Chapter 3 Algorithms

Chapter Summary

- ✓ **Algorithms**
  - **Example Algorithms**
  - **Algorithmic Paradigms**
- ✓ **Growth of Functions**
  - **Big-O and other Notation**
- ✓ **Complexity of Algorithms**

3.1 Algorithms

Problems and Algorithms

- **In many domains there are key general problems that ask for output with specific properties when given valid input.**
  - **to precisely state the problem, using the appropriate structures to specify the input and the desired output.**
  - **solve the general problem by specifying the steps of a procedure that takes a valid input and produces the desired output.**
    - **This procedure is called an *algorithm*.**

Algorithms

Definition: An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

Example: Describe an algorithm for finding the maximum value in a finite sequence of integers.

Solution: Perform the following steps:

1. **Set the temporary maximum equal to the first integer in the sequence.**
2. **Compare the next integer in the sequence to the temporary maximum.**
   - **If it is larger than the temporary maximum, set the temporary maximum equal to this integer.**
3. **Repeat the previous step if there are more integers. If not, stop.**
4. **When the algorithm terminates, the temporary maximum is the largest integer in the sequence.**

Specifying Algorithms

- **Algorithms can be specified in different ways. Their steps can be described in English or in pseudocode.**
- **Pseudocode is an intermediate step between an English language description of the steps and a coding of these steps using a programming language.**
- **The form of pseudocode we use is specified in Appendix 3. It uses some of the structures found in popular languages such as C++ and Java.**
- **Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language.**
- **Pseudocode helps us analyze the time required to solve a problem using an algorithm, independent of the actual programming language used to implement algorithm.**

Properties of Algorithms

Input: An algorithm has input values from a specified set.

Output: From the input values, the algorithm produces the output values from a specified set. The output values are the solution.

Definiteness: The steps of an algorithm must be defined precisely.

Correctness: An algorithm should produce the correct output values for each set of input values.

Finiteness: An algorithm should produce the output after a finite number of steps for any input.

Effectiveness: It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

Generality: The algorithm should work for all problems of the desired form.

Finding the Maximum Element in a Finite Sequence

- The algorithm in pseudocode:

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}**

Does this algorithm have all the properties listed on the previous slide?

Some Example Algorithm Problems

- **Three classes of problems will be studied in this section.**

  1. ***Searching Problems*: finding the position of a particular element in a list.**
  2. ***Sorting problems*: putting the elements of a list into increasing order.**
  3. ***Optimization Problems*: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.**

Searching Problems

Definition: The general searching problem

Locate an element x in a list of distinct elements $a_1, a_2, \ldots, a_n$ or determine that it is not in the list.

- **The solution to a searching problem**
  - **the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) or 0 if x is not in the list**
- **For example,**
  - **a library might want to check to see if a patron is on a list of those with overdue books before allowing him/her to checkout another book.**
- **Two different searching algorithms:**
  - **linear search and binary search**

Linear Search or sequential search

- An algorithm that linearly searches a sequence for a particular element.

ALGORITHM **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

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

Binary Search

Example, To search for 88 in the list

5 13 19 21 37 56 64 75 80 88 92 100.

Solution :

(1) **5 13 19 21 37 56 64 75 80 88 92 100**

(2) **64 75 80 88 92 100**

(3) **88 92 100**

(4) **88 92**

- **A binary search algorithm is more efficient than linear search**
  - The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.

Algorithm for Binary Search

ALGORITHM **The Binary Search Algorithm.**

Procedure *binary search* (x: integer,

$a_1, a_2, ..., a_n$: increasing integers)

i : = 1{i is left endpoint of search interval }

j: =n { j is right endpoint of search interval }

while $i < j$

begin

m: = $\lfloor (i + j)/2 \rfloor$

if $x > a_m$ then $i := m + 1$

else $j := m$

end

if $x = a_i$ then *location* : = i

else *location* : = 0

{location is the subscript of term equal to x ,or 0 if x is not found }

Obviously, on sorted sequences, binary search is more efficient than linear search.

How can we analyze the efficiency of algorithms?

We can measure the

- **time (number of elementary computations) and**
- **space (number of memory cells) that the algorithm requires.**

These measures are called computational complexity and space complexity, respectively.

