CS1382 Discrete Computational Structures

Lecture 07: The Growth of Functions

Spring 2019

Richard Matovu

TEXAS TECH UNIVERSITY

The Growth of Functions

- In both Computer Science and in Mathematics, there are many times when we care about how fast a function grows.
    - Understand how quickly an algorithm can solve a problem as the size of the input grows.
        - Compare the efficiency of two different algorithms for solving the same problem.
        - Determine whether it is practical to use a particular algorithm as the input grows.

Big-*O* Notation

- Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers.

  We say that *f*(*x*) is *O* (*g*(*x*)) if there are constants *C* and *k* such that

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

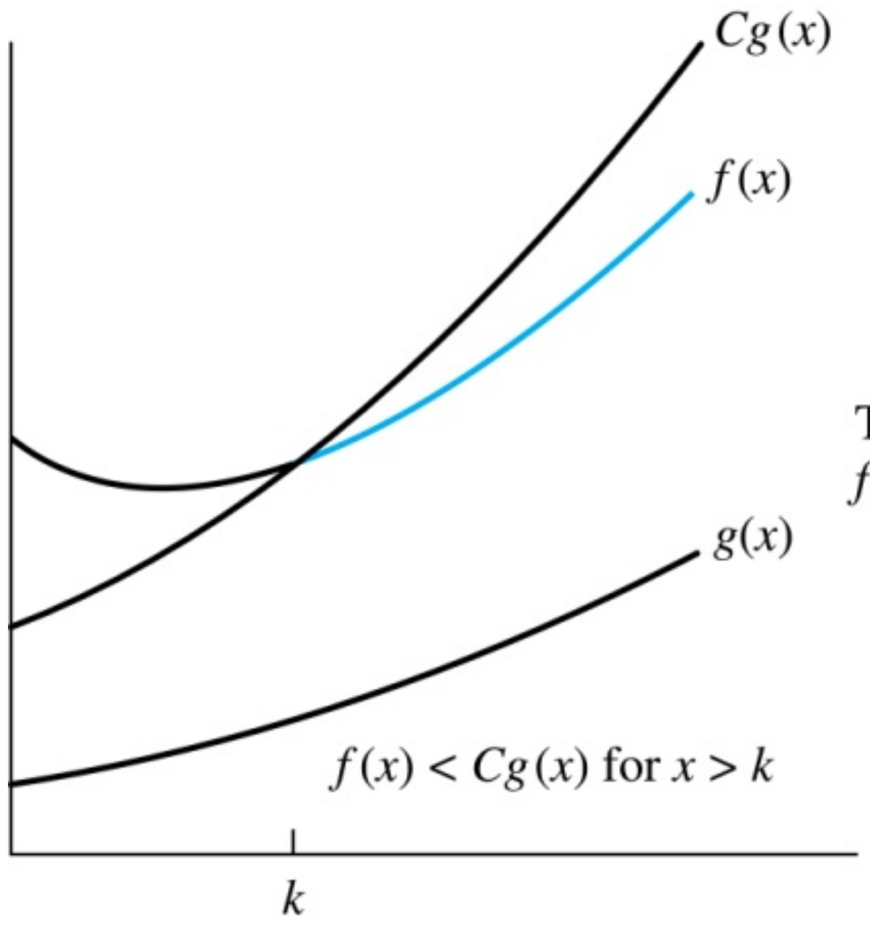
  whenever $x > k$. (illustration on next slide)

- This is read as "*f*(*x*) is big-*O* of *g*(*x*)" or "*g* asymptotically dominates *f*."

- The constants C and k are called ***witnesses*** to the relationship *f*(*x*) is *O*(*g*(*x*)).

- Only one pair of witnesses is needed.

Illustration of Big-*O* Notation



$f(x)$ is $O\,(g(x)\,)$

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

Some Important Points about Big-*O* Notation

- If one pair of witnesses is found, then there are infinitely many pairs.

  We can always make the *k* or the *C* larger and still maintain the inequality

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

  - Any pair *C'* and *k'* where *C* < C ' and *k* < k' is also a pair of witnesses since

    $|f(x)| \leq C|g(x) \leq C'|g(x)|$ whenever $x > k' > k$.

