.

According to theorem 2, if $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$.

Which is bigger, $O(n^2)$ or $O(n \log n)$?

So $2n^2 + 3\ n \log n$ is just $O(n^2)$

Growth of Combinations of Functions

Multiplication of functions -

Theorem 3:

If $f_1(x)$ is $O(g_1(x))$ and
$f_2(x)$ is $O(g_2(x))$, then
$(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

Example: What is the complexity of the function $3n\log(n!)$?

Example

What is the complexity of the function $f(n) = 3n \bullet \log(n!)$?

We know that $\log(n!)$ is $O(n \log n)$.

We know that $3n$ is $O(n)$.

According to theorem 3, if $f1(x)$ is $O(g1(x))$ and $f2(x)$ is $O(g2(x))$, then $(f1f2)(x)$ is $O(g1(x)g2(x))$.

So $3n \bullet \log(n!)$ is $O(n \bullet (n \log n))$, which is $O(n^2 \log n)$

Big-Omega

$f(x)=O(g(x))$ only provides an upper bound in terms of $g(x)$ for $f(x)$.

What do we do for a lower bound?

Big-Omega (Ω) provides a lower bound for a function to within a constant factor.

Big-Omega

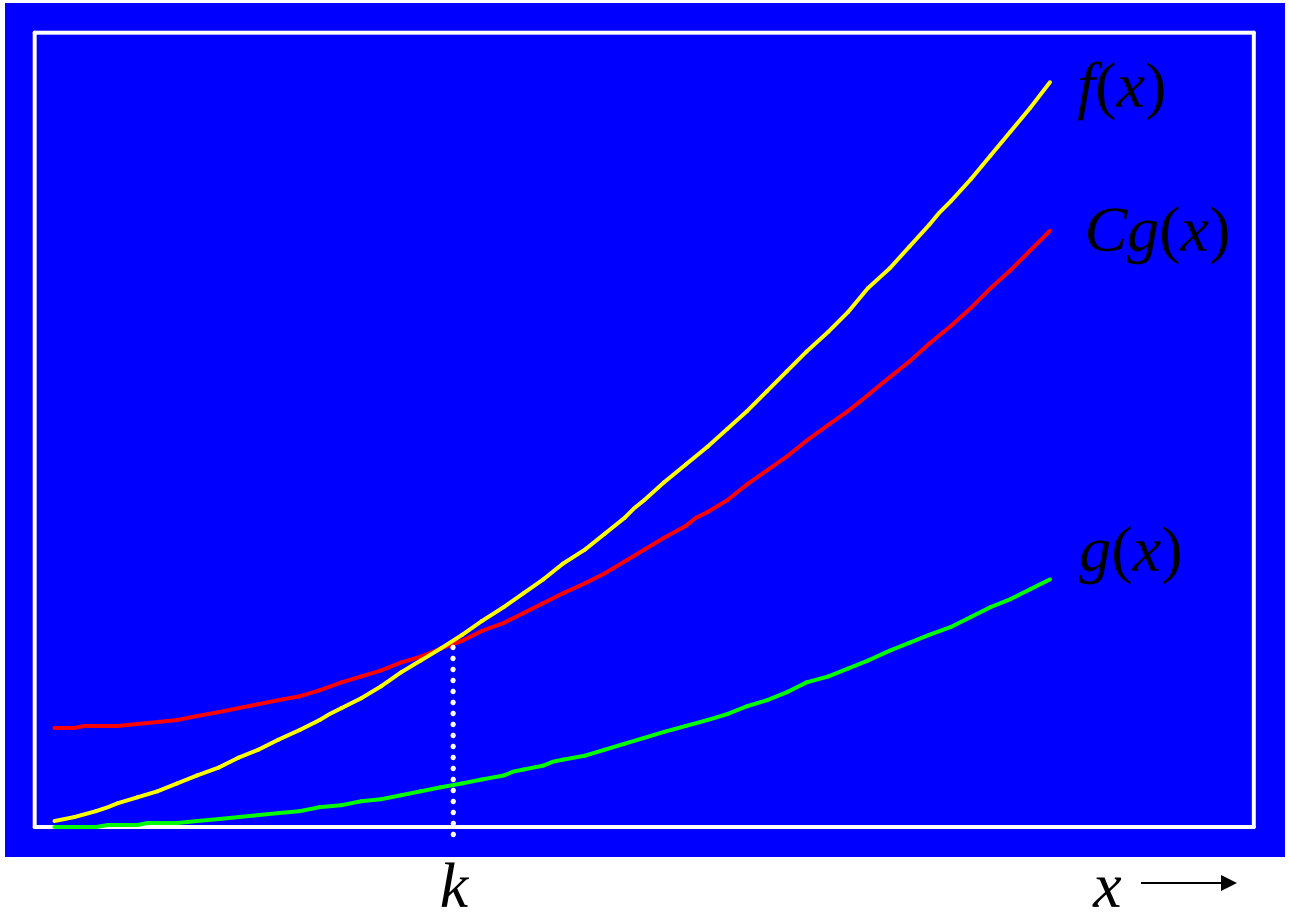
Let f and g be functions from **N** or **R** to **R**. We say that "$f(x)$ is $\mathbf{\Omega}(g(x))$" if there are positive constants C and k such that

