Lecture 3
Growth of Functions

Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

Lecture 3 Topics

- Asymptotic notation
- Comparison of functions
- Standard notations and common functions

Asymptotic notation

What does asymptotic mean?

Asymptotic describes the behavior of a function *in the limit* - for sufficiently large values of its parameter.

Asymptotic notation

The *order of growth* of the running time of an algorithm is defined as the highest-order term (usually the leading term) of an expression that describes the running time of the algorithm. We ignore the leading term's constant coefficient, as well as all of the lower order terms in the expression.

Example: The order of growth of an algorithm whose running time is described by the expression $an^2 + bn + c$ is simply n^2.

Big O

Let's say that we have some function that represents the sum total of all the running-time costs of an algorithm; call it f(n).

For merge sort, the actual running time is:

$$f(n) = cn(\log_2 n) + cn$$

We want to describe the running time of merge sort in terms of another function, *g(n)*, so that we can say *f(n)* = *O(g(n))*, like this:

$$cn(\log_2 n) + cn = O(n\log_2 n)$$

Big O

Definition:

For a given function g(n), O(g(n)) is the set of functions

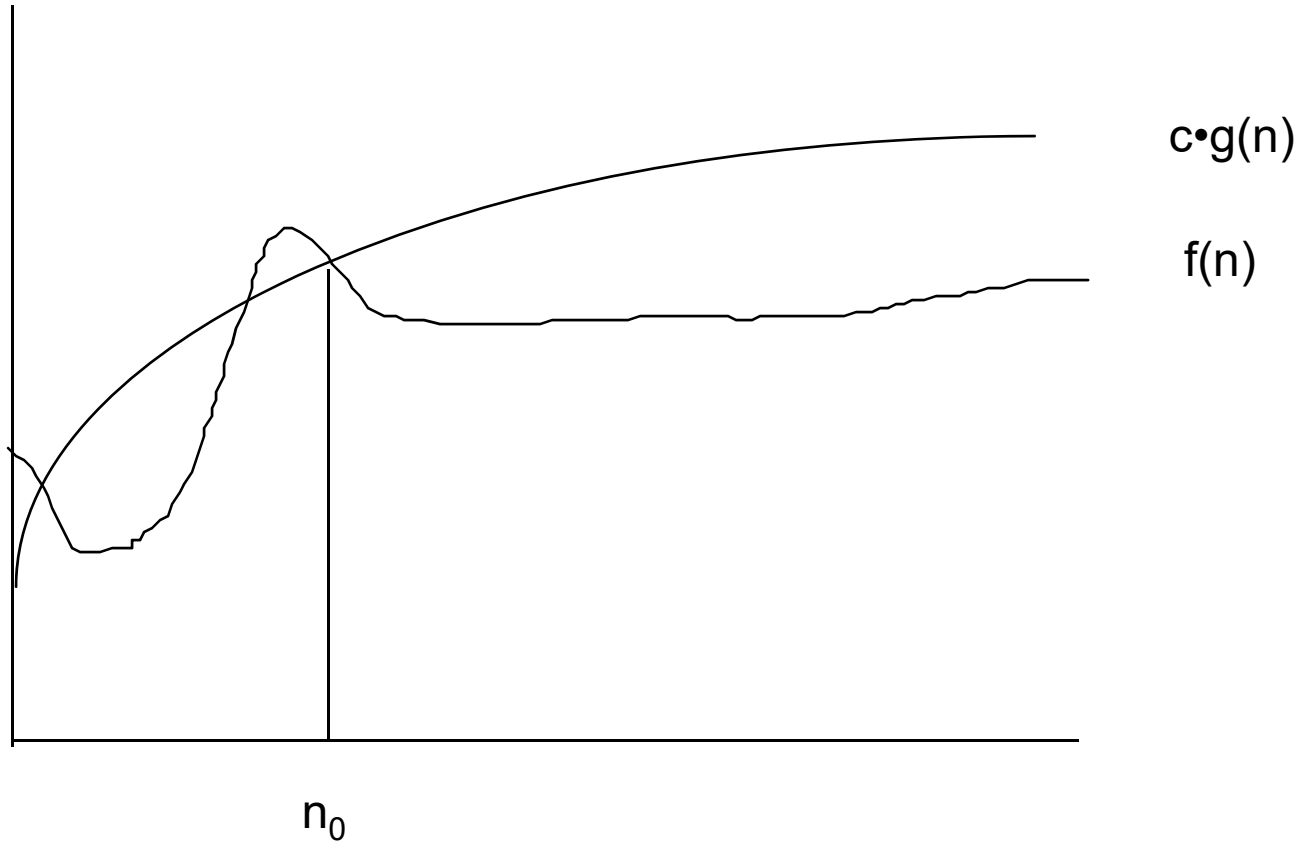
O(g(n)) = { f(n): there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0 \}$$

c is the multiplicative constant

n_0 is the threshold

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



Big O

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

  $f(n) = O(g(n))$
- Correct notation

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

Big O

- Question:

  How do you demonstrate that f(n) $\in$ O(g(n))?