- You may see " $f(x) = O(g(x))$" instead of " $f(x)$ is $O(g(x))$."

- Usually, we will drop the absolute value sign

Example - Big-*O* Notation

1. Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$

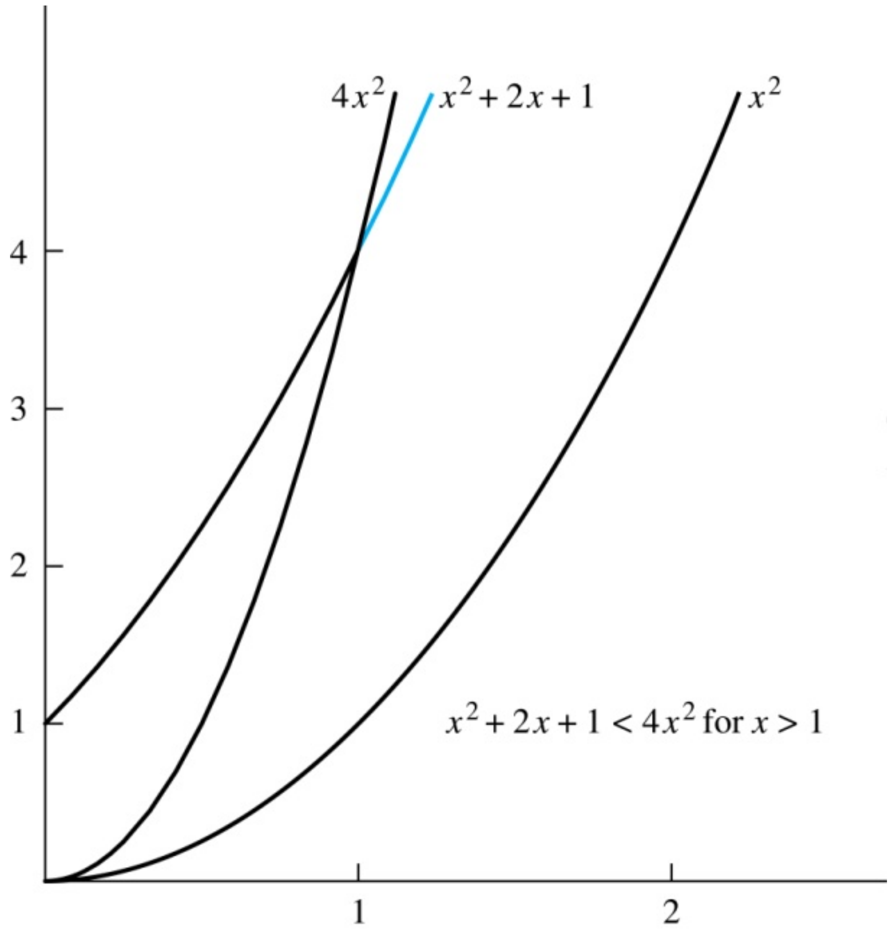
   Solution:

   $x^2 + 2x + 1 = O(x^2)$

   $x^2 + 2x + 1 \leq C x^2$

   Pick x = 1, then C = 4, and therefore I can take *C* = 4 and *k* = 1 as witnesses *(see graph)*

   Alternatively, Pick k = 2, C = ?



Big-O Notation - Examples

1. Show that $7x^2$ is $O(x^3)$.

- **Solution**:

  When $x > 7$, $7x^2 < x^3$. Take $C = 1$ and $k = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.

(Would $C = 7$ and $k = 1$ work?)

2. Show that n^2 is not $O(n)$.

- **Solution**:

  Suppose there are constants C and k for which $n^2 \leq Cn$, whenever $n > k$. Then (by dividing both sides of $n^2 \leq Cn$) by n, then $n \leq C$ must hold for all $n > k$. A contradiction!

Big-O Estimates for Polynomials

Example: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_o$

Where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is $O(x^n)$.

Proof: $|f(x)| = |a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0|$

$\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \cdots + |a_1| x^1 + |a_0|$

Assuming $x > 1$

$= x^n (|a_n| + |a_{n-1}| / x + \cdots + |a_1| / x^{n-1} + |a_0| / x^n)$

$\leq x^n (|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|)$

Uses triangle inequality, an exercise in Section 1.8.