$$|f(x)| \geq C\,|g(x)|$$

whenever $x > k$.

Read as: "$f(x)$ is big-Omega of $g(x)$"

Big-Omega



Example

Show that $8x^3 + 5x^2 + 7$ is $\Omega(x^3)$.

We say that "$f(x)$ is $\mathbf{\Omega}(g(x))$" if there are positive constants C and k such that $|f(x)| \geq C\,|g(x)|$ whenever $x > k$.

Can we find appropriate values for C and k?

Yes. $8x^3 + 5x^2 + 7$ is larger than x^3, so C = 1 works fine.

And when $x = 0$, the function $f(x) = 8x^3 + 5x^2 + 7 = 7$, while $g(x) = x^3 = 0$, so pick $k = 0$.

Big-Theta

$f(x) = O(g(x))$ only provides an upper bound in terms of $g(x)$ for $f(x)$.

$f(x) = \Omega(g(x))$ only provides a lower bound in terms of $g(x)$ for $f(x)$.

Big-Theta (Θ) provides both an upper bound and a lower bound for a function in terms of a reference function $g(x)$.

Big-Theta

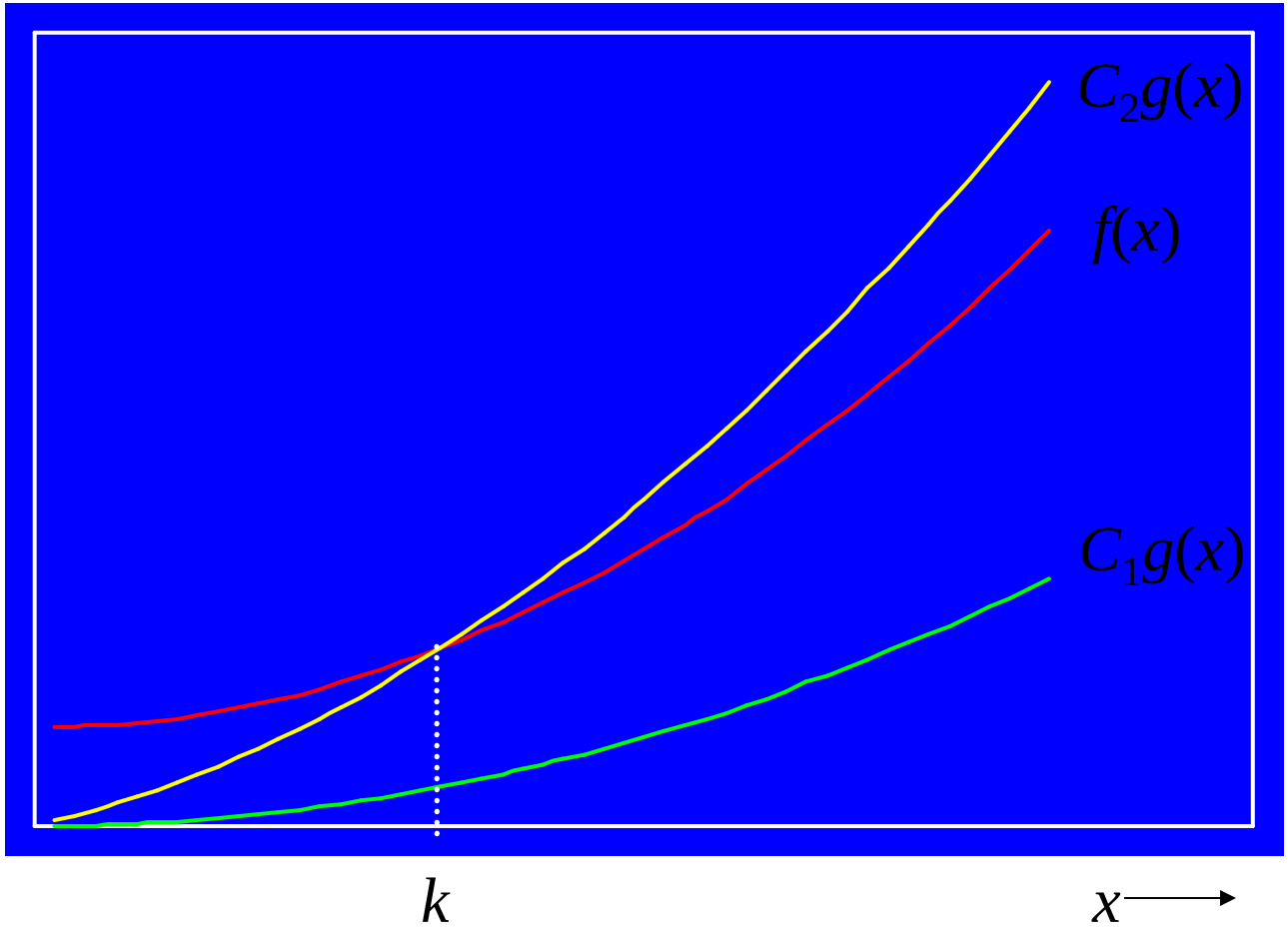
Let f and g be functions from **N** or **R** to **R**. We say that "$f(x)$ is $\Theta(g(x))$" if "$f(x)$ is $O(g(x))$" and "$f(x)$ is $\Omega(g(x))$", i.e., there are positive constants C_1, C_2, and k such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

whenever $x > k$.

Read as: "$f(x)$ is big-Theta of $g(x)$", or "$f(x)$ is of order $g(x)$"

Big-Theta



Example

Show that $3x^2 + x + 1$ is $\Theta(3x^2)$.

We say that "$f(x)$ is $\Theta(g(x))$" if "$f(x)$ is $O(g(x))$" and "$f(x)$ is $\Omega(g(x))$", i.e., there are positive constants c_1, c_2, and k such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ whenever $x > k$.

Can we find appropriate values for C_1, C_2, and k?

Yes.

$3x^2 \leq (3x^2 + x + 1)$ for all $x > 0$, so $C_1 = 3$.

$(3x^2 + x + 1) \leq (3x^2 + 3x^2)$, which $= 2 \bullet 3x^2$, for all $x > 1$, so $C_2 = 2$, and $k = 1$.

MTH 2215
APPLIED DISCRETE MATHEMATICS

Chapter 3, Section 3.3
The Complexity of Algorithms

Computational Complexity

- Means the cost of a program's execution
  - Running time, memory requirement, ...
- Doesn’t mean:
  - The cost of creating the program, i.e., # of statements, development time, etc.
- In this context, programs with lower complexity may require more development time.