- Answer:

  Show that you can find values for c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$

Big O

Example: Show that $7n - 2$ is $O(n)$.

Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $7n - 2 \leq cn$ for every integer $n \geq n_0$.

Choose $c = 7$ and $n_0 = 1$.

It is easy to see that $7n - 2 \leq 7n$ for every integer $n \geq 1$.

$\therefore$ $7n - 2$ is $O(n)$

Big O

Example: Show that $20n^3 + 10n \log n + 5$ is $O(n^3)$.

Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $20n^3 + 10n \log n + 5 \leq cn^3$ for every integer $n \geq n_0$.

How do we find c and n_0?

Note that $10n^3 > 10\, n \log n$, and that $5n^3 > 5$.

So, $15n^3 > 10n \log n + 5$

And $20n^3 + 15n^3 > 20n^3 + 10n \log n + 5$

Therefore, $35n^3 > 20n^3 + 10n \log n + 5$

Big O

So we choose c = 35 and $n_0 = 1$.

An algorithm that takes $20n^3 + 10n \log n + 5$ steps to run can't possibly take any more than $35n^3$ steps, for every integer $n \geq 1$.

Therefore $20n^3 + 10n \log n + 5$ is $O(n^3)$.

Big O

Example: Show that $\frac{1}{2} n^2 - 3n$ is $O(n^2)$

Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\frac{1}{2} n^2 - 3n \leq cn^2$ for every integer $n \geq n_0$.

Choose $c = \frac{1}{2}$ and $n_0 = 1$.

Now $\frac{1}{2} n^2 - 3n \leq \frac{1}{2} n^2$ for every integer $n \geq 1$.

Big O

Example: Show that $an(\log_2 n) + bn$ is $O(n \cdot \log n)$

Find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$an(\log_2 n) + bn \leq cn \cdot \log n$$

for every integer $n \geq n_0$.

Choose $c = a+b$ and $n_0 = 2$ (why 2?).

Now $an(\log_2 n) + bn \leq cn \cdot \log n$ for every integer $n \geq 2$.

Big O

- Question:

  Is $n = O(n^2)$?

- Answer:

  Yes. Remember that $f(n) \in O(g(n))$ if there exist positive constants c and n_0 such that

  $$0 \leq f(n) \leq c \bullet g(n) \text{ for all } n \geq n_0 \}$$

  If we set $c = 1$ and $n_0 = 1$, then it is obvious that $c \bullet n \leq n^2$ for all $n \geq n_0$.

Big O

- What does this mean about Big-O?
- When we write f(n) = O(g(n)) we mean that some constant times g(n) is an asymptotic upper bound on f(n); we are not claiming that this is a *tight* upper bound.

Big O

- Big-O notation describes an upper bound
- Assume we use Big-O notation to bound the *worst case* running time of an algorithm
- Now we have a bound on the running time of the algorithm on *every input*

Big O

- Is it correct to say "the running time of insertion sort is $O(n^2)$"?
- Technically, the running time of insertion sort depends on the characteristics of its input. If we have *n* items in our list, but they are already in sorted order, then the running time of insertion sort *on this particular input* is O(n).

Big O

- So what do we mean when we say that the running time of insertion sort is $O(n^2)$?
- What we normally mean is:

  the *worst case* running time of insertion sort is $O(n^2)$
- That is, if we say that "the running time of insertion sort is $O(n^2)$", we guarantee that under no circumstances will insertion sort perform worse than $O(n^2)$.

