Fall 2021 - MATH 1101 Discrete Structures
Lecture 13. Algorithms and Complexity

PART 1. ALGORITHMS AND COMPLEXITY

PART 2. GROWTH OF FUNCTIONS

PART 3. SEARCHING AND SORTING ALGORITHMS AND COMPLEXITY ISSUES
(additional reading**)**

Introduction

There are many general classes of problems that arise in discrete mathematics. For instance,

- given a sequence of integers, find the largest one;
- given a set, list all its subsets;
- given a set of integers, put them in increasing order;
- given a network, find the shortest path between two vertices.

When presented with such a problem, the first thing to do is to construct a model that translates the problem into a mathematical context. Discrete structures used in such models include sets, sequences, functions, permutations, relations, graphs, trees, networks, and finite state machines etc.

Setting up the appropriate mathematical model is only part of the solution. To complete the solution, a method is needed that will solve the general problem using the model. Ideally, what is required is a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence of steps is called an **algorithm**.

In suggested Lecture Notes, among others, we introduce algorithms for two extremely important problems in computer science:

- **searching** for an element in a list;
- **sorting** a list so its elements are in some prescribed order, such as increasing, decreasing, or alphabetic.

We also introduce the notion of an **algorithmic paradigm**, which provides a general method for designing algorithms. We discuss:

- **brute-force algorithms**, which find solutions using a straightforward approach;
- **greedy algorithms**, that follow the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum (usually they are used to solve optimization problems, for example, Dijkstra's algorithm).

Proofs are extremely important in the study of algorithms.

One important issue concerning an algorithm is its computational **complexity**, which measures the **processing time** and **computer memory** required by the algorithm to solve problems of a particular size. To measure the complexity of algorithms we use big-O and big-Θ notation, which we develop here. We illustrate the analysis of the complexity of algorithms, focusing on the time an algorithm takes to solve a problem. Furthermore, we discuss what the time complexity of an algorithm means in practical and theoretical terms.

PART 1. ALGORITHMS AND COMPLEXITY.

Definition 1: An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem. ■

EXAMPLE 1. Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.

Solution. We can specify a procedure for solving this problem in several ways. One method is simply to use the English language to describe the sequence of steps used. We now provide such a solution below.

We perform the following steps.

1. Set the temporary maximum equal to the first integer in the sequence. (The temporary maximum will be the largest integer examined at any stage of the procedure.)
2. Compare the next integer in the sequence to the temporary maximum, and 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 in the sequence.
4. Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence.

An algorithm can also be described using a computer language (pseudocodes).

ALGORITHM 1. 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} ■

Properties of Algorithms

There are several properties that algorithms generally share. They are useful to keep in mind when algorithms are described. These properties are:

- ***Input***. An algorithm has input values from a specified set.
- ***Output***. From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
- ***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 desired output after a finite (but perhaps large) number of steps for any input in the set.
- ***Effectiveness***. It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- ***Generality***. The procedure should be applicable for all problems of the desired form, not just for a particular set of input values. ■

EXAMPLE 2. Show that Algorithm 1 from Example 1 for finding the maximum element in a finite sequence of integers has all the properties listed.

Solution:

- The **input** to Algorithm 1 is a sequence of integers.
- The **output** is the largest integer in the sequence.
- Each step of the algorithm is **precisely defined**, because only assignments, a finite loop, and conditional statements occur.
- To show that the algorithm is **correct**, we must show that when the algorithm terminates, the value of the variable *max* equals the maximum of the terms of the sequence. To see this, note that the initial value of *max* is the first term of the sequence; as successive terms of the sequence are examined, *max* is updated to the value of a term if the term exceeds the maximum of the

terms previously examined. This (informal) argument shows that when all the terms have been examined, *max* equals the value of the largest term.

- The algorithm uses a **finite number of steps,** because it terminates after all the integers in the sequence have been examined.
- The algorithm can be carried out in a **finite amount of time** because each step is either a comparison or an assignment, there are a finite number of these steps, and each of these two operations takes a finite amount of time.
- Finally, Algorithm 1 is **general**, because it can be used to find the maximum of any finite sequence of integers. ■

Complexity of Algorithms

The analysis of algorithms is a major task in computer science. To compare algorithms, we must have some criteria to measure the **efficiency of algorithms**.

Suppose *M* is an algorithm, and suppose *n* is the size of the input data. The **time** and **space** used by the algorithm are the two main measures for the efficiency of *M*.

The ***complexity*** of an algorithm *M* is the function f(n) which gives the **running time** and/or **storage space** requirement of the algorithm in terms of the size *n* of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size. Accordingly, unless otherwise stated or implied, **the term "complexity" shall refer to the running time of the algorithm**.

The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. The operations used to measure time complexity can be:

- the comparison of integers,
- the addition of integers,
- the multiplication of integers,
- any other basic operation.

Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations.

EXAMPLE 3. Describe the time complexity of Algorithm 1 from Example 1 for finding the maximum element in a finite set of integers.

Solution: The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

To find the maximum element of a set with *n* elements, listed in an arbitrary order, the temporary maximum is first set equal to the initial term in the list. Then, after a comparison $i \leq n$ has been done to determine that the end of the list has not yet been reached, the temporary maximum and second term are compared, updating the temporary maximum to the value of the second term if it is larger. This procedure is continued, using two additional comparisons for each term of the list - one $i \leq n$, to determine that the end of the list has not been reached and another $max < a_i$, to determine whether to update the temporary maximum. Because two comparisons are used for each of the second through the nth elements and one more comparison is used to exit the loop when $i=n+1$, exactly $2(n-1)+1=2n-1$ comparisons are used whenever this algorithm is applied. Hence, the time complexity of the algorithm for finding the maximum of a set of n elements, **measured in terms of the number of comparisons used,** is the linear function of n. We say that this algorithm has time complexity $\Theta(n)$ (linear function on *n*, details are in Part 2). Note that for this algorithm the number of comparisons is independent of particular input of n numbers. ■

The complexity function f(n), which we assume gives the running time of an algorithm,

usually depends not only on the size *n* of the input data but also on the particular data. For example, suppose we want to search through an English short story TEXT for the first occurrence of a given 3-letter word *W*. Clearly, if *W* is the 3-letter word "the," then *W* likely occurs near the beginning of TEXT, so f(n) will be small. On the other hand, if *W* is the 3-letter word "zoo," then *W* may not appear in TEXT at all, so f(n) will be large.

The above discussion leads us to the question of finding the complexity function f(n) for certain cases. The two cases one usually investigates in complexity theory are as follows:

(1) *Worst-case complexity*: The maximum value of f(n) for any possible input.

The type of complexity analysis in Example 3 is a ***worst-case complexity***. By the worst-case performance of an algorithm, we mean the largest number of operations needed to solve the given problem using this algorithm on input of specified size. Worst-case analysis tells us how many operations an algorithm requires **to guarantee** that it will produce a solution. ■

(2) *Average-case complexity*: The expected value of f(n).

