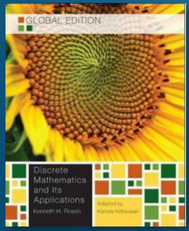
Algorithms

Chapter 3. Asymptotic Computational Complexity



- Taken from the instructor's resource of *Discrete Mathematics and Its Applications*, 7/e
- Edited by Shin Hong hongshin@handong.edu

The Growth of Functions

- In both computer science and in mathematics, there are many many times when we care about how fast a function grows.

- In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.
  - We can compare the efficiency of two different algorithms for solving the same problem.
  - We can also determine whether it is practical to use a particular algorithm as the input grows.
  - We'll study these questions in Section 3.3.

Big-*O* Notation (1/3)

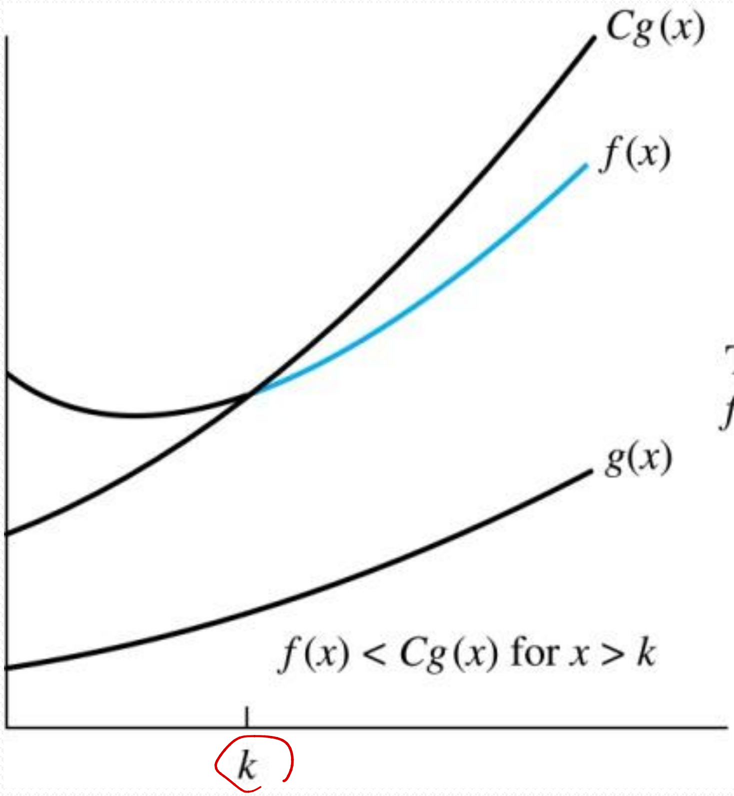
- Let f and g be functions from the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

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

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.

Big-O Notation (2/3)



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

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

Big-*O* Notation (3/3)

C와 k 의미

- 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 $O(g(x))$."
  - But this is an abuse of the equals sign since the meaning is that there is an inequality relating the values of f and g, for sufficiently large values of x.
  - It is ok to write $f(x) \in O(g(x))$, because $O(g(x))$ represents the the set of functions that are $O(g(x))$.
- Usually, we will drop the absolute value sign since we will always deal with functions that take on positive values.

Using the Definition of Big-*O* Notation

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

Solution: Since when $x > 1$, $x < x^2$ and $1 < x^2$

$$0 \le x^2 + 2x + 1 \le x^2 + 2x^2 + x^2 = 4x^2$$

- Can take $C = 4$ and $k = 1$ as witnesses to show that

  $f(x)$ is $O(x^2)$ (see graph on next slide)

- Alternatively, when $x > 2$, we have $2x \le x^2$ and $1 < x^2$. Hence, $0 \le x^2 + 2x + 1 \le x^2 + x^2 + x^2 = 3x^2$ when $x > 2$.
  - Can take $C = 3$ and $k = 2$ as witnesses instead.