Take $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$. Then $f(x)$ is $O(x^n)$.

The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-*O* Estimates for some Important Functions

- Use big-*O* notation to estimate the sum of the first *n* positive integers.

- **Solution**: $1 + 2 + \cdots + n \;\leq\; n + n + \cdots n \;=\; n^2$

  $1 + 2 + \ldots + n$ is $O(n^2)$ taking $C = 1$ and $k = 1$.

- Use big-*O* notation to estimate the factorial function

- **Solution**: $f(n) = n! = 1 \times 2 \times \cdots \times n\ .$

  $n! = 1 \times 2 \times \cdots \times n \;\leq\; n \times n \times \cdots \times n \;=\; n^n$

  $n!$ is $O(n^n)$ taking $C = 1$ and $k = 1$.

Continued →

Big-*O* Estimates for some Important Functions

- Use big-*O* notation to estimate log *n* !

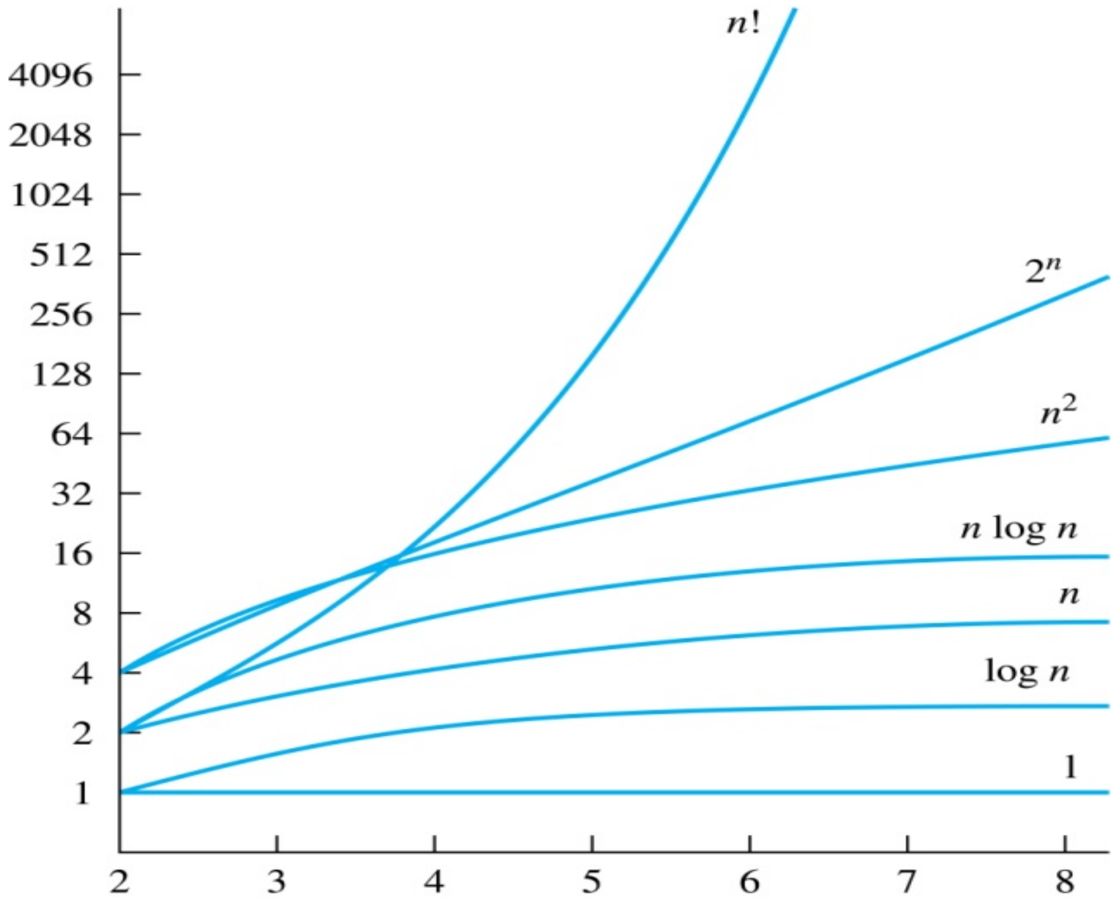
- **Solution**:

  Given that $n! \leq n^n$ (previous slide)

  then $\log(n!) \leq n \cdot \log(n)$.