In **average-case** analysis, the average number of operations used to solve the problem over all possible inputs of a given size is found. The analysis of the average case assumes a certain probabilistic distribution for the input data; one possible assumption might be that the possible permutations of a data set are equally likely. The average case also uses the following concept in probability theory. Suppose the numbers n_1, n_2, ..., n_k occur with respective probabilities p_1, p_2, ..., p_k. Then the *expectation* or *average value E for the complexity function* is given by

$$E=n_1p_1+n_2p_2+\cdots+ n_{1k}p_k$$ ■

Average-case time complexity analysis is usually much more complicated than worst-case analysis. However, the average-case analysis for some algorithms can be done without difficulty, for example, for the linear search algorithm. ■

Searching Algorithms I. We start to discuss one of searching algorithms (Linear Search=LS) now because of simplicity of a LS itself and average-case time complexity of it. The other searching algorithms and their complexity will be discussed in Part 3 in details.

The general searching problem can be described as follows: Locate an element x in a list of distinct elements a_1, a_2, ..., a_n, or determine that it is not in the list. The solution to this search problem is the location of the term in the list that equals x, that is, *i* is the solution of the searching problem if $x=a_i$ and is 0 if x is not in the list.

The Linear Search. The linear search algorithm begins by comparing x and a_1. When $x=a_1$, the solution is the location of a_1, namely, 1. When $x\neq a_1$, compare x with a_2. If $x=a_2$, the solution is the location of a_2, namely, 2. When $x\neq a_2$, compare x with a_3. Continue this process, comparing x successively with each term of the list until a match is found, where the solution is the location of that term, unless no match occurs. If the entire list has been searched without locating x, the solution is 0. The pseudocode for the linear search algorithm is displayed as Algorithm 2.

ALGORITHM 2. The Linear Search Algorithm.

procedure *linear search* (x: integer, a_1, a_2, ..., 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} ■

EXAMPLE 4. Describe the average-case performance of the linear search algorithm (ALGORITHM 2) in terms of the average number of comparisons used, assuming that the integer x is in the list and it is equally likely that x is in any position.

Solution: By hypothesis, the integer x is one of the integers a_1, a_2, ..., a_n in the list. If x is the first term a_1 of the list, three comparisons are needed, one $i \leq n$ to determine whether the end of the list has been reached, one $x \neq a_i$ to compare x and the first term, and one $i \leq n$ outside the loop. If x is the second term a_2 of the list, two more comparisons are needed, so that a total of five comparisons are used. In general, if x is the i-th term of the list a_i, two comparisons will be used at each of the i steps of the loop, and one outside the loop, so that a total of 2i + 1 comparisons are needed. Hence, the average number of comparisons used equals

$$\frac{3 + 5 + 7 + \cdots + (2n + 1)}{n} = \frac{(3 + (2n + 1))}{2n} n = n + 2$$

which is again the linear function of n, that is, the ALGORITHM 2 has an average-case time complexity $\Theta(n)$. ■

EXAMPLE 5. Describe the worst-case complexity of the linear search algorithm.

Solution: The number of comparisons used by Algorithm 2 will be taken as the measure of the time complexity. At each step of the loop in the algorithm, two comparisons are performed—one $i \leq n$, to see whether the end of the list has been reached and one $x \leq a_i$, to compare the element x with a term of the list. Finally, one more comparison $i \leq n$ is made outside the loop. Consequently, if $x = a_i$, (2i+1) comparisons are used. The most comparisons, (2n+2), are required when the element is not in the list. In this case, 2n comparisons are used to determine that x is not a_i, for i=1, 2, ..., n, an additional comparison is used to exit the loop, and one comparison is made outside the loop. So when x is not in the list, a total of (2n+2) comparisons are used.

Hence, the worst-case complexity of a linear search is the linear function of n:

$$f(n)=2n+1 \text{ or } f(n)=2n+2,$$

depending on whether the element is in list or not, that is, the algorithm requires $\Theta(n)$ comparisons in the worst case.

Exercises

1. List all the steps used by Algorithm 1 to find the max of the list {1, 8, 12, 9, 11, 2, 14, 5, 10, 4}.

2. Devise an algorithm that finds the sum of all the integers in a list.

3. Describe an algorithm that takes as input a list of n integers in non-decreasing order and produces the list of all values that occur more than once. (Recall that a list of integers is non-decreasing if each integer in the list is at least as large as the previous integer in the list.)

4. Describe an algorithm that takes as input a list of n integers and finds the location of the last even integer in the list or returns 0 if there are no even integers in the list.

5. A **palindrome** is a string that reads the same forward and backward. Describe an algorithm for determining whether a string of n characters is a palindrome.

6. Describe an algorithm that interchanges the values of the variables x and y, using only assignments. What is the minimum number of assignment statements needed to do this?

7. List all the steps used to search for 9 in the sequence 1, 3, 4, 5, 6, 8, 9, 11 using

a) a linear search. **b)** a binary search.

8. Describe an algorithm that inserts an integer x in the appropriate position into the list a_1, a_2,...,a_n of integers that are in increasing order.

9. Describe an algorithm that locates the first occurrence of the largest element in a finite list of integers, where the integers in the list are not necessarily distinct.

10. Describe an algorithm that produces the maximum, median, mean, and minimum of a set of three integers. (The median of a set of integers is the middle element in the list when these integers are listed in order of increasing size. The mean of a set of integers is the sum of the integers divided by the number of integers in the set.)

11. Describe an algorithm that puts the first three terms of a sequence of integers of arbitrary length in increasing order.

12. Describe an algorithm that determines whether a function from a finite set of integers to another finite set of integers is onto.

13. Describe an algorithm that will count the number of 1s in a bit string by examining each bit of the string to deter- mine whether it is a 1 bit.

14. In a list of elements the same element may appear several times. A **mode** of such a list is an element that occurs at least as often as each of the other elements; a list has more than one mode when more than one element appears the maximum number of times.

Devise an algorithm that finds a mode in a list of non-decreasing integers. (Recall that a list of integers is non-decreasing if each term is at least as large as the preceding term.)

15. Devise an algorithm that finds the first term of a sequence of integers that equals some previous term in the sequence.

16. Devise an algorithm that finds the first term of a sequence of positive integers that is less than the immediately preceding term of the sequence.

17. Use the bubble sort to sort 3, 1, 5, 7, 4, showing the lists obtained at each step.

18. Sort these lists using the selection sort: **a)** 3, 5, 4, 1, 2; **b)** 5, 4, 3, 2, 1; **c)** 1, 2, 3, 4, 5

19. Describe an algorithm based on the linear search for determining the correct position in which to insert a new element in an already sorted list.

20. How many comparisons does the insertion sort use to sort the list 1, 2, ..., n?

ANSWERS

1. *max* :=1, i := 2, *max* := 8, i := 3, *max* := 12, i := 4, i:=5, i:=6, i:=7,

max:=14, i:=8, i:=9, i:=10, i := 11

2. procedure *AddUp*($a_1, \ldots, a_n$: integers)

sum := a_1

for i : = 2 **to** n

sum := *sum* + a_i

return *sum*

3. procedure *duplicates*($a_1, a_2, \ldots, a_n$: integers in non-decreasing order)

k := 0 {this counts the duplicates}

j := 2

while $j \leq n$

if $a_j = a_{j-1}$ **then**

k := k + 1

$c_k := a_j$

while $j \leq n$ and $a_j = c_k$

j := j + 1

j := j + 1

{c_1, c_2,. ..,c_k is the desired list}

4. procedure *last even location*(a_1,a_2, . . . ,a_n: integers)

k := 0

for i := 1 **to** n

if a_i is even **then** k:=i

return k {k = 0 if there are no evens}

5. procedure *palindrome check*(a_1 a_2 . . . a_n: string)

answer := **true**

for i := 1 **to** ⌊n/2⌋

if $a_i \neq a_{n+1-i}$ **then** *answer* := **false**

return *answer*

6. procedure *interchange*(x, y: real numbers)

z := x

x := y

y := z

The minimum number of assignments needed is three.