$\boxed{x>1}$ k

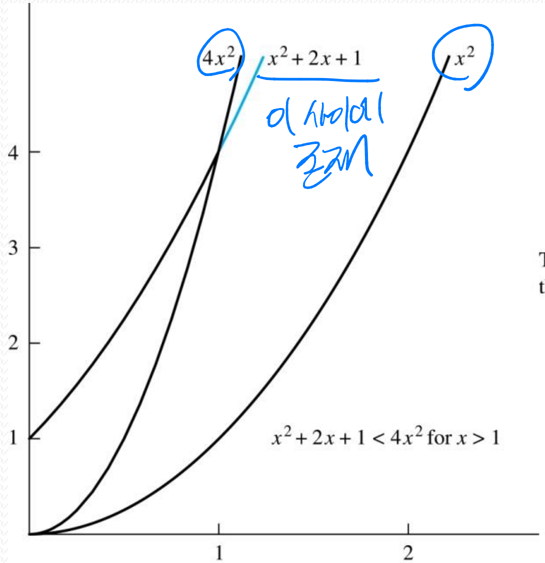
$x^2+2x+1 \le x^2+2x^2+x^2$

$\le 4x^2$ c

$\therefore f(x) = O(x^2)$

Illustration of Big-*O* Notation

$$f(x) = x^2 + 2x + 1 \text{ is } O(x^2)$$



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

Big-*O* Notation

- Both $f(x) = x^2 + 2x + 1$ and $g(x) = x^2$ are such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
  We say that the two functions are of the *same order*. (More on this later)

- If $f(x)$ is $O(g(x))$ and *h(x)* is larger than *g(x)* for all positive real numbers, then $f(x)$ is $O(h(x))$.

- Note that if $|f(x)| \leq C|g(x)|$ for *x* > *k* and if $|h(x)| > |g(x)|$ trivial하다.
  for all *x*, then $|f(x)| \leq C|h(x)|$ if *x* > *k*. Hence, $f(x)$ is $O(h(x))$.

- For many applications, the goal is to select the function *g*(*x*) in *O(g(x))* as small as possible (up to multiplication by a constant, of course).

Using the Definition of Big-O Notation

Example: 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?)

Example: 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!

$n^2 - Cn \leq 0$ whenever $n > k$

$n(n-C) \leq 0$ ~ $n - C \leq 0$

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 $O(x^n)$.

Uses triangle inequality, an exercise in Section 1.8.

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

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

Assuming $x > 1$

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

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

- Take $C = |a_n| + |a_{n-1}| + \cdots + |a_1| + |a_1|$ and $k = 1$. Then $f(x)$ is $O(x^n)$.
- The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-Theta Notation

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

- **Definition**: 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 $\Omega(g(x))$. → $|f(x)| \geq C|g(x)|$ whenever $x > k$ 이 C와 k가 존재,
- 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.

→ $\Omega(g(x))$ → $O(g(x))$

→ 샌드위치 정리

Big Theta Notation

Example: Show that the sum of the first n positive integers is $\Theta(n^2)$.

Solution: Let $f(n) = 1 + 2 + \cdots + n$.

$Cn^2 \leq f(n) \leq O(n^2)$

$\Rightarrow f(n) = \Theta(n^2)$

- 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. Summing only the terms greater than $n/2$ we obtain the inequality

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\ &= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (n/2)(n/2) = n^2/4 \end{aligned}$$

- Taking $C = 1/4$, $f(n) > Cn^2$ 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:

- $3x^2 + 8x \log x \le 11x^2$ for $x > 1$, since $0 \le 8x \log x \le 8x^2$.
  - Hence, $3x^2 + 8x \log x$ is $O(x^2)$.
- x^2 is clearly $O(3x^2 + 8x \log x)$ $x>0$
- Hence, $3x^2 + 8x \log x$ is $\Theta(x^2)$.

$x^2 \le 3x^2+8x\log x \le 8x^2$

$\Omega(x^2) \le 3x^2+8x\log x \le O(x^2) \rightarrow 3x^2+8x\log x = \Theta(x^2)$

The Complexity of Algorithms (1/2)

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size?
  - Time complexity - How much time does this algorithm use to solve a problem?
  - Space complexity - How much computer memory does this algorithm use to solve a problem?

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.)

- 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.

The Complexity of Algorithms (2/2)

- We will measure time complexity in terms of the number of operations an algorithm uses and we will use big-O and big-Theta notation to estimate the time complexity.

- We can use this analysis to see whether it is practical to use this algorithm to solve problems with input of a particular size. We can also compare the efficiency of different algorithms for solving the same problem.

- We ignore implementation details (including the data structures used and both the hardware and software platforms) because it is extremely complicated to consider them.

Complexity Analysis of Algorithms

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

procedure *max*($a_1, a_2, \ldots., a_n$: integers)
  $max := a_1$ a_2 a_3 a_n
  for $i := 2$ to n
    if $max < a_i$ then $max := a_i$
  return *max* {*max* is the largest element}

Solution: Count the number of comparisons.

- The $max < a_i$ comparison is made $n - 2$ times. (?)
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.

Hence, the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Linear Search

Ex. Determine the time complexity of the linear search algorithm.

procedure *linear search*(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*

Solution: Count the number of comparisons.

- At each step two comparisons are made; $i \leq n$ and $x \neq a_i$.
- To end the loop, one comparison $i \leq n$ is made.
- After the loop, one more $i \leq n$ comparison is made.

If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$.