Big Omega

Definition:

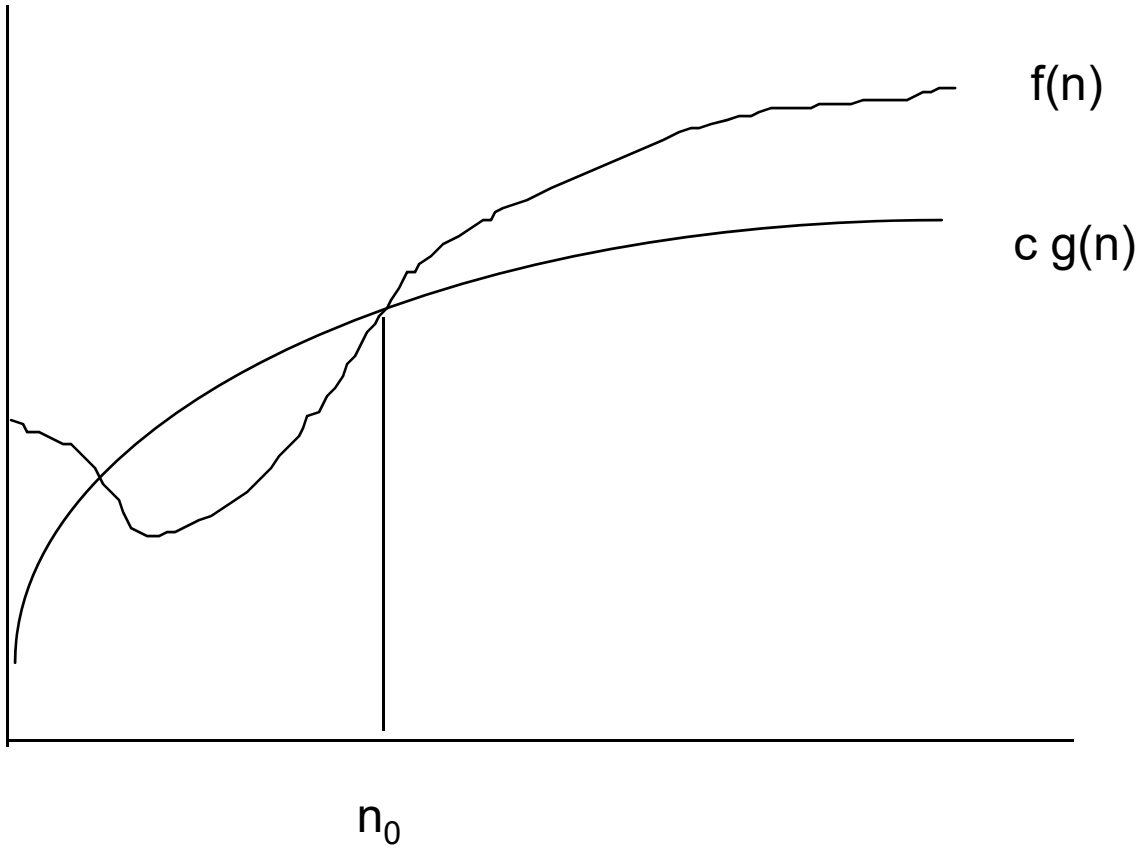
For a given function g(n), $\Omega(g(n))$ is the set of functions:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$$0 \leq c\, g(n) \leq f(n)$$

for all $n \geq n_0 \}$

$f(n) \in \Omega(g(n))$



Big Omega

- We know that Big-O notation provides an asymptotic upper bound on a function.
- Big-Omega notation provides an *asymptotic lower bound* on a function.
- Basically, if we say that $f(n) = \Omega(g(n))$ then we are guaranteeing that, beyond n_0, f(n) never performs any better than c g(n).

Big Omega

- We usually use Big-Omega when we are talking about the *best case* performance of an algorithm.
- For example, the best case running time of insertion sort (on an already sorted list) is $\Omega(n)$.
- But this also means that insertion sort never performs any better than $\Omega(n)$ on any type of input.
- So the running time of insertion sort is $\Omega(n)$.

Big Omega

- Could we say that the running time of insertion sort is $\Omega(n^2)$?
- No. We know that if its input is already sorted, the curve for merge sort will dip below n^2 and approach the curve for n.
- Could we say that the *worst case* running time of insertion sort is $\Omega(n^2)$?
- Yes.

Big Omega

- It is interesting to note that, for any two functions f(n) and g(n), $f(n) = \Theta(g(n))$ if and only if f(n) = O(g(n)) and $f(n) = \Omega(g(n))$.