7. Linear search: i:=1, i:=2, i:=3, i:=4, i:=5, i:=6, i:=7, *location*:=7; binary search: i:=1, j:=8, m:=4, i:=5, m:=6, i:=7, m:=7, j:=7, *location*:=7

8. procedure *insert*(x, a_1, a_2, . . . , a_n: integers) {the list is in order: $a_1 \leq a_2 \leq \cdots \leq a_n$}

a_{n+1} :=x+1

i := 1

while x > a_i

i := i + 1

for j := 0 **to** n − i

a_{n-j+1} := a_{n-j}

a_i := x

{x has been inserted into correct position}

9. procedure *first largest*(a_1 , . . . , a_n: integers)

max := a_1

location := 1

for i := 2 **to** n

if *max* < a_i **then**

max := a_i

location := i

return *location*

10. procedure *mean-median-max-min*(a, b, c: integers)

mean := (a+b+c)/3 {the six different orderings of a, b, c with respect to ≥ will be handled separately}

if a≥b **then**

if b≥c **then** *median* := b; *max* := a; *min* := c

.

.

.

(The rest of the algorithm is similar.)

11. procedure *first-three*(a_1 , a_2 , . . . , a_n: integers)

if $a_1 > a_2$ **then** interchange a_1 and a_2

if $a_2 > a_3$ **then** interchange a_2 and a_3

if $a_1 > a_2$ **then** interchange a_1 and a_2

12. procedure *onto*(f : function from A to B where A = {a_1,...,a_n}, B = {b_1,...,b_m}, a_1,...,a_n, b_1,...,b_m are integers)

for i := 1 **to** m

hit(b_i) := 0

count := 0

for j :=1 **to** n

if *hit*(f (a_j)) = 0 **then**

hit(f (a_j)) := 1

count := *count* + 1

if *count* = m **then return true else return false**

13. procedure *ones*(a: bit string, a = a_1a_2 . . . a_n)

count:= 0

for i := 1 **to** n

if a_i := 1 **then**

count := *count* + 1

return *count*

14. procedure *find a mode*(a_1, a_2, . . . , a_n: non-decreasing integers)

value := a_i

count := 1

while i≤ n and a_i =*value*

count := *count* + 1

i := i + 1

if *count* > *modecount* **then**

modecount := *count*

mode := *value*

return *mode*

15. procedure *find duplicate*(a_1, a_2, . . . , a_n: integers)

location:=0

i:=2

while i≤n and *location*=0

j:=1

while j<i and *location*=0

if a_i=a_j **then** *location*:=i

else j:=j+1

i:=i+1

return *location* {*location* is the subscript of the first value that repeats a previous value in the sequence}

16. procedure *find decrease*(a_1, a_2, . . . , a_n: positive integers)

location := 0

i:=2

while i ≤ n and *location* = 0

if $a_i < a_{i-1}$ **then** *location* := i

else i := i + 1

return *location* {*location* is the subscript of the first value less than the immediately preceding one}

17. At the end of the first pass: 1, 3, 5, 4, 7; at the end of the second pass: 1, 3, 4, 5, 7; at the end of the third pass: 1, 3, 4, 5, 7; at the end of the fourth pass: 1, 3, 4, 5, 7

18. a) 1, 5, 4, 3, 2; 1, 2, 4, 3, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5

b) 1, 4, 3, 2, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5

c) 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5

19. We carry out the linear search algorithm given as Algorithm 2 in this Part 1, except that we replace $x \neq a_i$ by $x<a_i$, and we replace the **else** clause with **else** *location* :=n+1.

20. $2+3+4+\cdots+n=(n^2+n-2)/2$

PART 2. GROWTH OF FUNCTIONS

The time required to solve a problem depends, as it was mentioned in Part 1, on more than only the number of operations it uses. The time also depends on the hardware and software used to run the program that implements the algorithm. However, when we change the hardware and software used to implement an algorithm, we can closely approximate the time required to solve a problem of size n by multiplying the previous time required by a constant. For example, on a supercomputer we might be able to solve a problem of size n a million times faster than we can on a PC. However, this factor of one million will not depend on n, and we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. In other words, we do not have to worry about the hardware and software used to implement an algorithm.

To estimate the growth of time complexity function which depends on input size n we will intensively use the concept of big-O notation, and the related big-Omega and big-Theta notations. We will explain how big-O, big-Omega, and big-Theta estimates are constructed and establish estimates for some important functions that are used in the analysis of algorithms. ■

BIG-O NOTATION

The growth of functions is often described using a special notation. Definition 2 below describes this notation.

Definition 2: 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. [This is read as "f (x) is big-oh of g(x)."] ■

Remark: Intuitively, **the definition that f(x) is O(g(x)) says that f(x) grows slower that some fixed multiple of g(x) as x grows without bound**. **The goal in using big-O notation to estimate functions is to choose a <u>function g(x), as simple as possible,</u> that grows relatively slowly so that f(x) is O(g(x)).**

The constants C and k in the definition of big-O notation are called **witnesses** to the relationship f(x) is O(g(x)). To establish that f(x) is O(g(x)) we need only one pair of witnesses to this relationship.

That is, to show that f (x) is O(g(x)), we need find only *one* pair of constants C and k, the witnesses, such that $|f(x)|\leq C|g(x)|$ whenever $x>k$.

Note that when there is one pair of witnesses to the relationship f (x) is O(g(x)), there are *infinitely many* pairs of witnesses. To see this, note that if C and k are one pair of witnesses, then any pair C' and k', where $C<C'$ and $k<k'$, is also a pair of witnesses, because $|f(x)|\leq C|g(x)|\leq C'|g(x)|$ whenever $x>k'>k$.

Working with the Definition og Big-O Notation.

A useful approach for finding a pair of witnesses is to first select a value of k for which the size of |f(x)| can be readily estimated when x>k and to see whether we can use this estimate to find a value of C for which $|f(x)|\leq C|g(x)|$ if x>k. This approach is illustrated in Example 6.

EXAMPLE 6. Show that $f(x)=x^2+2x+1$ is $O(x^2)$.

Solution: We observe that we can readily estimate the size of f(x) when x>1 because $x<x^2$ and $1<x^2$ when x>1. It follows that

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

whenever x>1. Consequently, we can take C=4 and k=1 as witnesses to show that f (x) is $O(x^2)$. That is, $f(x)=x^2+2x+1<4x^2$ whenever x>1. (Note that it is not necessary to use absolute values here because all functions in these equalities are positive when x is positive.)