Sorting

- To sort the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on).
- Sorting is an important problem because:
  - A nontrivial percentage of all computing resources are devoted to sorting different kinds of lists, especially applications involving large databases of information that need to be presented in a particular order (e.g., by customer, part number etc.).
  - An amazing number of fundamentally different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively.
  - Sorting algorithms are useful to illustrate the basic notions of computer science.
- A variety of sorting algorithms are studied in this book: binary, insertion, bubble, selection, merge, quick, and tournament.

Greedy Algorithms

- **Optimization problems:** minimize or maximize some parameter over all possible inputs.
- Among the many optimization problems we will study are:
  - Finding a route between two cities with the smallest total mileage.
  - Determining how to encode messages using the fewest possible bits.
  - Finding the fiber links between network nodes using the least amount of fiber.
- Optimization problems can often be solved using a **greedy algorithm**, which makes the "best" choice at each step.
- Making the "best choice" at each step does not necessarily produce an optimal solution to the overall problem, but in many instances, it does.
- After specifying what the "best choice" at each step is, we try to prove that this approach always produces an optimal solution, or find a counterexample to show that it does not.
- The greedy approach to solving problems is an example of an algorithmic paradigm, which is a general approach for designing an algorithm. We return to algorithmic paradigms in Section 3.3.

Greedy Algorithms: Making Change

Example: Design a greedy algorithm for making change (in U.S. money) of *n* cents with the following coins: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent) , using the least total number of coins.

Idea: **At each step choose the coin with the largest possible value that does not exceed the amount of change left.**

1. **If *n* = 67 cents, first choose a quarter leaving 67−25 = 42 cents. Then choose another quarter leaving 42 −25 = 17 cents**
2. **Then choose 1 dime, leaving 17 − 10 = 7 cents.**
3. **Choose 1 nickel, leaving 7 – 5 = 2 cents.**
4. **Choose a penny, leaving one cent. Choose another penny leaving 0 cents.**

Greedy Algorithms: Making Change

Solution: Greedy change-making algorithm for n cents. The algorithm works with any coin denominations $c_1, c_2, \ldots, c_r$.

procedure *change*($c_1, c_2, \ldots, c_r$: values of coins, where $c_1 > c_2 > \ldots > c_r$; n: a positive integer)

for $i := 1$ to r

$d_i := 0$ [d_i counts the coins of denomination c_i]

while $n \geq c_i$

$d_i := d_i + 1$ [add a coin of denomination c_i]

$n = n - c_i$

[d_i counts the coins c_i]

Proving Optimality for U.S. Coins

- **Show that the change making algorithm for *U.S.* coins is optimal.**

Lemma 1: If *n* is a positive integer, then *n* cents in change using quarters, dimes, nickels, and pennies, using the fewest coins possible has at most 2 dimes, 1 nickel, 4 pennies, and cannot have 2 dimes and a nickel. The total amount of change in dimes, nickels, and pennies must not exceed 24 cents.

Proof: By contradiction

- **If we had 3 dimes, we could replace them with a quarter and a nickel.**
- **If we had 2 nickels, we could replace them with 1 dime.**
- **If we had 5 pennies, we could replace them with a nickel.**
- **If we had 2 dimes and 1 nickel, we could replace them with a quarter.**
- **The allowable combinations, have a maximum value of 24 cents: 2 dimes and 4 pennies.**

Proving Optimality for U.S. Coins

Theorem: The greedy change-making algorithm for U.S. coins produces change using the fewest coins possible.

Proof: By contradiction.

1. **Assume there is a positive integer *n* such that change can be made for *n* cents using quarters, dimes, nickels, and pennies, with a fewer total number of coins than given by the algorithm.**
2. **Then, $q' \leq q$ where q' is the number of quarters used in this optimal way and *q* is the number of quarters in the greedy algorithm's solution. But this is not possible by Lemma 1, since the value of the coins other than quarters can not be greater than 24 cents.**
3. **Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and quarters.**

3.2

The Growth of Functions

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)