Computational Complexity

- *Time Complexity*: Gives the approximate number of operations required to solve a problem of a given size.
- *Space Complexity*: Gives the approximate amount of memory required to solve a problem of a given size.

Different types of analysis

- Worst-case analysis
  - Maximum number of operations
  - Is a guarantee over all inputs of a given size
- Best-case analysis
  - Minimum number of operations
  - Not very practical
- Average-case analysis
  - Average number of operations over all possible inputs of a given size
  - Done with an assumed input probability distribution
  - Can be complicated

Common Complexity Functions

Complexity	**Term**
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^b)$	polynomial
$O(b^n)$, where $b > 1$	exponential
$O(n!)$	factorial

Actual time used by Algorithms

- An algorithm requires $f(n)$ bit operations where $f(n) = 2^n$. How much time is needed to solve a problem of size 50 if each bit operation takes 10^{-6} second?
- An algorithm requires $f(n)$ bit operations where $f(n) = n^2$. How large a problem can be solved in 1 second using this algorithm if each bit operation takes 10^{-6} second?

Types of Problems

- *Tractable* - A problem that is solvable using an algorithm with reasonable (low order polynomial) worst-case complexity
- *Intractable* - A problem that cannot be solved using an algorithm with reasonable worst-case complexity
- *Unsolvable* - A problem for which it can be shown that no algorithm exists for solving them

Complexity of Statements

- Simple statements (i.e., initialization of variables) have a complexity of $O(1)$.
- Conditional statements have a complexity of $O(\max(f(n),g(n)))$, where $f(n)$ is upper bound on **then** part and $g(n)$ is upper bound on **else** part.

Complexity of Statements (Cont.)

- Loops have a complexity of $O(g(n)f(n))$, where $g(n)$ is an upper bound on the number of loop iterations and $f(n)$ is an upper bound on the body of the loop.
  - If $g(n)$ and $f(n)$ are constant, then this is constant time.

Complexity of Statements (Cont.)

Repetitive halving or doubling of a loop counter results in *logarithmic* complexity.

$i \leftarrow 1$ **while** ($i \leq n$) display i $i = i * 2$	$i \leftarrow$ n **while** ($i \geq 1$) display i $i = i / 2$

Both of the above loops have O(log(n)) complexity.

Complexity of Statements (Cont.)

- Blocks of statements with complexities $f_1(n)$, $f_2(n)$, …, $f_k(n)$, have complexity $O(f_1(n) + f_2(n) + ... + f_k(n))$.

Analysis of Algorithms

Linear search:

```
   procedure linearSearch (x; a_1, …, a_n)
   (Note: x is an integer, a is an array
   of distinct integers)
1   i := 1
2   while (i ≤ n) and (x ≠ a_i)
3     i := i + 1
4   if i ≤ n
5     then location := i
6     else location := 0

(Note: location is subscript of element
   that equals x, or 0 if x not found)
```

Analysis of Algorithms

Lines 1, 4, 5, and 6 of the Linear Search algorithm require one step each to execute.

The running time of the Linear Search is dominated by the cost of the *while* loop in lines 2 and 3.

What is the *worst case*? Either *x* isn't found in the array, or it is found in the last element. That will take *n* steps, where *n* is the number of elements in the array.

Analysis of Algorithms

What is the *best case*? The element we are looking for is found in the first element of the array. In that case, the body of the *while* loop will execute only once.

What is the *average case*? The element we are looking for is found in the middle element of the array. In that case, the body of the *while* loop will execute $n/2$ times.

Analysis of Algorithms

What is the running time of the Linear Search algorithm?

Worst case = $O(n)$

Average case = $O(n)$

Best case = $O(1)$

If someone asks us for the running time of an algorithm, we usually give the running time for the worst case. (Why?)