Big Theta

- Definition

  For a given function g(n), $\Theta(g(n))$ is the set of functions:

  $\Theta(g(n)) = \{f(n)$: there exist positive constants

  c_1, c_2 and n_0 such that

  $$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

  for all $n \geq n_0$ $\}$

Big Theta

- What does this mean?
- When we use Big-Theta notation, we are saying that function *f(n)* can be "sandwiched" between some *small* constant times *g(n*) and some *larger* constant times *g(n)*.
- In other words, *f(n)* is equal to *g(n)* to within a constant factor.

$$f(n) \in \Theta(g(n))$$

n

Big Theta

- If $f(n) = \Theta(g(n))$, we can say that $g(n)$ is an *asymptotically tight bound* for $f(n)$.
- Basically, we are guaranteeing that f(n) never performs any better than c_1 g(n), but also never performs any worse than c_2 g(n).
- We can see this visually by noting that, after n_0, the curve for f(n) never goes below c_1 g(n) and never goes above c_2 g(n).

Big Theta

- Let's look at the performance of the merge sort.
- We said that the performance of merge sort was
  $cn(\log_2 n) + cn$
- Does this depend upon the characteristics of the input for merge sort? That is, does it make a difference if the list is already sorted, or reverse sorted, or in random order?
- No. Unlike insertion sort, merge sort behaves exactly the same way for any type of input.

Big Theta

- The running time of merge sort is:

  $cn(\log_2 n) + cn$

- So, using asymptotic notation, we can discard the "+ cn" part of this equation, giving:

  $cn(\log_2 n)$

- And we can disregard the constant multiplier, c, which gives us the running time of merge sort:

  $\Theta(n(\log_2 n))$

Big Theta

- Why would we prefer to express the running time of merge sort as $\Theta(n(\log_2 n))$ instead of $O(n(\log_2 n))$?
- Because Big-Theta is more precise than Big-O.
- If we say that the running time of merge sort is $O(n(\log_2 n))$, we are merely making a claim about merge sort's asymptotic upper bound, whereas of we say that the running time of merge sort is $\Theta(n(\log_2 n))$, we are making a claim about merge sort's asymptotic upper *and lower* bounds.

Big Theta

- Would it be incorrect to say that the running time of merge sort is $O(n(\log_2 n))$?
- No, not at all.
- It is just that we are not giving all of the information that we have about the running time of merge sort.
- But sometimes all we need to know is the worst-case behavior of an algorithm. If that is so, then Big-O notation is fine.

Big Theta

- One final note: the definition of $\Theta(g(n))$ technically requires that every member f(n) $\in$ $\Theta(g(n))$ be asymptotically nonnegative – that is, f(n) must be nonnegative whenever n is sufficiently large.
- We assume that every function used within Θ notation (and the other notations used in your textbook's Chapter 3) is asymptotically nonnegative

Little o

Definition:

For a given function g(n), o(g(n)) is the set of functions:

o(g(n))= {f(n): for any positive constant c,

there exists a constant n_0 such that

$$0 \leq f(n) < c\, g(n)$$

for all $n \geq n_0$ }

Little o

- Note the $<$ instead of $\leq$ in the definition of Little-o:

  $0 \leq f(n) < c\, g(n)$ for all $n \geq n_0$
- Contrast this to the definition used for Big-O:

  $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$
- Little-o notation denotes an *upper bound that is not asymptotically tight*. We might call this a *loose* upper bound.
- Examples:

  $2n \in o(n^2)$ but $2n^2 \notin o(n^2)$

Little o

Given that $f(n) = o(g(n))$, we know that g grows *strictly faster* than f. This means that you can multiply g by a positive constant c and beyond n_0 g will always exceed f.

I couldn't find a graph to demonstrate little-o, but here is an example:

$n^2 = o(n^3)$ but

$n^2 \neq o(n^2)$.

Why? Because if c = 1, then $f(n) = c\ g(n)$, and the definition insists that $f(n)$ be <u>less than</u> $c\ g(n)$.

Little-omega

- Definition:

  For a given function g(n), ω(g(n)) is the set of functions:

  ω(g(n))= {f(n): for any positive constant *c*,

  there exists a constant n_0 such that

  $$0 \leq c\, g(n) < f(n)$$

  for all $n \geq n_0$ }

Little-omega

- Note the < instead of ≤ in the definition:

  $0 \leq c\, g(n) < f(n)$

- Contrast this to the definition used for Big-Ω:

  $0 \leq c\, g(n) \leq f(n)$

- Little-omega notation denotes a *lower bound that is not asymptotically tight.* We might call this a *loose* lower bound.
- Examples:

  $n \notin \omega(n^2)$ $\qquad\qquad$ $n \in \omega(\lg n)$

Little-omega

I couldn't find a graph to demonstrate little-omega, but here is an example:

n^3 is $\omega(n^2)$ but

$n^3 \neq \omega(n^3)$.