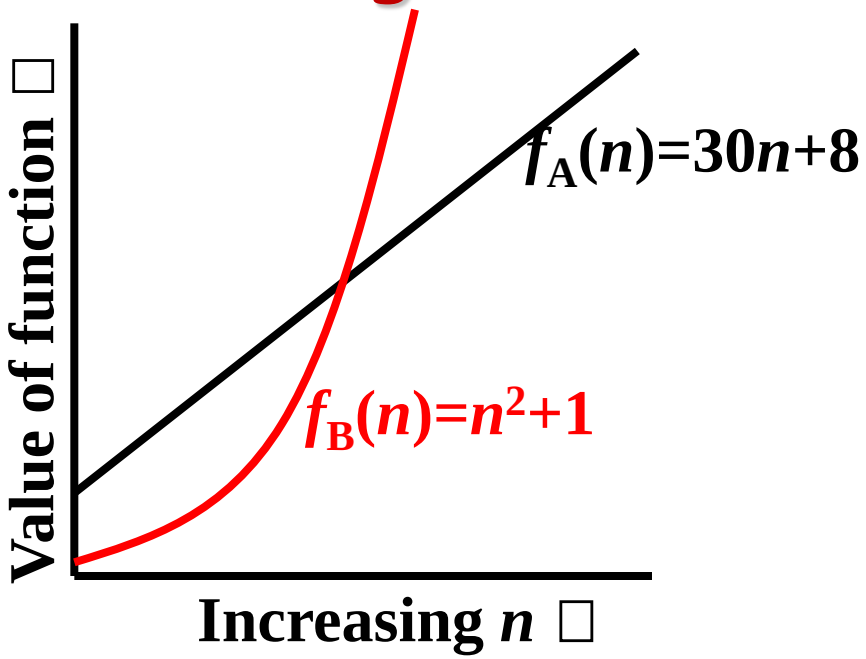
Example of Orders of Growth

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...

Big-O?

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.

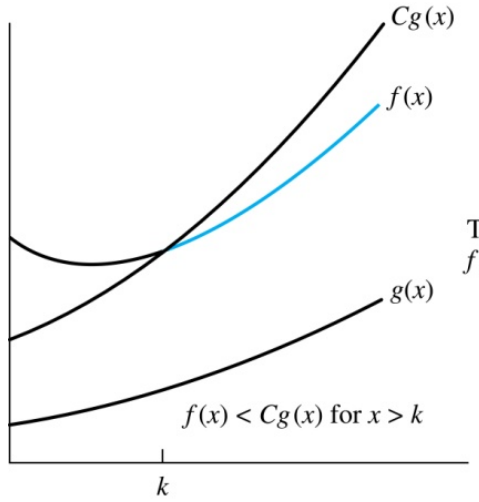
For large numbers of user records, the $O(n^2)$ function will always take more time.

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

Example1 : Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution：

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

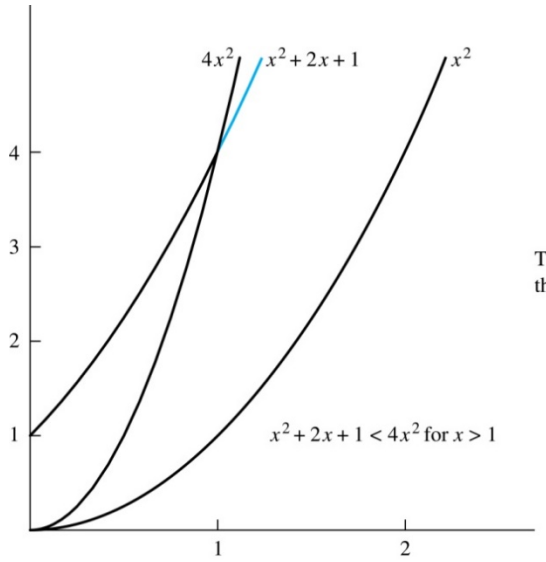
$$\leq x^2 + 2x^2 + 1 \qquad \text{For all } x > 1$$

$$\leq x^2 + 2x^2 + x^2 \qquad \text{For all } x > 1$$

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

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

$f(x)$ is $O(x^2)$



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

Note：

- $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$ whenever $x>2$
- x^2 is $O(x^2+2x+1)$

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

Example2 : 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) $x^3 \leq C(7x^2)$

$x \leq 7C$

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

Big-O Estimates for Polynomials

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

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 $O(x^n)$.**

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

Example3: 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 $k = 1$.