Alternatively, we can estimate the size of f(x) when x>2. When x>2, we have $2x\leq x^2$ and $1\leq x^2$. Consequently, if x>2, we have

$$0\leq x^2+2x+1\leq x^2+x^2+x^2=3x^2.$$

It follows that C=3 and k=2 are also witnesses to the relation f (x) is $O(x^2)$. ■

Observe that in statement "f(x) is $O(x^2)$," x^2 *can be replaced by any function with larger values than* x^2. For example, f (x) is $O(x^3)$, f(x) is $O(x^2+x+7)$, and so on.

It is also true that x^2 is $O(x^2+2x+1)$, because $x^2<x^2+2x+1$ whenever x>1. This means that C=1 and k=1 are witnesses to the relationship x^2 is $O(x^2+2x+1)$.

Note that in Example 6 we have two functions, $f(x)=x^2+2x+1$ and $g(x)=x^2$, such that f(x) is O(g(x)) and g(x) is O(f(x)) - the latter fact following from the inequality $x^2\leq x^2+2x+1$, which holds for all nonnegative real numbers x. We say that two functions f(x) and g(x) that satisfy both of these big-O relationships are of the **same order**. We will return to this notion later in this part.

Remark: The fact that f(x) is O(g(x)) is sometimes written **f(x)=O(g(x)).** However, the equals sign in this notation does *not* represent a genuine equality. Rather, this notation tells us that an inequality holds relating the values of the functions f and g for sufficiently large numbers in the domains of these functions. However, it is acceptable to write **$f(x)\in O(g(x))$** because O(g(x)) represents the set of functions that are O(g(x)).

When f(x) is O(g(x)), and h(x) is a function that has larger absolute values than g(x) does for sufficiently large values of x, it follows that f(x) is O(h(x)). In other words, the function g(x) in the relationship f(x) is O(g(x)) can be replaced by a function with larger absolute values. To see this, note that if

$|f(x)|\leq C|g(x)|$ if x>k, and

if $|h(x)|>|g(x)|$ for all x>k,

then $|f(x)|\leq C|h(x)|$ if x>k.

Hence, f(x) is O(h(x)).

When big-O notation is used, the **function g** in the relationship f(x) is O(g(x)) **is chosen to be as small as possible** (sometimes from a set of reference functions, such as functions of the form

x^n, where n is a positive integer).

In subsequent discussions, we will almost always deal with functions that take on only positive values. All references to absolute values can be dropped when working with big-O estimates for such functions.

Example 7 below illustrates how big-O notation is used to estimate the growth of functions.

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

Solution: Note that when $x>7$, we have $7x^2<x^3$. (We can obtain this inequality by multiplying both sides of $x>7$ by x^2.) Consequently, we can take $C=1$ and $k=7$ as witnesses to establish the relationship $7x^2$ is $O(x^3)$. Alternatively, when $x>1$, we have $7x^2<7x^3$, so that both $C=7$ and $k=1$ are also witnesses to the relationship $7x^2$ is $O(x^2)$. ■

Example 8 below illustrates how to show that a big-O relationship does not hold.

EXAMPLE 8. Show that n^2 is not $O(n)$.

Solution: To show that n^2 is not $O(n)$, we must show that no pair of witnesses C and k exist such that $n^2 \leq Cn$ whenever $n>k$. We will use a proof by contradiction to show this.

Suppose that there are constants C and k for which $n^2 \leq Cn$ whenever $n>k$. Observe that when $n>0$ we can divide both sides of the inequality $n^2 \leq Cn$ by n to obtain the equivalent inequality $n \leq C$. However, no matter what C and k are, the inequality $n \leq C$ cannot hold for all n with $n>k$. In particular, once we set a value of k, we see that when n is larger than the maximum of k and C, it is not true that $n \leq C$ even though $n>k$. This contradiction shows that n^2 is not $O(n)$. ■

EXAMPLE 9. Show that x^3 is not $O(7x^2)$.

Solution: To determine whether x^3 is $O(7x^2)$, we need to determine whether witnesses C and k exist, so that $x^3 \leq C(7x^2)$ whenever $x>k$. We will show that no such witnesses exist using a proof by contradiction.

If C and k are witnesses, the inequality $x^3 \leq C(7x^2)$ holds for all $x>k$. Observe that the inequality $x^3 \leq C(7x^2)$ is equivalent to the inequality $x \leq 7C$, which follows by dividing both sides by the positive quantity x^2. However, no matter what C is, it is not the case that $x \leq 7C$ for all $x>k$ no matter what k is, because x can be made arbitrarily large. It follows that no witnesses C and k exist for this proposed big-O relationship. Hence, x^3 is *not* $O(7x^2)$. ■

Theorem 1. Let $f(x)=a_n x^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers. Then $f(x)$ is $O(x^n)$.

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

$$|f(x)|=|a_n x^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0| \leq |a_n|x^n+|a_{n-1}|x^{n-1}+\ldots+|a_1|x+|a_0|=x^n(|a_n|+|a_{n-1}|/x+\ldots+|a_1|/x^{n-1}+|a_0|/x^n) \leq$$
$$\leq x^n(|a_n|+|a_{n-1}|+\ldots+|a_1|+|a_0|).$$

This shows that

$$|f(x)| \leq Cx^n$$

where $C=|a_n|+|a_{n-1}|+\ldots+|a_1|+|a_0|$ whenever $x>1$. Hence, the witnesses $C=|a_n|+|a_{n-1}|+\ldots+|a_1|+|a_0|$ and $k=1$ show that $f(x)$ is $O(x^n)$. ■

We now give some examples involving functions that have the set of positive integers as their domains.

EXAMPLE 10. How can big-O notation be used to estimate the sum of the first n positive integers?

Solution: Because each of the integers in the sum of the first n positive integers does not exceed n, it follows that

$$1+2+\ldots+n \leq n+n+\ldots+n=n^2$$

From this inequality it follows that $1+2+3+\ldots+n$ is $O(n^2)$, taking C=1 and k=1 as witnesses. (In this example the domains of the functions in the big-O relationship are the set of positive integers.) ■

EXAMPLE 11. Give big-O estimates for the following functions:

1. factorial function
2. logarithm of the factorial function

Solution: A big-O estimate for n! can be obtained by noting that each term in the product does not exceed n. Hence,

$$n!=1\cdot 2\cdot 3\cdot\ldots\cdot n\leq n\cdot n\cdot n\cdot\ldots\cdot n=n^n. \quad (1)$$

This inequality shows that

$$\mathbf{n!=O(n^n)} \quad (2)$$

taking C=1 and k=1 as witnesses.

Taking logarithms of both sides of the inequality (1), we obtain

$$\log(n!)\leq\log n^n=n\log n \quad (3)$$

This implies that

$$\mathbf{\log(n!)=O(n\log n)} \quad (4)$$

taking C=1 and k=1 as witnesses. ■

EXAMPLE 12. The following statements are true

$$\mathbf{n=O(2^n)} \quad (5)$$

$$\mathbf{\log n=O(n)} \quad (6)$$

Here $\log n=\log_2 n$.