Analysis of Algorithms

Binary search:

```
    procedure BinarySearch (x; a_1, …, a_n)
    (Note: x is an integer, a is a sorted array of
     integers)
1   i := 1 (i is left endpoint of search interval)
2   j := n (j is right endpoint of search interval)
3   while i < j do
4   begin
5     m := ⌊ (i + j) / 2 ⌋
6     if x > a_m then i := m + 1
7       else j := m
8   end
9   if x = a_i then location := 1
10    else location := 0
```

Analysis of Algorithms

What is the running time of the Binary Search algorithm?

Again, the cost of the Binary Search algorithm is dominated by the cost of the *while* loop.

Each iteration through the loop elininates ½ of the remaining elements. Repeatedly halving n takes $\log_2 n$ steps to reach a single element. At that point, either x is found, or it isn't in the array. Every case takes the same amount of time.

Worst case = $O(\log n)$

Average case = $O\log n)$

Best case = $O(\log n)$

Analysis of Algorithms

Bubble sort:

```
  procedure bubbleSort (a1, …, an)
  (Note: a is a real array with n ≥ 2)
1 count := 0
2 for i := 1 to n – 1
3    for j := 1 to n – i
4       count := count + 1
5       if aj > aj+1
6          then swap aj and aj+1
```

Analysis of Algorithms

Line 4 is nested within two loops, so the number of times it will be executed is:

# of times outer loop is executed * # of times inner loop is executed

The outer loop executes n -1 times

The inner loop executes n - i times. In the first iteration the inner loop executes n – 1 times, and in the last iteration the loop executes 1 time.

Analysis of Algorithms

The total number of times this line will be executed will be:

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{(n-1)n}{2}$$

This is $(n^2 - n) / 2$, which is $O(n^2)$. This is the worst-case performance (or worst-case complexity) of Bubble sort – also the best case and average case.

Analysis of Algorithms

Insertion sort:

```
procedure insertionSort (a1, …, an)
1     count := 0
2     for j := 2 to n
3     begin
4         i := 1
5         while aj > ai
6             count := count + 1
7             i := i + 1
8         m := aj
9         for k := 0 to j – i – 1
10            aj-k := aj-k-1
11        ai = m
12    end
```

Analysis of Algorithms

Line 6 is nested within two loops, so the number of times it will be executed is:

# of times outer loop is executed *
# of times inner loop is executed.

The outer loop executes *n* -1 times.

The inner loop executes until $a_j \leq a_i$, which in the worst case can take *j* steps.

Analysis of Algorithms

The total number of times this line will be executed will be:

$$2 + 3 + \ldots + n = \frac{n(n+1)}{2} - 1$$

This is $((n^2 + n) / 2) - 1$, which is $O(n^2)$. This is the worst-case performance of Insertion sort. (But if the smaller elements start off at the end of the list, we get better performance.)

MTH 2215
APPLIED DISCRETE MATHEMATICS

Chapter 3, Section 3.4
The Integers and Division

Division

- $a|b$: If a and b are integers with $a \neq 0$, we say that a divides b (or $a|b$) if there is an integer c such that $b = ac$.
  - a is a factor of b
  - b is a multiple of a
- If a, b, and c are integers:
  - if $a|b$ and $a|c$, then $a|(b+c)$
  - if $a|b$, then $a|bc$ for all integers c
  - if $a|b$ and $b|c$, then $a|c$

Division Algorithm

- Let a be an integer and d a positive integer. Then there are unique integers q and r with $0 \leq r < d$ such that $a = dq + r$

  d is the *divisor*

  a is the *dividend*

  q is the *quotient*

  r is the *remainder* (Note: has to be positive)

We say that $q = a$ **div** d, and $r = a$ **mod** d.

Division Algorithm

- Example: What are the quotient and remainder when 101 is divided by 11?
- The division algorithm tells us that we have unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- So substitute 101 for a and 11 for d:

  $101 = 11\ (q) + r$

- Now solve the equation (see next slide)

Division Algorithm

Given the equation $101 = 11\ (q) + r$, what numbers can we substitute for q and r to make this equation true?

