The Growth of Functions

Section 3.2

Section Summary

- Big-O Notation
- Big-O Estimates for Important Functions
- Big-Omega and Big-Theta Notation

The Growth of Functions

- In both computer science and in mathematics, there are 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. (An Example in the next slide).
    - 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.
  - Two of the areas of mathematics where questions about the growth of functions are studied are:
    - number theory (covered in Chapter 4)
    - combinatorics (covered in Chapters 6 and 8)

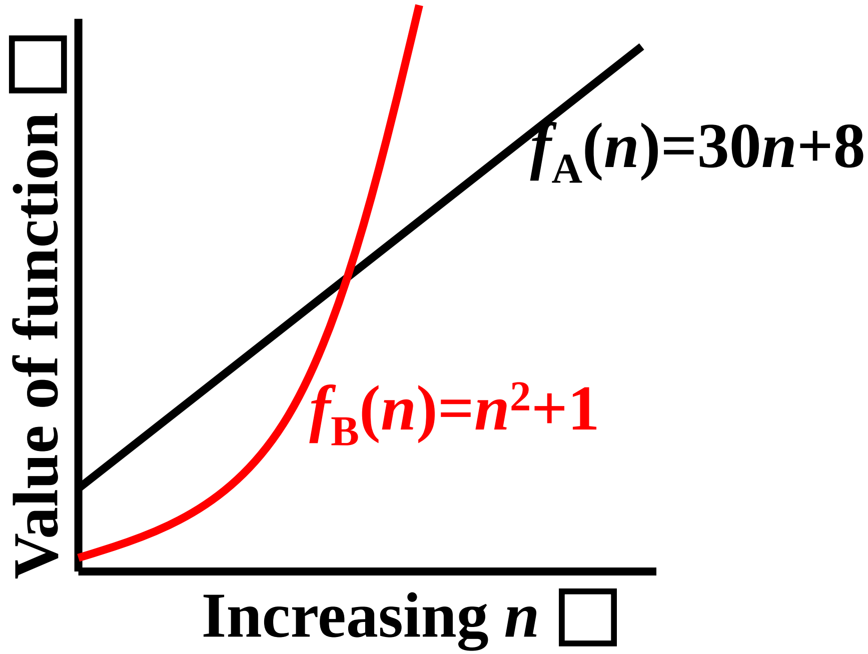
Example of Orders of Growth

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...

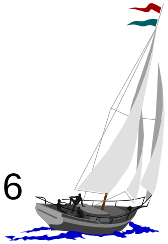
Concept of order of growth

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.

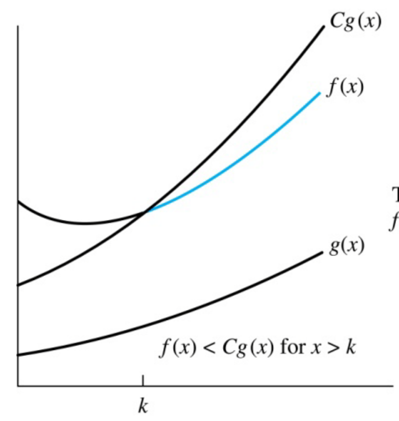


Big-O Notation

Definition: **Let f and g be functions from Z(or R) to R. 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 $x > k$.



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

- "$f(x)$ is $O(g(x))$" is read as: "$f(x)$ is big-oh of $g(x)$"
- The constants C and k are called *witnesses* to the relationship $f(x)$ is $O(g(x))$. Only one pair of witnesses is needed.

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 $O(g(x))$."**
  - 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 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:

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

$\leq x^2 + 2x^2 + 1$ For all $x > 1$

$\leq x^2 + 2x^2 + x^2$ For all $x > 1$

$= 4x^2 \quad = Cx^2 \quad = Cg(x)$

We have: $C = 4, \quad k = 1, \quad g(x) = x^2$

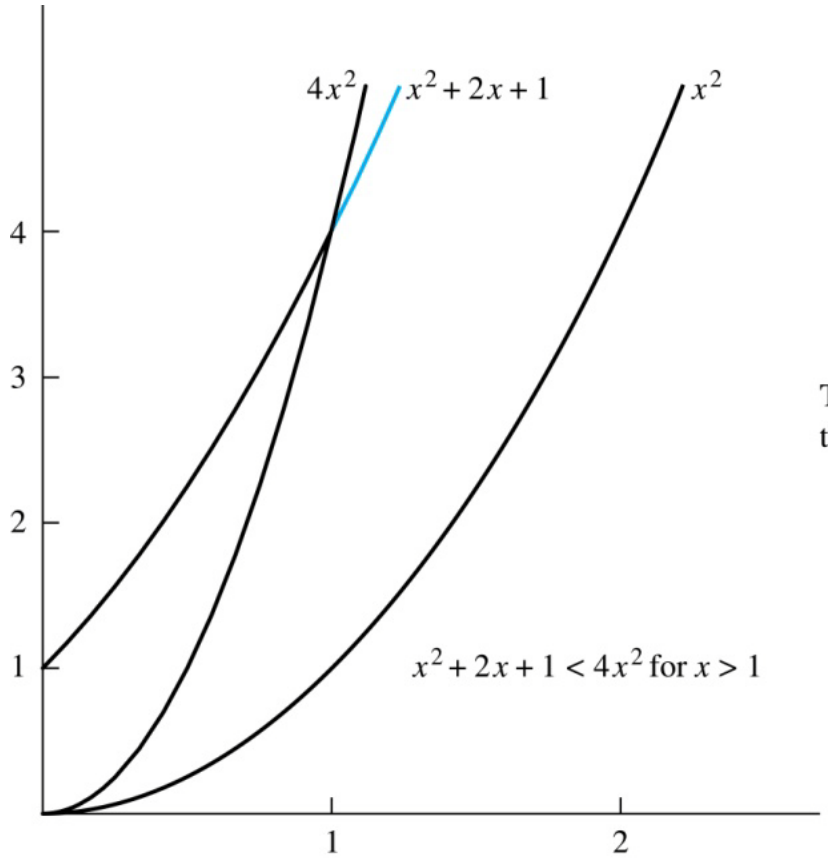
$f(x)$ is $O(x^2)$ (see graph on next slide)

2) $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$ whenever $x>2$

3) x^2 is $O(x^2+2x+1)$

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 $|h(x)| > |g(x)|$ 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 $O(x^3)$. Is it also true x^3 is $O(7x^2)$?

***Solution*:**

1) Note that when $x>7$, we have $7x^2 < x^3$.

Consequently, we can take $C=1$, and $k=7$, and to establish the relation $7x^2$ is $O(x^3)$.

2) $x^3 \le C(7x^2)$

$x \le 7C$

Note that no C exists for which $x \le 7C$ for all $x>k$.

Big-O Estimates for Polynomials

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

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

Proof:

Using the triangle inequality, if $x > 1$ we have

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0| \\
&\le |a_n| x^n + |a_{n-1}| x^{n-1} + \ldots + |a_1| x + |a_0| \\
&= x^n (|a_n| + |a_{n-1}|/x + \ldots + |a_1|/x^{n-1} + |a_0|/x^n) \\
&\le x^n (|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|)
\end{aligned}
$$

It follows that $|f(x)| \le Cx^n$

Big-O Estimates for some Important Functions

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

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

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

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

$$f(n) = n! = 1 \times 2 \times \cdots \times n\ .$$

Solution:

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

$$n! \text{ is } O(n^n) \text{ taking } C = 1 \text{ and } k = 1.$$

Big-O Estimates for some Important Functions

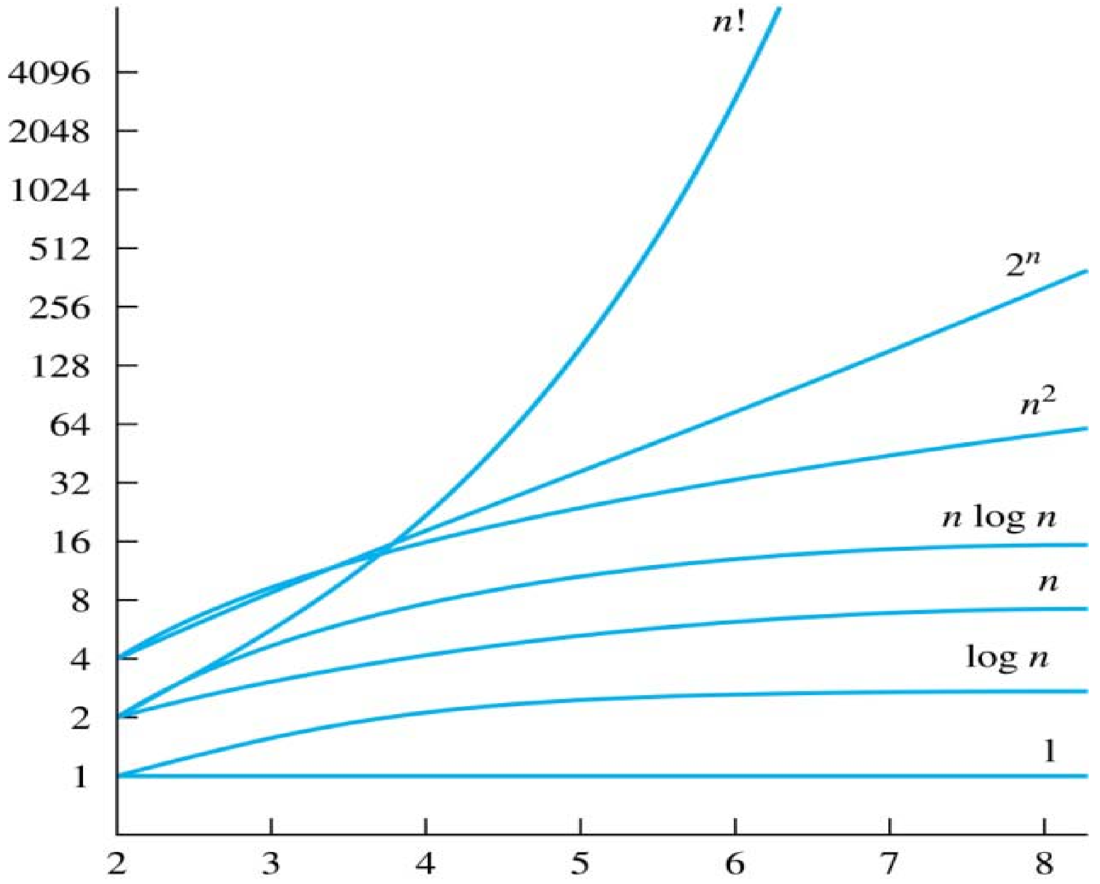
Example: Use big-*O* notation to estimate log *n*!

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

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

Hence, $\log(n!)$ is $O(n \cdot \log(n))$ taking $C = 1$ and $k = 1$.

Display of Growth of Functions



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

Useful Big-O Estimates Involving Logarithms, Powers, and Exponents

- If $d > c > 1$, then

  n^c is $O(n^d)$, but n^d is not $O(n^c)$.
- If $b > 1$ and c and d are positive, then

  $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$.
- If $b > 1$ and d is positive, then

  n^d is $O(b^n)$, but b^n is not $O(n^d)$.
- If $c > b > 1$, then

  b^n is $O(c^n)$, but c^n is not $O(b^n)$.

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 $O(g(x))$.**

  - By the definition of big-O notation, there are constants C_1, C_2, k_1, k_2 such that $|f_1(x) \le C_1|g_1(x)|$ when $x > k_1$ and $f_2(x) \le C_2|g_2(x)|$ when $x > k_2$.
  - $|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)|$

    $\le |f_1(x)| + |f_2(x)|$ by the triangle inequality $|a + b| \le |a| + |b|$
  - $|f_1(x)| + |f_2(x)| \le C_1|g_1(x)| + C_2|g_2(x)|$

    $\le C_1|g(x)| + C_2|g(x)|$ where $g(x) = \max(|g_1(x)|, |g_2(x)|)$

    $= (C_1 + C_2)\,|g(x)|$

    $= C|g(x)|$ where $C = C_1 + C_2$
  - Therefore $|(f_1 + f_2)(x)| \le C|g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$.

Combinations of Functions

Example : What is the complexity of the function $n^2 + \log(n!)$?

Solution:

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

$$\log(n!) = O(n \log n)$$

Since $O(n^2) > O(n \log n)$,

$$n^2 + n \log(n!) = O(n^2)$$

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

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

Solution:

$$3n = O(n)$$

$$\log(n!) = O(n \log n)$$

$$3n\log(n!) = O(n \times n \log n) = O(n^2 \log n)$$

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.