Solution: Using the inequality $n<2^n$, we quickly can conclude that $\mathbf{n=O(2^n)}$ by taking k=C=1 as witnesses. Note that because the logarithm function is increasing, taking logarithms (base 2) of both sides of this inequality shows that $\log n<n$. It follows that

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

(Again we take C=k=1 as witnesses.) If we have logarithms to a base b, where b is different from 2, we still have $\mathbf{(\log_b n)=O(n)}$ **because**

$$\mathbf{\log_b n}=\frac{\log n}{\log b}<\frac{n}{\log b}=\frac{1}{\log b}n$$

whenever *n* is a positive integer. We take C=1/logb and k=1 as witnesses. ■

Useful Big-O Estimates Involving Logarithms, Powers, and Exponential Functions.

We now give some useful facts that help us determine whether big-O relationships hold between pairs of functions when each of the functions is a power of a logarithm, a power, or an exponential function of the form b^n where b>1.

Theorem 1 shows that if f(n) is a polynomial of degree d, then f(n) is $O(n^d)$. Applying this theorem, we see that **if $1<c<d$, then n^c is $O(n^d)$.** Clear that the reverse of this relationship does not hold (proof by contradiction - the similar arguments as in examples 8 and 9). Putting these facts together, we see that:

if $\mathbf{1<c<d}$, then $\mathbf{n^c=O(n^d)}$**, but** $\mathbf{n^d\neq O(n^c)}$**.**

In Example 12, we demonstrated that n is $O(2^n)$. More generally, we have the following:

Theorem 2. Whenever $\mathbf{d>0}$ and $\mathbf{b>1}$, we have

$$\mathbf{n^d=O(b^n)}, \text{ but } \mathbf{b^n\neq O(n^d)},$$

that is, 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.

Theorem 2 is the corollary of the following simple Lemma from Calculus 1.

Lemma 1. Let f(x) and g(x) are positive-valued functions. The following two statements are valid.

If $\lim_{x\to\infty}\frac{f(x)}{g(x)} = C < \infty$, (C≥0) then f(x)=O(g(x)).

If $\lim_{x\to\infty}\frac{f(x)}{g(x)} = \infty$, then f(x) is not O(g(x)).

Proof. Let $\lim_{x\to\infty}\frac{f(x)}{g(x)} = C < \infty$. Use the definition of limit with ε=1. Then we have

$$\exists k>0 \text{ such that } \left|\frac{f(x)}{g(x)} - c\right| < 1 \text{ whenever } x>k.$$

In other words, $\frac{f(x)}{g(x)} < 1 + C$ or f(x)<(1+C)g(x). Denote C_1=1+C. Now taking as pair of witnesses (k, C_1) we obtain that for all x>k we have f(x)<C_1g(x), hence, f(x)=O(g(x)).

Now let $\lim_{x\to\infty}\frac{f(x)}{g(x)} = \infty$. By definition of infinite limit ∀C>0 ∃k>0 such that $\frac{f(x)}{g(x)}>C$ whenever x>k or f(x)>Cg(x) whenever x>k. Therefore, there is no pair (k, C) of positive numbers which play the role of witnesses in the Definition of big-O. That is, f(x)≠O(g(x)). ■

Note 1. As mentioned above, big-O notation is used to estimate the number of operations needed to solve a problem using a specified procedure or algorithm. The functions used in these estimates often include the following:

$$1, \quad \log n, \quad n, \quad n\log n, \quad n^2, \quad 2^n, \quad n! \qquad (7)$$

Using Calculus 1 it is easy to prove that

$$\lim_{n\to\infty}\frac{1}{\log n} = \lim_{n\to\infty}\frac{\log n}{n} = \lim_{n\to\infty}\frac{n}{n\log n} = \lim_{n\to\infty}\frac{n\log n}{n^2} = \lim_{n\to\infty}\frac{n^2}{2^n} = \lim_{n\to\infty}\frac{2^n}{n!} = 0 \qquad (8)$$ ■

Therefore, based on Lemma 1 we conclude that in the list (1) each function is Big O of the next one:

$1=O(\log n)$; $\log n=O(n)$; $n=O(n\log n)$ $n\log n=O(n^2)$; $n^2=O(2^n)$; $2^n=O(n!)$

***Proof* (Theorem 2).** Let $f(x)=x^d$ and $g(x)=b^x$. Here repeated applications of L'Hôpital's rule shows that $\lim_{x\to\infty}\frac{x^d}{b^x}$=0 (C=0) and $\lim_{x\to\infty}\frac{b^x}{x^d} = \infty$. Therefore, **$n^d=O(b^n)$**, but **$b^n\neq O(n^d)$**, ■

In Example 12 we also showed that $\log_b n$ is O(n) whenever b>1. More generally, whenever **b>1 and c>0, d>0**, the following theorem can be proved

Theorem 3. 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. That is,

$$(\log_b n)^c = O(n^d), \text{ but } n^d \neq (O(\log_b n)^c).$$

Proof. Exercise ■

Theorem 4. Given 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. That is:

1<b<c if and only if **$b^n=O(c^n)$.**

Proof. Exercise ■

The Growth of Combinations of Functions.

Many algorithms are made up of two or more separate sub-procedures. The number of steps used by a computer to solve a problem with input of a specified size using such an algorithm is the sum of the number of steps used by these sub-procedures. To give a big-O estimate for the number of steps needed, it is necessary to find big-O estimates for the number of steps used by each sub-procedure and then combine these estimates.

Big-O estimates of combinations of functions can be provided if care is taken when different big-O estimates are combined. In particular, it is often necessary to estimate the growth of the sum and the product of two functions. What can be said if big-O estimates for each of two functions are known? To see what sort of estimates hold for the sum and the product of two functions, suppose that $f_1(x)=O(g_1(x))$ and $f_2(x)=O(g_2(x))$.

Theorem 5. Suppose that $f_1(x)=O(g_1(x))$, $f_2(x)=O(g_2(x))$. Then $(f_1+f_2)(x)=O(\max(|g_1(x)|, |g_2(x)|))$.

Proof. From the definition of big-O notation, there are constants C_1, C_2, k_1, and 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 $x>k_2$

To estimate the sum of $f_1(x)$ and $f_2(x)$, note that $|(f_1+f_2)(x)|=|f_1(x)+f_2(x)|$ $|f_1(x)|+|f_2(x)|$.

When x is greater than both k_1 and k_2, it follows from the inequalities for $|f_1(x)|$ and $|f_2(x)|$ that

$$|f_1(x)|+|f_2(x)|\leq C_1|g_1(x)|+C_2|g_2(x)|\leq C_1|g(x)|+C_2|g(x)|=(C_1+C_2)|g(x)|=C|g(x)|,$$

where $C=C_1+C_2$ and $g(x)=\max(|g_1(x)|, |g_2(x)|)$. [Here max(a, b) denotes the maximum, or larger, of a and b.]

This inequality shows that $|(f_1+f_2)(x)|\leq C|g(x)|$ whenever $x>k$, where $k=\max(k_1, k_2)$. ■

We often have big-O estimates for f_1 and f_2 in terms of the same function g. In this situation, Theorem 5 can be used to show that $(f_1+f_2)(x)=O(g(x))$, because $\max(g(x), g(x))=g(x)$. This result is stated in Corollary 1.