Example4: Use big-*O* notation to estimate the factorial function

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

Solution:

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

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

Big-O Estimates for some Important Functions

Example5: Use big-*O* notation to estimate log *n*!

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

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

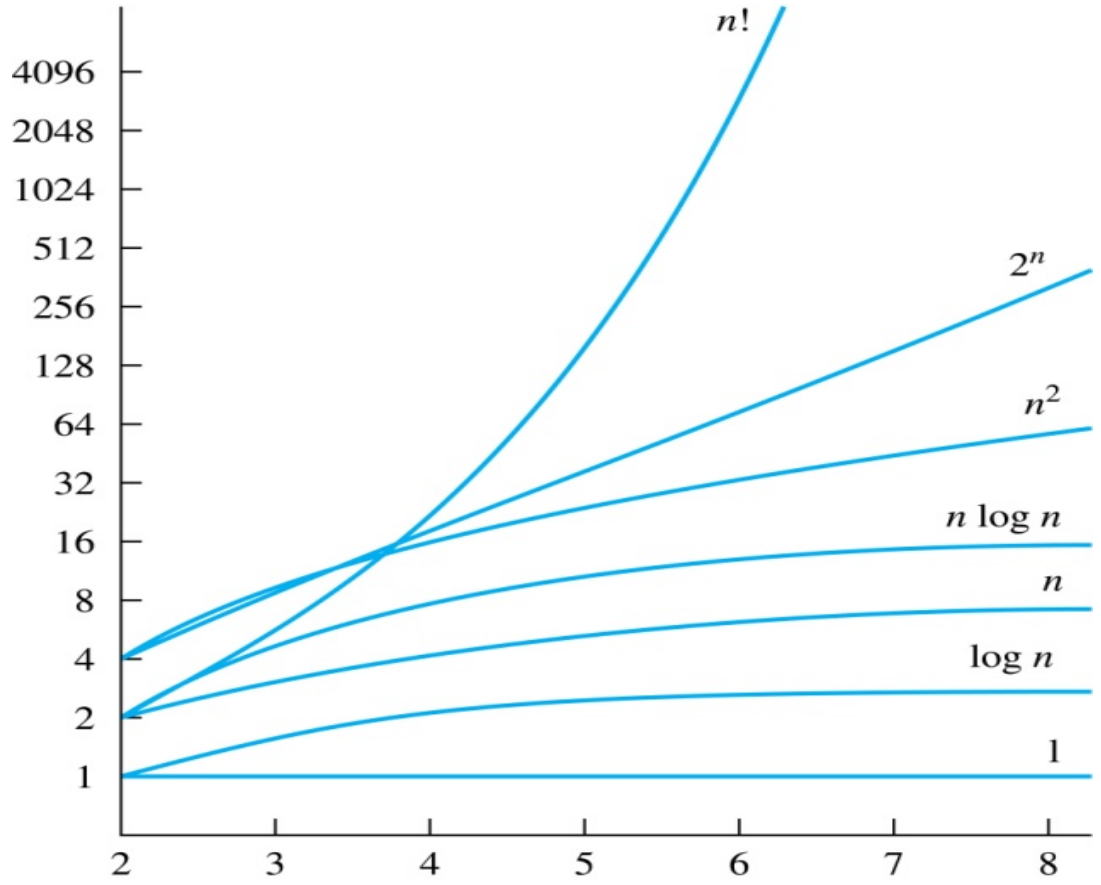
Hence, log(*n*!) is $O(n \cdot \log(n))$ taking $C = 1$ and $k = 1$.

Big-O Estimates for some Important Functions

Problem:

1. n is $O(2^n)$?
2. $\log n < n$?
3. $\log_b n$ is $O(n)$?

Display of the Growth of Functions commonly used in big-O estimates



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 $O((\log_b n)^c)$.
  - Every positive power of the logarithm of n to the base b, where $b > 1$, is big-O of every positive power of n, but the reverse relationship never holds.
- If $b > 1$ and d is positive, then n^d is $O(b^n)$, but b^n is not $O(n^d)$.
  - Every power of n is big-O of every exponential function of n with a base that is greater than one, but the reverse relationship never holds.
- If $c > b > 1$, then b^n is $O(c^n)$, but c^n is not $O(b^n)$.
  - if we have two exponential functions with different bases greater than one, one of these functions is big-O of the other if and only if its base is smaller or equal.