Why? Because if c = 1, then $f(n) = c\ g(n)$, and the definition insists that $c\ g(n)$ be strictly <u>less than</u> $f(n)$.

Comparison of Notations

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \omega(g(n)) \approx a > b$$

Asymptotic notation

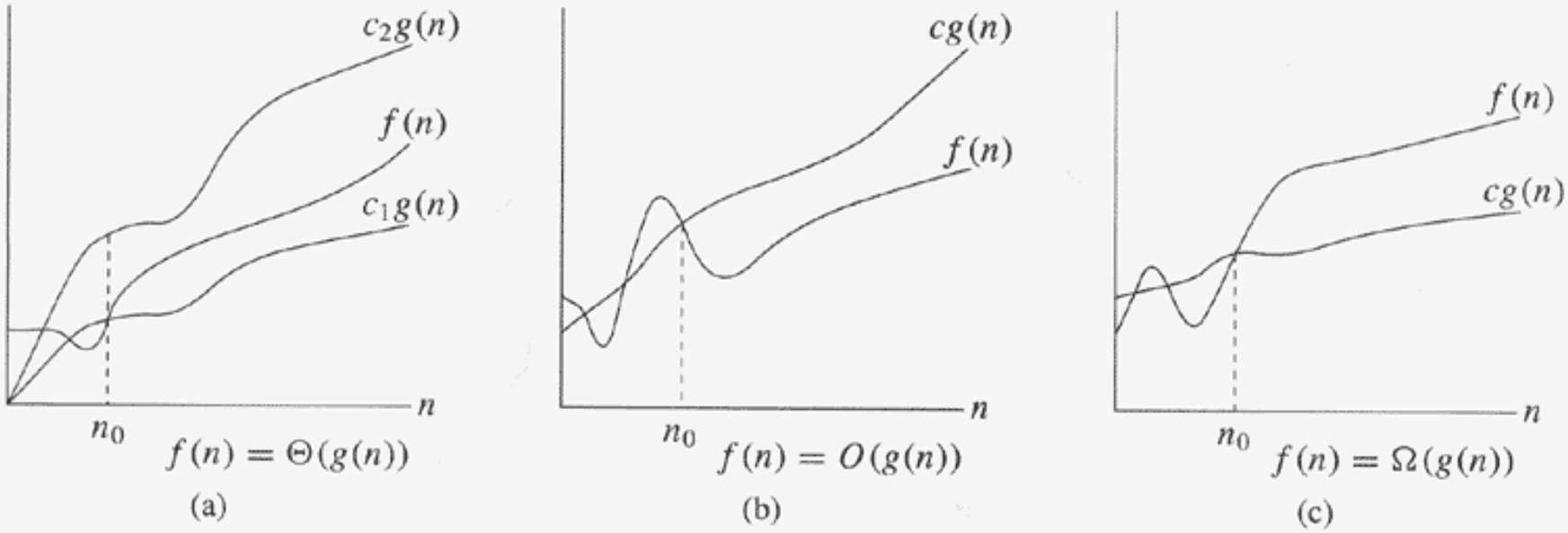


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

Asymptotic Notation in Equations and Inequalities

- When asymptotic notation stands alone on right-hand side of equation, '=' is used to mean '$\in$'.
- In general, we interpret asymptotic notation as standing for some anonymous function we do not care to name.

  Example: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that

  $2n^2 + 3n + 1 = 2n^2 + f(n)$ for some $f(n) \in \Theta(n)$.

  (In this case, $f(n) = 3n + 1$, which is in $\Theta(n)$.)

Asymptotic Notation in Equations and Inequalities

- This use of asymptotic notation eliminates inessential detail in an equation (e.g., we don't have to specify lower-order terms; they are understood to be included in anonymous function).
- The number of anonymous functions in an expression is the number of times asymptotic notation appears.

  E.g., $\Sigma O(i)$ is not the same as $O(1)+(2)+\ldots+O(n)$.

Asymptotic Notation in Equations and Inequalities

- Appearance of asymptotic notation on left-hand side of equation means, no matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Example: $2n^2 + \Theta(n) = \Theta(n^2)$ means that for any function $f(n) \in \Theta(n)$

there is some function

$g(n) \in \Theta(n^2)$

such that $2n^2 + f(n) = g(n)$ for all n.