We know that $101 / 11 = 9.18\ldots$, so we might try to replace q with 9:

$$101 = 11\ (9) + r$$

$$101 = 99 + r$$

So, $r = 2$.

Division Algorithm

- Example: What are the quotient and remainder when –11 is divided by 3?
- Watch out! We have a negative dividend!
- The division algorithm tells us that we have unique integers q and r with $0 \leq r < d$ such that $a = dq + r$.
- So substitute -11 for a and 3 for d:

  -11 = 3 (q) + r

- Now solve the equation (see the next slides)

Division Algorithm

What is the strategy with negative numbers?

Use the equation $a = dq + r$, with $0 \leq r < d$.

The product of the divisor d and the quotient q must either:

a) exactly equal the dividend a, so that the remainder r is 0, or

b) produce a “more negative” negative number, so that a positive remainer r can be added to the dq to equal the dividend.

Division Algorithm

Given the equation -11 = 3 (q) + r ,
what numbers can we substitute for *q* and *r* to make this equation true?

We know that -11 / 3 = -3.67…, so we might try to replace *q* with -3:

-11 = 3 (-3) + r

-11 = -9 + r, which would make *r* = -2

But remember that $0 \leq r < d$; *r* must be positive!

Division Algorithm

So let's try again: given the equation
-11 = 3 (q) + r , what numbers can we substitute for *q* and *r* to make this equation true?

This time, we replace *q* with -4:

-11 = 3 (-4) + r

This gives:

-11 = -12 + r, which would make *r* = 1

Since *r* must be positive, this is correct.

Modular Arithmetic

-6	-5	-4	-3	-2	-1	0	1	2	3	***4***	5	6

Let's find 5 mod 2.
What is the largest number *less than* 5 divisible by 2? ***4***
What *positive* number do we have to add to this number to get 5? ***1***

Let's find -5 mod 2.
What is the largest number *less than* -5 divisible by 2? ***-6***
What *positive* number do we have to add to this number to get -5? ***1***

Modular Arithmetic

- If a is an integer and m a positive integer, a **mod** m is the remainder when a is divided by m.
- If $a = qm + r$ and $0 \leq r < m$, then
  a **mod** $m = r$
- Example: Find 17 **mod** 5.
- Example: Find –133 **mod** 9.

Modular Arithmetic

- Example: Find 17 **mod** 5.

  $a = dq + r$

  $17 = 5(q) + r$

  We know 17 / 5 = 3.4, so set q to 3

  $17 = 5(3) + r$

  $17 = 15 + r$

  17 = 15 + 2, so $r = 2$ and 17 **mod** 5 = 2.

Modular Arithmetic

- Example: Find -133 **mod** 9.

  $a = dq + r$

  $-133 = 9(q) + r$

  We know -133 / 9 = -14.7. Choosing q = -14 isn't going to work, because 9 • -14 = -126, and we can't add a positive remainder r to -126 to get -133. So choose q = - 15.

  $-133 = 9(-15) + r$

  $-133 = -135 + r$, so $r = 2$, and -133 **mod** 9 = 2.

Modular Arithmetic (Cont.)

- Let a and b be integers and m be a positive integer.

  a is congruent to b modulo m if (a-b) is divisible by m.
- Notation: $a \equiv b \pmod{m}$
- $a \equiv b \pmod{m}$ iff (a **mod** m)=(b **mod** m)
- Let m be a positive integer.

  $a \equiv b \pmod{m}$ iff there is an integer k such that $a = b + km$.

Modular Arithmetic (Cont.)

- Is 17 congruent to 5 modulo 6? That is, is $17 \equiv 5 \pmod 6$?

We know that $a \equiv b \pmod m$ iff there is an integer k such that $a = b + km$.