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

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

$f_8(n) = 10000$ (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)
$f_1(n) = (1.5)^n$ (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)
$f_9(n) = n!$ ($n!$ grows faster than c^n for every c)

Big-Omega Notation

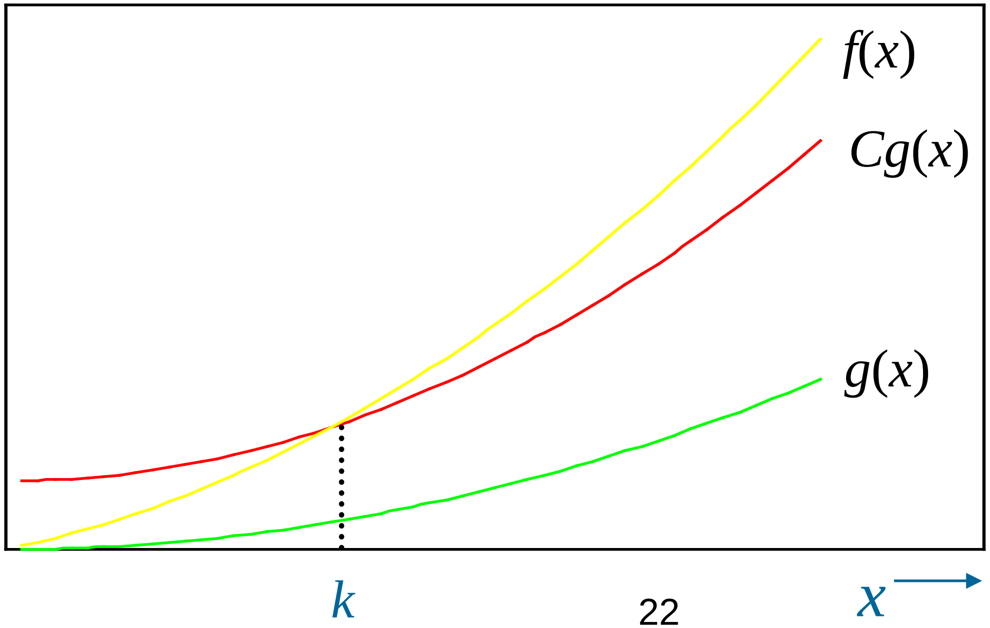
Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$

if there are constants *C* and *k* such that

$$|f(x)| \geq C|g(x)| \quad \text{when } x > k.$$

We say that "*f*(*x*) is big-Omega of *g*(*x*)."

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



Big-Omega Notation

- **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 $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. This follows from the definitions.** See the text for details.

Big-Omega Notation

Example :Show that $f(x) = 8x^3 + 5x^2 + 7$ **is** $\Omega(x^3)$.

Solution:

$$f(x) = 8x^3 + 5x^2 + 7$$

$$\geq 8x^3 \qquad \text{For all } x > 1$$

$$= Cx^3 \qquad = Cg(x)$$

We have: $C = 8, \quad k = 1, \quad g(x) = x^3$

$f(x)$ **is** $\Omega(x^3)$

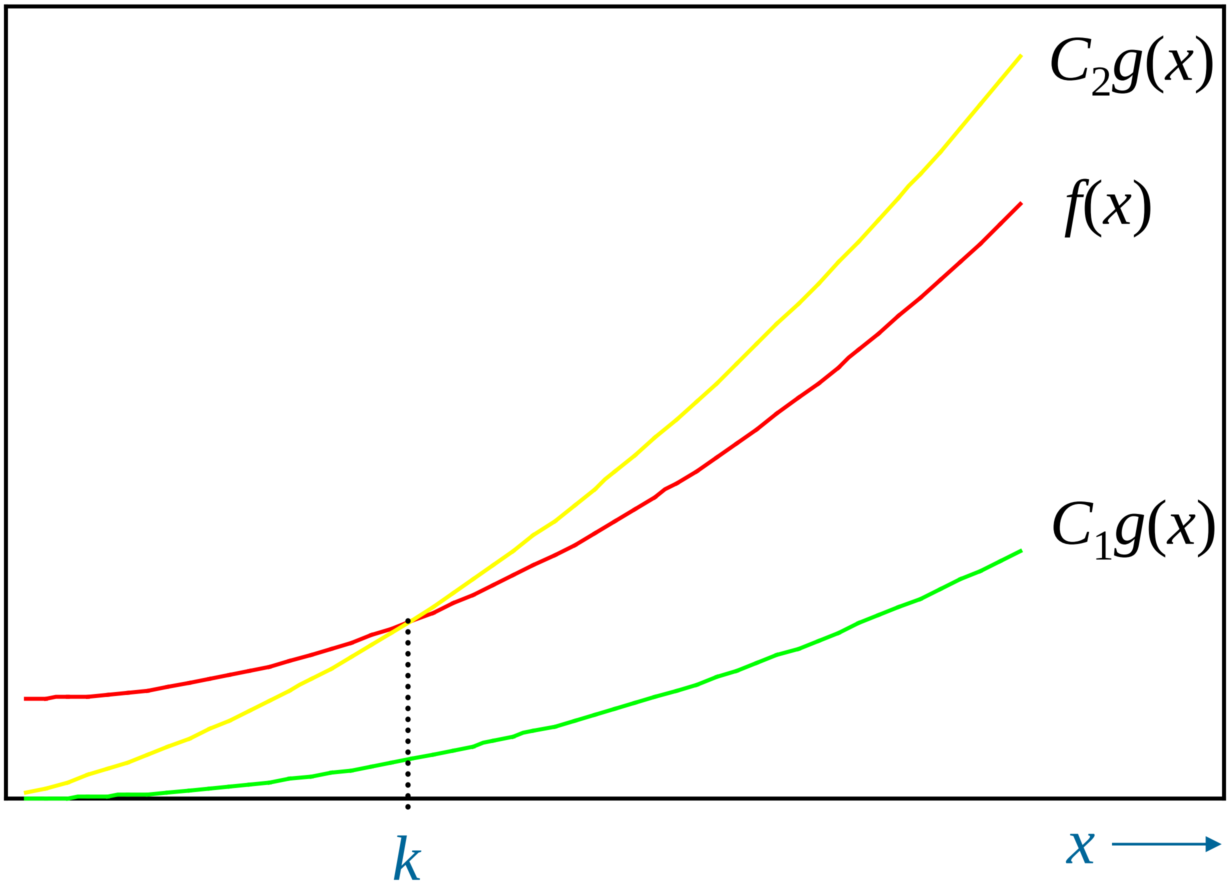
Big-Theta

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 $\Omega(g(x))$.

- 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



Big-Theta Notation

〖Example 7〗 Show that $f(x) = 3x^2 + 8x\log x$ is $\Theta(x^2)$.

Solution:

$$\left.\begin{aligned} f(x) &= 3x^2 + 8x\log x \\ &\le 3x^2 + 8x^2 \qquad \text{For } x > 1 \\ &= 11x^2 \quad = C_2x^2 \quad = C_2 g(x) \end{aligned}\right\} \quad f(x) \text{ is } O(x^2)$$

$$\left.\begin{aligned} f(x) &= 3x^2 + 8x\log x \\ &\ge 3x^2 \qquad \text{For } x > 1 \\ &= C_1x^2 \quad = C_1 g(x) \end{aligned}\right\} \quad f(x) \text{ is } \Omega(x^2)$$

We have: $C_1 = 3,\ C_2 = 11,\ k = 1,\ g(x) = x^2$

$f(x)$ is $\Theta(x^2)$

Big-Theta Notation

- **When** $f(x)$ is $\Theta(g(x))$ **it must also be the case that** $g(x)$ is $\Theta(g(x))$.
- **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** $\Theta(x^n)$**).**

(The proof is an exercise.)

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 $\Theta(x^{199})$).

Complexity of Algorithms

Section 3.3

The Complexity of Algorithms

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

- **In this course, we focus on time complexity. The space complexity of algorithms is studied in later courses.**

The Complexity of Algorithms

- 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.

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 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

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)
  $max := a_1$
  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 $n - 1$ 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

Example: 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*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

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 $\Theta(n)$.

Average-Case Complexity of Linear Search

Example: Describe the average case performance of the linear search algorithm. (Although usually it is very difficult to determine average-case complexity, it is easy for linear search.)

Solution: Assume the element is in the list and that the possible positions are equally likely. By the argument on the previous slide, if $x = a_i$, the number of comparisons is $2i + 1$.

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} = \frac{2\left[\frac{n(n+1)}{2}\right]}{n} + 1 = n + 2$$

Hence, the average-case complexity of linear search is $\Theta(n)$.

Understanding the Complexity of Algorithms

TABLE 1 **Commonly Used Terminology for the Complexity of Algorithms.**

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Homework:

Seventh Edition: P. 216 8,26,31,54,71