  Hence, log(*n*!) is *O* (*n* · log(*n*)) taking *C* = 1 and *k* = 1.

Display of Growth of Functions



Note the difference in behavior of functions as *n* gets larger

Combinations of Functions

- 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 $O(\max(|g_1(x)|,|g_2(x)|))$.

- If $f_1(x)$ and $f_2(x)$ are both $O(g(x))$ then

  $(f_1 + f_2)(x)$ is $O(g(x))$.

- 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 $O(g_1(x)g_2(x))$.

Ordering Functions by Order of Growth

Put the functions below in order so that each function is big-O of the next function on the list.

- $f_1(n) = (1.5)^n$
- $f_2(n) = 8n^3+17n^2 +111$
- $f_3(n) = (\log n)^2$
- $f_4(n) = 2^n$
- $f_5(n) = \log (\log n)$
- $f_6(n) = n^2 (\log n)^3$
- $f_7(n) = 2^n (n^2 +1)$
- $f_8(n) = n^3+ n(\log n)^2$
- $f_9(n) = 10000$
- $f_{10}(n) = n!$

We solve this exercise by successively finding the function that grows slowest among all those left on the list.

- $f_9(n) = 10000$ (constant, does not increase with n)
- $f_5(n) = \log (\log n)$ (grows slowest of all the others)
- $f_3(n) = (\log n)^2$ (grows next slowest)
- $f_6(n) = n^2 (\log n)^3$ (next largest, $(\log n)^3$ factor smaller than any power of n)
- $f_2(n) = 8n^3+17n^2 +111$ (tied with the one below)
- $f_8(n) = n^3+ n(\log n)^2$ (tied with the one above)
- $f_1(n) = (1.5)^n$ (next largest, an exponential function)
- $f_4(n) = 2^n$ (grows faster than one above since $2 > 1.5$)
- $f_7(n) = 2^n (n^2 +1)$ (grows faster than above because of the $n^2 +1$ factor)
- $f_{10}(n) = n!$ ($n!$ grows faster than c^n for every c)

Big-Omega Notation

Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers.

We say that $f(x)$ is $\Omega(g(x))$ if there are constants *C* and *k* such that

$$|f(x)| \geq C|g(x)|$$ when $x > k$.

Ω is the upper case version of the lower case Greek letter ω.

- We say that "*f*(*x*) is big-Omega of *g*(*x*)."
- Big-*O* gives an upper bound on the growth of a function, while Big-Omega gives a lower bound.
- Big-Omega tells us that a function grows at least as fast as another.
- *f*(*x*) is Ω(*g*(*x*)) if and only if *g*(*x*) is *O*(*f*(*x*)). This follows from the definitions.

Big-Omega Notation - Example

- Show that $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$ where $g(x) = x^3$.

- **Solution**:
  $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers *x*.

  - Is it also the case that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$?

Big-Theta Notation

Θ is the upper case version of the lower case Greek letter θ.

Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. The function $f(x)$ is $\Theta(g(x))$ if

$f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

- We say that "f is big-Theta of *g*(*x*)" and also that "*f*(*x*) is of *order g*(*x*)" and also that "*f*(*x*) and *g*(*x*) are of the *same order*."
- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1, C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$.
- This follows from the definitions of big-*O* and big-Omega.

Big Theta Notation - Example

- Show that the sum of the first n positive integers is $\Theta(n^2)$.
- **Solution**: Let $f(n) = 1 + 2 + \cdots + n$.
  - We have already shown that $f(n)$ is $O(n^2)$.
  - To show that $f(n)$ is $\Omega(n^2)$, we need a positive constant C such that $f(n) > Cn^2$ for sufficiently large n.
    - Taking $C = ¼$, $f(n) > Cn^2$ for all positive integers n.
  - Hence, $f(n)$ is $\Omega(n^2)$, and we can conclude that $f(n)$ is $\Theta(n^2)$.