Comparison of Functions

- Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ and $g(n) = o(h(n))$ imply $f(n) = o(h(n))$

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$ imply $f(n) = \omega(h(n))$

Comparison of Functions

- Reflexivity:

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

Comparison of Functions

- Symmetry:

$f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Comparison of Functions

- Transpose symmetry:

$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Comparison of Functions

- Analogies:

$f(n) = o(g(n)) \approx a < b$

$f(n) = O(g(n)) \approx a \leq b$

$f(n) = \Theta(g(n)) \approx a = b$

$f(n) = \Omega(g(n)) \approx a \geq b$

$f(n) = \omega(g(n)) \approx a > b$

Comparison of Functions

- Asymptotic relationships:

  f(n) is asymptotically smaller than g(n) if

  $f(n) = o(g(n))$

  f(n) is asymptotically larger than g(n) if

  $f(n) = \omega(g(n))$

Comparison of Functions

- Asymptotic relationships:

  Not all functions are asymptotically comparable. That is, it may be the case that neither $f(n) = o(g(n))$ nor $f(n) = \omega(g(n))$ is true.

Standard Notation

- Pages 51 – 56 contain review material from your previous math courses. Please read this section of your textbook and refresh your memory of these mathematical concepts.
- The remaining slides in this section are for your aid in reviewing the material; we will not go over them in class.

Monotonicity

- A function f(n) is *monotonically increasing* if $m \leq n$ implies $f(m) \leq f(n)$.
- A function f(n) is *monotonically decreasing* if $m \leq n$ implies $f(m) \geq f(n)$.
- A function f(n) is *strictly increasing* if $m < n$ implies $f(m) < f(n)$.
- A function f(n) is *strictly decreasing* if $m < n$ implies $f(m) > f(n)$.

Floor and ceiling

- For any real number x, the floor of x is the greatest integer less than or equal to x.

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing.

- For any real number x, the ceiling of x is the least integer greater than or equal to x.

The ceiling function $f(x) = \lceil x \rceil$ is monotonically increasing.

Modulo arithmetic

- For any integer *a* and any positive integer *n*, the value of *a modulo n* (or *a mod n*) is the remainder we have after dividing *a* by *n*.
- a mod n = a - $\lfloor a/n \rfloor$ n
- if (a mod n) = (b mod n), then a ≡ b mod n (read as “a is equivalent to b mod n”)

Polynomials

- Given a nonnegative integer d, a polynomial in n of degree d is a function p(n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the constants a_0, a_1, ..., a_d are the coefficients of the polynomial and $a_d \neq 0$.

Polynomials

- A polynomial is asymptotically positive if and only if $a_d > 0$.
- If a polynomial $p(n)$ of degree d is asymptotically positive, then $p(n) = \Theta(n^d)$.
- For any real constant $a \geq 0$, n^a is monotonically increasing.
- For any real constant $a \leq 0$, n^a is monotonically decreasing.
- A function is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

Exponentials

- For all n and $a \geq 1$, the function a^n is monotonically increasing in n.
- For all real constants a and b such that $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0$$

This means that $n^b = o(a^n)$, which means that any *exponential* function with a base strictly greater than 1 grows *faster* than any *polynomial* function.

Logarithms

- $\lg n = \log_2 n$ (binary logarithm)
- $\ln n = \log_e n$ (natural logarithm)
- $\lg^k n = (\lg n)^k$ (exponentiation)
- $\lg \lg n = \lg (\lg n)$ (composition)
- $\lg n + k$ means $(\lg n) + k$, not $\log (n + k)$
- If $b > 1$ and we hold b constant, then, for $n > 0$, the function $\log_b n$ is strictly increasing.
- Changing the base of a logarithm from one constant to another only changes the value of the logarithm by a constant factor.

Logarithms

- A function is *polyalgorithmically bounded* if $f(n) = O(\lg^k n)$ for some constant k.
- $\lg^b n = o(n^a)$ for any constant $a > 0$
- This means that any positive polynomial function grows faster than any polyalgorithmic function.

Factorials

- N factorial is defined for integers ≥ 0 as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0 \end{cases}$$

A weak upper bound on n! is $n! \leq n^n$

$n! = o(n^n)$

$n! = \omega(2^n)$

$\lg(n!) = \Theta(n \lg n)$

Fibonacci numbers

- The Fibonacci numbers are defined by the recurrence:

  $F_0 = 0$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2} \geq 2$

- Fibonacci numbers grow exponentially

Conclusion

- Asymptotic notation
- Comparison of functions
- Standard notations and common functions