Corollary 1. Suppose that $f_1(x)=O(g(x))$ and $f_2(x)=O(g(x))$. Then $(f_1+f_2)(x)=O(g(x))$. ■

Theorem 6. Suppose that $f_1(x)=O(g_1(x))$ and $f_2(x)=O(g_2(x))$. Then $(f_1\cdot f_2)(x)=O(g_1(x)g_2(x))$.

Proof. When x is greater than $\max(k_1, k_2)$ it follows that

$|(f_1\cdot f_2)(x)|=|f_1(x)||f_2(x)|\leq C_1|g_1(x)|C_2|g_2(x)|\leq C_1C_2\ |(g_1\ g_2)(x)|\leq C|(g_1\cdot g_2)(x)|$,

where $C=C_1\cdot C_2$. From this inequality, it follows that $f_1(x)f_2(x)$ is $O((g_1\cdot g_2)(x))$, because there are constants C and k, namely, $C=C_1\cdot C_2$ and $k=\max(k_1, k_2)$, such that $|(f_1\cdot f_2)(x)|\leq C|g_1(x)\cdot g_2(x)|$ if $x>k$. ■

The goal in using big-O notation to estimate functions is to choose a function g(x) as simple as possible, that grows relatively slowly so that f (x) is O(g(x)). Examples 13 and 14 illustrate how to use Theorems 5 and 6 to do this. The type of analysis given in these examples is often used in the analysis of the time used to solve problems using computer programs.

EXAMPLE 13. Give a big-O estimate for $f(n)=3n\log(n!)+(n^2+3)\log n$, where n is a positive integer.

Solution: First, the product $3n\log(n!)$ will be estimated. From Example 6 we know that $\log(n!)$ is $O(n\log n)$. Using this estimate and the fact that $3n$ is $O(n)$, Theorem 6 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 5 to combine the two big-O estimates for the products shows that $f(n)=3n\log(n!)+(n^2+3)\log n$ is $O(n^2\log n)$. ■

EXAMPLE 14. Give a big-O estimate for $f(x)=(x+1)\log(x^2+1)+3x^2$.

Solution: First, a big-O estimate for $(x+1)\log(x^2+1)$ will be found. Note that $(x+1)$ is $O(x)$. Furthermore, $x^2+1\leq 2x^2$ when $x>1$. Hence,

$$\log(x^2+1)\leq\log(2x^2)=\log 2+\log x^2=\log 2+2\log x\leq 3\log x,$$

if x>2. This shows that $\log(x^2+1)$ is $O(\log x)$. From Theorem 6 it follows that $(x+1)\log(x^2+1)$ is $O(x\log x)$. Because $3x^2$ is $O(x^2)$, Theorem 5 tells us that f (x) is $O(\max(x\log x, x^2))$. Because $x\log x \leq x^2$, for x>1, it follows that f(x) is $O(x^2)$. ■

BIG-Ω (Omega) and BIG-Θ (Theta) NOTATION

Big-O notation is used extensively to describe the growth of functions, but it has limitations. In particular, when f (x) is O(g(x)), we have an upper bound, in terms of g(x), for the size of f(x) for large values of x. However, big-O notation does not provide a lower bound for the size of f(x) for large x. For this, we use **big-Ω notation**. When we want to give both an upper and a lower bound on the size of a function f(x), relative to a reference function g(x), we use **big-Θ notation**. Both big-**Ω** and big-**Θ** notations were introduced by Donald Knuth in the 1970s. His motivation for introducing these notations was the common misuse of big-O notation when both an upper and a lower bound on the size of a function are needed.

We now define big-**Ω** notation and illustrate its use. After doing so, we will do the same for big-Theta notation.

Definition 3: 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 Ω(g(x)) [read as "f (x) is big-Omega of g(x)" write f(x)=Ω(g(x))] if there are positive constants C and k such that

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

There is a strong connection between big-O and big-Omega notation. In particular, clear that **f(x)=Ω(g(x))** if and only if **g(x)=O(f(x))**.

EXAMPLE 15. The function $f(x)=8x^3+5x^2+7$ is Ω(g(x)), where g(x) is the function $g(x)=x^3$. This is easy to see because $f(x)=8x^3+5x^2+7 \geq 8x^3$ for all positive real numbers x. This is equivalent to saying that $g(x)=x^3$ is $O(8x^3+5x^2+7)$, which can be established directly by turning the inequality around. ■

Often, it is important to know **the order of growth** of a function in terms of some relatively simple reference function such as x^n when n is a positive integer or c^x, where c>1. **Knowing the order of growth requires that we have both an upper bound and a lower bound for the size of the function. That is, given a function f (x), we want a reference function g(x) such that f(x)=O(g(x)) and f(x)=Ω(g(x)).** Big-Theta notation, defined as follows, is used to express both of these relationships, providing both an upper and a lower bound on the size of a function.

Definition 4: 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)=Θ(g(x)) if f(x)=O(g(x)) and f(x)=Ω(g(x)). When f(x)=Θ(g(x)) we say that f is big-Theta of g(x), that f(x) is of *order* g(x), and that f(x) and g(x) are of the *same order*. ■

When f(x)=Θ(g(x)), it is also the case that g(x)=Θ(f (x)). Also note that f(x)=Θ(g(x)) if and only if f(x)=O(g(x)) and g(x)=O(f(x)) (Exercise). Furthermore, note that f(x) is Θ(g(x)) if and only if there are real numbers C_1 and C_2 and a positive real number k such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

whenever x > k. The existence of the constants C_1, C_2, and k tells us that f(x) is Ω(g(x)) and that f (x) is O(g(x)), respectively.

Usually, when big-Theta notation is used, the function g(x) in Θ(g(x)) is a relatively

simple reference function, such as x^n, c^x, logx, and so on, while f(x) can be relatively complicated.

EXAMPLE 16. We showed earlier that the sum of the first n positive integers is $O(n^2)$. Is this sum of order n^2 ?

Solution: Let $f(n)=1+2+3+\ldots+n$. Because we already know that $f(n)=O(n^2)$, to show that f(n) is of order n^2 we need to find a positive constant C such that $f(n)>Cn^2$ for sufficiently large integers n. To obtain a lower bound for this sum, we can ignore the first half of the terms. Summing only the terms greater than $\lceil n/2 \rceil$, we find that:

$1+2+\ldots+n \geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \ldots + n$

$\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \ldots + \lceil n/2 \rceil$

$= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil$

$\geq (n/2)(n/2)$

$= n^2/4.$

Here symbol $\lceil\ \rceil$ means the *ceiling function* which assigns to the real number x the smallest integer that is greater than or equal to x.

This shows that f(n) is $\Omega(n^2)$. We conclude that f(n) is of order n^2, or f(n) is $\Theta(n^2)$. ■

EXAMPLE 17. Show that $3x^2+8x\log x$ is $\Theta(x^2)$.

Solution: Because $0 \leq 8x\log x \leq 8x^2$, it follows that $3x^2+8x\log x \leq 11x^2$ for $x>1$. Consequently, $3x^2+8x\log x$ is $O(x^2)$. Clear that x^2 is $O(3x^2+8x\log x)$. Finally, $3x^2+8x\log x$ is $\Theta(x^2)$. ■

One useful fact is that the leading term of a polynomial determines its order. For example, if $f(x)=3x^5+x^4+17x^3+2$, then f(x) is of order x^5. ■

Theorem 7. Let $f(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers with $a_n \neq 0$. Then f(x) is of order x^n.