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)

The Growth of 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 $|f_1(x) \leq C_1|g_1(x)|$ when $x > k_1$ and $f_2(x) \leq C_2|g_2(x)|$ when $x > k_2$.
- $|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)|$
  $\leq |f_1(x)| + |f_2(x)|$ by the triangle inequality $|a + b| \leq |a| + |b|$
- $|f_1(x)| + |f_2(x)| \leq C_1|g_1(x)| + C_2|g_2(x)|$
  $\leq C_1|g(x)| + C_2|g(x)|$ 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 $|(f_1 + f_2)(x)| \leq C|g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$.

The Growth of Combinations of Functions

Example6 : What is the complexity of the function $n^2 + \log(n!)$?

Solution:

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

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

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

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

The Growth of Combinations of Functions

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1f_2)(x)$ is $O(g_1(x)g_2(x))$.

Example7 : What is the complexity of the function $3n\log(n!)+ (n^2 + 3) \log n$?

Solution:

First, the product $3n \log(n!)$ will be estimated.

- From previous Example we know that $\log(n!)$ is $O(n \log n)$.
- Using this estimate and the fact that $3n$ is $O(n)$, Theorem 3 gives the estimate that $3n \log(n!)$ is $O(n^2 \log n)$.

Next, the product $(n^2 + 3) \log n$ will be estimated.

- Because $(n^2 + 3) < 2n^2$ when $n > 2$, it follows that $n^2 + 3$ is $O(n^2)$.
- Thus, from Theorem 3 it follows that $(n^2 + 3) \log n$ is $O(n^2 \log n)$.
- Using Theorem 2 to combine the two big-O estimates for the products shows that $f(n) = 3n \log(n!) + (n^2 + 3) \log n$ is $O(n^2 \log n)$.

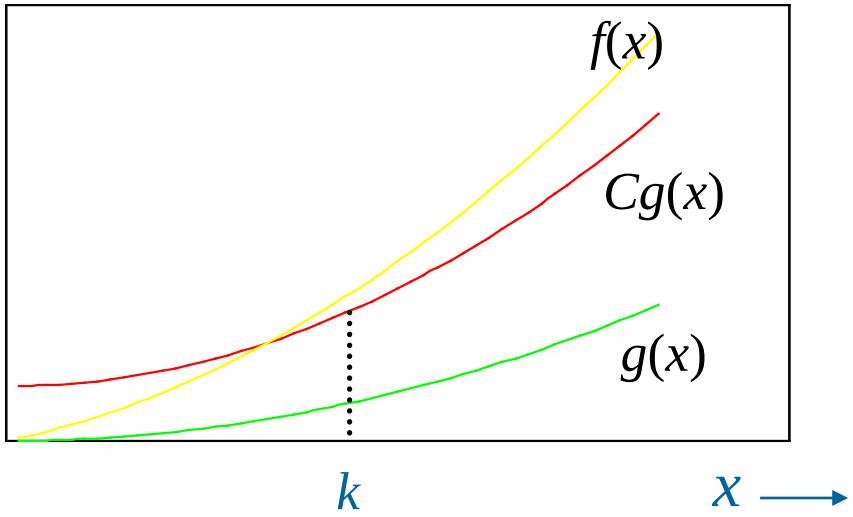
Big-Omega Notation

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**

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

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 8: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