So we ask if there exists some integer k such that $17 = 5 + k \bullet 6$

Subtract 5 from both sides: $12 = k \bullet 6$

Divide both sides by 6: $2 = k$

So there is an integer, $k = 2$, such that $a = b + km$.

Modular Arithmetic (Cont.)

- Is 24 congruent to 14 modulo 6, i.e., is $24 \equiv 14 \pmod 6$?

We know that $a \equiv b \pmod m$ iff there is an integer k such that $a = b + km$.

So we ask if there exists some integer k such that $24 = 14 + k \bullet 6$

Subtract 14 from both sides: $10 = k \bullet 6$

Divide both sides by 6: $10/6 = 5/3 = 1.67 = k$

No; there is no integer, k, such that $a = b + km$.

Modular Arithmetic (Cont.)

- Let m be a positive integer.

  If $a \equiv b \pmod{m}$ and

  $c \equiv d \pmod{m}$,

  then $a + c \equiv b + d \pmod{m}$

  $ac \equiv bd \pmod{m}$

- $7 \equiv 2 \pmod{5}$ and $11 \equiv 1 \pmod{5}$
- $(7 + 11) \equiv (2+1) \pmod{5}$ *or*, $18 \equiv 3 \pmod{5}$
- $(7 \cdot 11) \equiv (2 \cdot 1) \pmod{5}$ *or*, $77 \equiv 2 \pmod{5}$

Applications of Modular Arithmetic

- Hashing functions
- Pseudorandom number generation
- Cryptography

MTH 2215
APPLIED DISCRETE MATHEMATICS

Chapter 3, Section 3.5
Primes and the Greatest Common Divisors

Primes

- A positive integer *p* is called *prime* if the only positive factors of *p* are 1 and *p*.
  - Otherwise *p* is called a *composite*.
- Is 7 prime?
- Is 9 prime?

Prime factorization

- Every positive integer can be written uniquely as the product of primes, with the prime factors written in increasing order.
- Example - Find the prime factorization of these integers: 100, 641, 999, 1024
- $100 = 10 \cdot 10 = (5 \cdot 2) \cdot (5 \cdot 2) = 2 \cdot 2 \cdot 5 \cdot 5 = 2^2 5^2$
- $641 = 641$ (a prime)
- $999 = 27 \cdot 37 = 3^3 \cdot 37$
- $1024 = 2^{10}$

Prime factorization (Cont.)

- If n is a composite integer, then n has a prime factor less than or equal to $\sqrt{n}$.
- Example - Show that 101 is prime.
- The square root of is ≈ 10.05. The primes ≤ 10.05 are 2, 3, 5, and 7. But 101 is not evenly divisible by 2, 3, 5, or 7. Thus, 101 must itself be a prime number.

Greatest Common Divisor

- Let a and b be integers, not both zero. The *greatest common divisor* (gcd) of a and b is the largest integer d such that $d|a$ and $d|b$.
- Notation: $\gcd(a,b) = d$
- Example: What is the gcd of 45 and 60?

Greatest Common Divisor

- What is the gcd of 45 and 60?
- The positive divisors of 45 are 3, 5, 9, and 15.
- The positive divisors of 60 are 2, 3, 4, 5, 6, 10, 12, 15, 20, and 30.
- The common positive divisors of 45 and 60 are 3, 5, and 15.
- The greatest common divisor is 15.

Greatest Common Divisor (Cont.)

- gcd(a,b) can be computed using the prime factorizations of a and b.
- $a = p_1^{a1} p_2^{a2} \ldots p_n^{an}$
- $b = p_1^{b1} p_2^{b2} \ldots p_n^{bn}$
- gcd(a,b) =

$$p_1^{\min(a1,b1)} p_2^{\min(a2,b2)} \ldots p_n^{\min(an,bn)}$$

Greatest Common Divisor (Cont.)