Proof. Immediately follows from the Theorem 1 and Definition 4 ■

EXERCISES.

In Exercises 1-7, to establish a big-O relationship, find witnesses C and k such that $|f(x)| \leq C|g(x)|$ whenever $x>k$.

1. Determine whether each of these functions is O(x).

a) $f(x)=10$ **b)** $f(x)=3x+7$

c) $f(x)=x^2+x+1$ **d)** $f(x)=5\log x$

e) $f(x)=\lfloor x \rfloor$ **f)** $f(x)=\lceil x/2 \rceil$

2. Use the definition of "f(x) is O(g(x))" to show that x^4+9x^3+4x+7 is $O(x^4)$.

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

4. Find the least integer n such that f (x) is $O(x^n)$ for each of these functions.

a) $f(x)=2x^3+x^2\log x$ b) $f(x)=3x^3+(\log x)^4$

c) $f(x)=(x^4+x^2+1)/(x^3+1)$ d) $f(x)=(x^4+5\log x)/(x^4+1)$

5. Show that $x^2+4x+17$ is $O(x^3)$ but that x^3 is not $O(x^2+4x+17)$

6. Show that $3x^4+1$ is $O(x^4/2)$ and $x^4/2$ is $O(3x^4+1)$.

7. Show that 2^n is $O(3^n)$ but that 3^n is not $O(2^n)$.

8. Explain what it means for a function to be O(1).

9. Suppose that f(x), g(x), and h(x) are functions such that f(x) is O(g(x)) and g(x) is O(h(x)). Show

that f(x) is O(h(x))

10. Determine whether each of the functions 2^{n+1} and 2^{2n} is $O(2^n)$.

11. Arrange the functions, 1000logn, nlogn, $\sqrt{n}$, 2n!, 2^n, 3^n, and $n^2/1{,}000{,}000$ in a list so that each function is big-O of the next function.

12. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly n(log n) operations and the second algorithm uses exactly $n^{3/2}$ operations. As n grows, which algorithm uses fewer operations?

13. Give as good a big-O estimate as possible for each of these functions.

a) $(n^2+8)(n+1)$ **b)** $(n\log n+n^2)(n^3+2)$ **c)** $(n!+2^n)(n^3+\log(n^2+1))$

14. Give a big-O estimate for each of these functions. For the function g in your estimate that f(x) is O(g(x)), use a simple function g of the smallest order.

a) $n\log(n^2+1)+n^2\log n$ **b)** $(n\log n+1)^2+(\log n+1)(n^2+1)$ **c)** $n^{2\wedge n}+ n^{n\wedge 2}$

15. For the functions below determine whether that function is $\Omega(x^2)$ and whether it is $\Theta(x^2)$

a) $f(x)=17x+11$ **b)** $f(x)=x^2+1000$ **c)** $f(x)=x\log x$

d) $f(x)=x^4/2$ **e)** $f(x)=2^x$ **f)** $f(x)=\lfloor x\rfloor\cdot\lceil x\rceil$

16. Show that nlogn is O(logn!).

17. Let f(x) and g(x) be functions from the set of real numbers to the set of real numbers. We say that the functions f and g are **asymptotic** and write $f(x)\sim g(x)$ if $\lim_{x\to\infty} f(x)/g(x)=1$.

For each of these pairs of functions, determine whether f and g are asymptotic.

a) $f(x)=\log(x^2+1)$, $g(x)=\log x$ **b)** $f(x)=2^{x+3}$, $g(x)=2^{x+7}$

c) $f(x)=2^{2\wedge x}$, $g(x)=2^{x\wedge 2}$ **d)** $f(x)= 2^{x\wedge 2+x+1}$, $g(x)= 2^{x\wedge 2+x}$

Answers

1. The choices of C and k are not unique.

a) C=1, k=10 **b)** C=4, k=7 **c)** No

d) C=5, k=1 **e)** C=1, k=0 **f)** C=1, k=2

2. $x^4+9x^3+4x+7 \le 4x^4$ for all x > 9; witnesses C=4, k=9

3. $(x^2+1)/(x+1)=x-1+2/(x+1)<x$ for all x>1; witnesses C=1, k=1

4. The choices of C and k are not unique.

a) n=3, C=3, k=1 **b)** n=3, C=4, k=1

c) n=1, C=2, k=1 **d)** n=0, C=2, k=1

5. $x^2+4x+17\le 3x^3$ for all x>17, so $x^2+4x+17$ is $O(x^3)$, with witnesses C=3, k=17. However, if x^3 were $O(x^2+4x+17)$, then $x^3\le C(x^2+4x+17)\le 3Cx^2$ for some C, for all sufficiently large x, which implies that $x\le 3C$ for all sufficiently large x, which is impossible. Hence, x^3 is not $O(x^2+4x+17)$.

6. $3x^2+1\le 4\ x^4=8(x^4/2)$ for all x>1, so $3x^4+1$ is $O(x^4/2)$, with witnesses C=8, k=1. Also $x^4/2 \le 3x^4+1$ for all x>0, so $x^4/2$ is $O(3x^4+1)$, with witnesses C=1, k=0.

7. Because $2^n\le 3^n$ for all n>0, it follows that 2^n is $O(3^n)$, with witnesses C=1, k=0. However, if 3^n were $O(2^n)$, then for some C, $3^n\le C2^n$ for all sufficiently large n. This says that $C\ge(3/2)^n$ for all sufficiently large n, which is impossible. Hence, 3^n is not $O(2^n)$.

8. All functions for which there exist real numbers k and C with $|f(x)|\le C$ for x>k. These are the functions f (x) that are bounded for all sufficiently large x.

9. There are constants C_1, C_2, k_1, and k_2 such that $|f(x)| \le C_1|g(x)|$ for all $x>k_1$ and $|g(x)| \le C_2|h(x)|$ for all $x>k_2$. Hence, for $x>\max(k_1, k_2)$ it follows that $|f(x)| \le C_1|g(x)| \le C_1C_2|h(x)|$. This shows that f(x) is O(h(x)).

10. 2^{n+1} is $O(2^n)$; 2^{2n} is not.

11. $1000 \cdot \log n$, $\sqrt{n}$, , $n\log n$, $n^2/1000000$, 2^n, 3^n, $2n!$

12. The algorithm that uses nlogn operations

13. a) $O(n^3)$ **b)** $O(n^5)$ **c)** $O(n^3 \cdot n!)$

14. a) $O(n^2 \log n)$ **b)** $O(n^2(\log n)^2)$ **c)** $O(n^{2\wedge n})$

15. a) Neither $\Theta(x^2)$ nor $\Omega(x^2)$ **b)** $\Theta(x^2)$ and $\Omega(x^2)$ **c)** Neither $\Theta(x^2)$ nor $\Omega(x^2)$

d) $\Omega(x^2)$, but not $\Theta(x^2)$ **e)** $\Omega(x^2)$, but not $\Theta(x^2)$ **f)** $\Omega(x^2)$ and $\Theta(x^2)$

16. It can be easily shown (by induction on n, for example) that $(n-i)(i+1) \ge n$ for i=0, 1, ..., n−1. Hence, $(n!)^2=(n \cdot 1)((n-1) \cdot 2) \cdot ((n-2) \cdot 3) \cdots (2 \cdot (n-1)) \cdot (1 \cdot n) \ge n^n$. Therefore, $2\log(n!) \ge n\log n$.