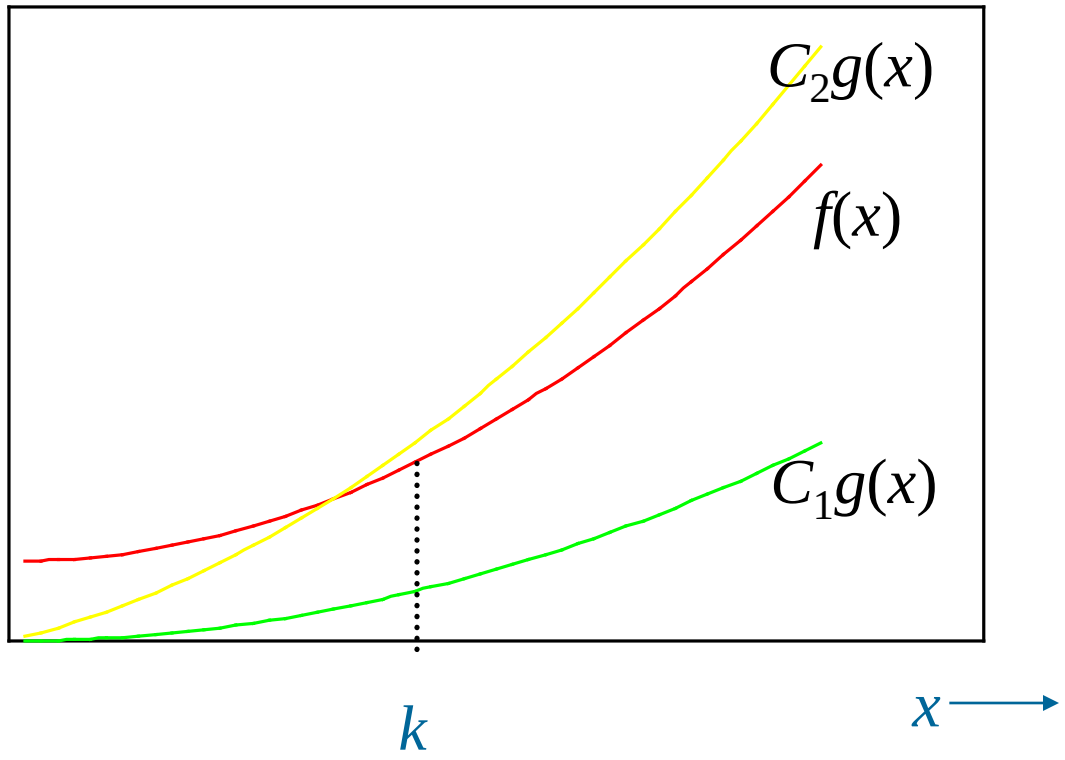
Θ is the upper case version of the lower case Greek letter θ.

Definition: Let f and g be functions from the set of integers or the set of real numbers. 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 9: Show that $f(x) = 3x^2 + 8x\log x$ is $\Theta(x^2)$.

Solution:

$$f(x) = 3x^2 + 8x\log x$$
$$\leq 3x^2 + 8x^2 \qquad \text{For } x > 1$$
$$= 11x^2 \qquad = C_2x^2 \qquad = C_2g(x)$$

} $f(x)$ is $O(x^2)$

$$f(x) = 3x^2 + 8x\log x$$
$$\geq 3x^2 \qquad \text{For } x > 1$$
$$= C_1x^2 \qquad = C_1g(x)$$

} $f(x)$ 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 10:

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

Homework:

SE: P. 216 8,26,31,54,71

EE: P. 229 8,26,31,54,71

3.3
Complexity of Algorithms

Introduction

- When does an algorithm provide a satisfactory solution to a problem?
  - **Correctness**
  - **Efficiency**

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

- How to measure time complexity?
  - in terms of the number of operations an algorithm uses , such as comparisons and arithmetic operations (addition, multiplication, etc.)
  - use big-*O* and big-Theta notation

- 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.

Types of Time Complexity analysis

- Types of Time Complexity analysis
  - worst-case time complexity
  - best-case time complexity
  - average case time complexity

- 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(a1, a2, ...., an: integers)
   max := a1
   for i := 2 to n
      if max < ai then max := ai
   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, a1, a2, ...,an: distinct integers)
i := 1
while (i ≤ n and x ≠ ai)
    i := i + 1
if i ≤ 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 $2i + 1$.

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

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

Others

- **Algorithmic Paradigms**
  - a general approach based on a particular concept that can be used to construct algorithms for solving a variety of problems.
- **Some Algorithmic paradigms**
  - Greedy algorithm
  - Brute-force algorithm
  - Divide-and-conquer algorithms
  - Dynamic programming
  - Backtracking
  - Probabilistic algorithms

Others

- **Understanding the Complexity of Algorithms**
  - **tractable vs. intractable**
  - **unsolvable vs. solvable**
  - **P vs. NP**
  - **NP-complete problems**