- Find gcd(120, 500).
- Let's solve this in two ways. First method:
- The positive divisors of 120 are: 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60
- The positive divisors of 500 are: 2, 4, 5, 10, 20, 25, 50, 100, 125, 250
- The common divisors of 120 and 150 are: 2, 4, 5, 10, and 20
- The greatest common divisor is 20

Greatest Common Divisor (Cont.)

- Find gcd(120, 500).
- Second method:
- The prime factorization of 120 is: 2^3, 3, and 5
- The prime factorization of 500 is: 2^2, 5^3
- gcd(120,500) =

$$p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \ldots p_n^{\min(a_n,b_n)}$$

$$= 2^{\min(3,2)} \bullet 3^{\min(1,0)} \bullet 5^{\min(1,3)} = 2^2 \bullet 3^0 \bullet 5^1$$

- So the greatest common divisor = 20

Greatest Common Divisor (Cont.)

- Definition: The integers a and b are *relatively prime* if their greatest common divisor is 1; that is, $\gcd(a,b) = 1$.
- Are 17 and 22 are relatively prime?
- Yes; 17 and 22 have no positive common divisors other than 1, so they are relatively prime.

Greatest Common Divisor (Cont.)

- A set of integers $a_1, a_2, \ldots, a_n$ are *pairwise relatively prime* if the gcd of every possible pair is 1.
- Are 10, 17, 21 pairwise relatively prime?

  gcd(10, 17) = 1

  gcd(17, 21) = 1

  gcd(10, 21) = 1
- Therefore, 10, 17, and 21 are pairwise relatively prime.

Greatest Common Divisor (Cont.)

Are 10, 19, 24 pairwise relatively prime?

gcd(10, 19) = 1

gcd(19, 24) = 1

gcd(10, 24) = 2

Since gcd(10, 24) = 2, these numbers are *not* pairwise relatively prime.

Least Common Multiple

- The *least common multiple* (lcm) of the positive integers a and b is the smallest positive integer m such that $a|m$ and $b|m$.
- Notation: $\text{lcm}(a,b) = m$
- Example: What is the lcm of 6 and 15?
- Certainly 90 (6 • 15) is divisible by both 6 and 15, but is there a smaller number divisible by both? Yes: 30.

Least Common Multiple (Cont.)

- lcm(a,b) can be computed using the prime factorizations of a and b.
- $a = p_1^{a1} p_2^{a2} \ldots p_n^{an}$
- $b = p_1^{b1} p_2^{b2} \ldots p_n^{bn}$
- lcm(a,b) =

$$p_1^{\max(a1,b1)} p_2^{\max(a2,b2)} \ldots p_n^{\max(an,bn)}$$

Least Common Multiple (Cont.)

- Example: Find lcm(120, 500).
- What are the prime factorizations of 120 and 500?

$120 = 12 \bullet 10 = 6 \bullet 2 \bullet 5 \bullet 2 = 2^3 \bullet 3^1 \bullet 5^1$

$500 = 100 \bullet 5 = 10 \bullet 10 \bullet 5 = 2 \bullet 5 \bullet 2 \bullet 5 \bullet 5 = 2^2 \bullet 5^3$

$\text{lcm}(2^3 3^1 5^1, 2^2 5^3) = 2^{\max(3, 2)} 3^{\max(1, 0)} 5^{\max(1, 3)} =$

$2^3 3^1 5^3 = 3000$

Relationship between gcd and lcm

- If a and b are positive integers, then

$$ab = \gcd(a,b) \bullet \operatorname{lcm}(a,b)$$

- Example:

$$\gcd(120, 500) \bullet \operatorname{lcm}(120, 500)$$
$$= 20 \bullet 3000$$
$$= 60000$$
$$= 120 \bullet 500$$

Conclusion

- In this chapter we have covered:

Algorithms –

- Definition
- Efficiency (Big-Oh and Big-Theta, especially)
- Complexity

Integers: Division and Modulo

Primes

Greatest Common Divisor

Least Common Multiple