Big-Theta Notation - Example

- Show that $f(x) = 3x^2 + 8x \log x$ is $\Theta(x^2)$.
- **Solution**:
  - $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$
    - Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
  - $3x^2 + 8x \log x \geq 3x^2$
    - Hence, $3x^2 + 8x \log x$ is $\Omega(x^2)$.
  - Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$ for $c_1 = 3$ and $c_2 = 11$.

Big-Theta Notation

- When $f(x)$ is $\Theta(g(x))$ it must also be the case that $g(x)$ is $\Theta(f(x))$.
- Note that $f(x)$ is $\Theta(g(x))$ if and only if it is the case that

  $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

- Sometimes writers are careless and write as if big-*O* notation has the same meaning as big-Theta.

Big-Theta Estimates for Polynomials

Theorem:

Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_o$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is of order x^n (or $\Theta(x^n)$).

Example:

- The polynomial $f(x) = 8x^5 + 5x^2 + 10$ is order of x^5 (or $\Theta(x^5)$).

- The polynomial $f(x) = 8x^{199} + 7x^{100} + x^{99} + 5x^2 + 25$ is order of x^{199} (or $\Theta(x^{199})$).

CS1382 Discrete Computational Structures

Complexity of Algorithms

Spring 2019

Richard Matovu

TEXAS TECH UNIVERSITY

The Complexity of Algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size?

  To answer this question, we ask:

  - How much time does this algorithm use to solve a problem?
  - How much computer memory does this algorithm use to solve a problem?

- **Time complexity of an algorithm**

  - Analyzing the time the algorithm uses to solve the problem given input of a particular size.

- **Space Complexity of algorithm**

  - Analyzing the computer memory the algorithm uses to solve the problem given input of a particular size

Time Complexity

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.). We can estimate the time a computer may actually use to solve a problem using the amount of time required to do basic operations.
- We ignore minor details, such as the “house keeping” aspects of the algorithm.
- We will focus on the ***worst-case time*** complexity of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the ***average case time complexity*** of an algorithm. This is the average number of operations an algorithm uses to solve a problem over all inputs of a particular size.

Complexity Analysis of Algorithms

a_1 := 1000
$a_2 := a_1 ** 2$

O(*1*)

for i := 1 to n
    a_i := 0

O(*n*)

for i := 1 to n
    for j := 1 to n
        a_i := 0

O(n^2)

if a > 0
    a_i := 0
else
    for i := 1 to n
        for j :=1 to n
            a_i := 0

O(n^2)

Complexity Analysis of Algorithms

Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

```
procedure max(a_1, a_2, …., a_n: integers)
max := a_1
for i := 2 to n
    if max < a_i then max := a_i
return max {max is the largest element}
```

- **Solution**:

  The time complexity of the algorithm is $O(n)$.

Worst-Case Complexity of Linear Search

Determine the time complexity of the linear search algorithm.

procedure *linearsearch*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location* {*location* is the subscript of the term that equals x, or is 0 if x is not found}

- **Solution**:

  The time complexity of the algorithm is $O(n)$.

Worst-Case Complexity of Binary Search

Describe the time complexity of binary search.

```
procedure binarysearch(x: integer, a1,a2,…, an: increasing integers)
i := 1 {i is the left endpoint of interval}
j := n {j is right endpoint of interval}
while i < j
      m := ⌊(i + j)/2⌋
      if x > am then i := m + 1
      else j := m
 if x = ai then location := i
 else location := 0
 return location {location is the subscript i of the term ai  equal to x, or 0 if x is not found}
```

- **Solution**:

  The time complexity of the algorithm is O(*lgn*) better than linear search.

Worst-Case Complexity of Bubble Sort

Describe the time complexity of bubble sort.

procedure *bubblesort* ($a_1,\ldots,a_n$: real numbers with $n \geq 2$)

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}

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

- **Solution**:

  The time complexity of the algorithm is $O(n^2)$.

Worst-Case Complexity of Insertion Sort

Describe the time complexity of insertion sort

```
procedure insertionsort (a_1,…,a_n: real numbers with n ≥ 2)
  for j := 2 to n
      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

{Now a_1,…,a_n is in increasing order}
```

- **Solution**:

  The time complexity of the algorithm is O(n^2)

Questions?

Thank You!