Important Note. From Example 11 (Part 2) we know that (logn!) is O(nlogn), therefore, we establish that (**logn!) is Θ(nlogn) or (logn!) and (nlogn) are of the same order.**

17. All are not asymptotic.

PART 3. SEARCHING AND SORTING ALGORITHMS AND COMPLEXITY ISSUES. (For additional reading)

In this Part we continue to analyze searching algorithms and their complexity. We also make the similar discussions for sorting algorithms computation.

SEARCHING ALGORITHMS II.

THE BINARY SEARCH. In Part 1 we discussed the Linear Search (LS) algorithm. We will now consider another searching algorithm which is called the **binary search algorithm (BS)**. BS algorithm can be used when the list has terms occurring in order of increasing size (for instance: if the terms are numbers, they are listed from smallest to largest; if they are words, they are listed in lexicographic, or alphabetic, order).

BS algorithm proceeds by comparing the element to be located to the middle term of the list. The list is then split into two smaller sublists of the same size, or where one of these smaller lists has one fewer term than the other. The search continues by restricting the search to the appropriate sublist based on the comparison of the element to be located and the middle term. It will be shown that the BS algorithm is much more efficient than the LS algorithm. Example 18 demonstrates how BS works.

EXAMPLE 18. To search for 19 in the list: 1, 2, 3, 5, 6, 7, 8, 10, 12, 13, 15, 16, 18, 19, 20, 22, first split this list, which has 16 terms, into two smaller lists with eight terms each, namely,

1, 2, 3, 5, 6, 7, 8, 10 12, 13, 15, 16, 18, 19, 20, 22.

Then, compare 19 and the largest term in the first list. Because 10 < 19, the search for 19 can be restricted to the list containing the 9th through the 16th terms of the original list. Next, split this list, which has eight terms, into the two smaller lists of four terms each, namely,

12, 13, 15, 16 18, 19, 20, 22.

Because 16<19 (comparing 19 with the largest term of the first list) the search is restricted to the second of these lists, which contains the 13th through the 16th terms of the original list. The list 18 19 20 22 is split into two lists, namely,

18, 19 20, 22.

Because 19 is not greater than the largest term of the first of these two lists, which is also 19, the search is restricted to the first list: 18 19, which contains the 13th and 14th terms of the original list. Next, this list of two terms is split into two lists of one term each: 18 and 19. Because 18 < 19, the search is restricted to the second list: the list containing the 14th term of the list, which is 19. Now that the search has been narrowed down to one term, a comparison is made, and 19 is located as the 14th term in the original list. ■

Description of Binary Search Algorithm. We now specify the steps of the binary search algorithm. To search for the integer x in the list $a_1, a_2, \ldots, a_n$, where $a_1<a_2<\ldots<a_n$, begin by comparing x with the middle term a_m of the list, where $m=\lfloor(n+1)/2\rfloor$. (Recall that $\lfloor x\rfloor$ is the greatest integer not exceeding x.) If $x>a_m$, the search for x is restricted to the second half of the list, which is $a_{m+1}, a_{m+2}, \ldots, a_n$. If x is not greater than a_m, the search for x is restricted to the first half of the list, which is $a_1, a_2, \ldots, a_m$. The search has now been restricted to a list with no more than $\lceil n/2\rceil$ elements. (Recall that $\lceil x\rceil$ is the smallest integer greater than or equal to x.) Using the same procedure, compare x to the middle term of the restricted list. Then restrict the search to the first or second half of the list. Repeat this process until a list with one term is obtained. Then determine whether this term is x. ■

Pseudocode for the binary search algorithm is displayed as Algorithm 3.

ALGORITHM 3. The Binary Search Algorithm

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)

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

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

while i<j

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

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

    else j := m

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

else *location* :=0

return *location*{*location* is the subscript i of the term a_i equal to x, or 0 if x is not found} ■

Algorithm 3 proceeds by successively narrowing down the part of the sequence being searched. At any given stage only the terms from a_i to a_j are under consideration. In other words, i and j are the smallest and largest subscripts of the remaining terms, respectively. Algorithm 3 continues narrowing the part of the sequence being searched until only one term of the sequence remains. When this is done, a comparison is made to see whether this term equals x.

Theorem 8. The worst-case time complexity of the binary search algorithm (specified as Algorithm 3) in terms of the number of comparisons used (and ignoring the time required to compute $m=\lfloor(i+j)/2\rfloor$ in each iteration of the loop in the algorithm) is $\Theta(\log n)$.

Proof. For simplicity, assume there are $n=2^k$ elements in the list $a_1, a_2, \ldots, a_n$, where k is a nonnegative integer. Note that k=logn. (If n, the number of elements in the list, is not a power of 2, the list can be considered part of a larger list with 2^{k+1} elements, where $2^k<n<2^{k+1}$. Here 2^{k+1} is the smallest power of 2 larger than n.)

At each stage of the algorithm, i and j, the locations of the first term and the last term of the restricted list at that stage, are compared to see whether the restricted list has more than one term. If i < j, a comparison is done to determine whether x is greater than the middle term of the restricted list.

At the first stage the search is restricted to a list with 2^{k-1} terms. So far, two comparisons have been

used. This procedure is continued, using two comparisons at each stage to restrict the search to a list with half as many terms. In other words, two comparisons are used at the first stage of the algorithm when the list has 2^k elements, two more when the search has been reduced to a list with 2^{k-1} elements, two more when the search has been reduced to a list with 2^{k-2} elements, and so on, until two comparisons are used when the search has been reduced to a list with $2^1=2$ elements. Finally, when one term is left in the list, one comparison tells us that there are no additional terms left, and one more comparison is used to determine if this term is x.

Hence, at most $2k+2=2\log n+2$ comparisons are required to perform a binary search when the list being searched has 2^k elements. (If n is not a power of 2, the original list is expanded to a list with 2^{k+1} terms, where $k=\lfloor \log n \rfloor$, and the search requires at most $2\lceil \log n \rceil+2$ comparisons). It follows that in the worst case, binary search requires O(log n) comparisons. Note that in the worst case, $2\log n+2$ comparisons are used by the binary search. Hence, the binary search uses $\Theta(\log n)$ comparisons in the worst case, because $2\log n+2 = \Theta(\log n)$. ■

From Theorem 8 it follows that in the worst case, the binary search algorithm is more efficient than the linear search algorithm, because we know by Example 5 that the linear search algorithm has $\Theta(n)$ worst-case time complexity.

SORTING ALGORITHMS.

Ordering the elements of a list is a problem that occurs in many contexts. Suppose that we have a list of elements of a set.

Sorting is putting these elements into a list in which the elements are in increasing order. For instance, sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9. Sorting the list *d*, *h*, *c*, *a*, *f* (using alphabetical order) produces the list *a*, *c*, *d*, *f*, *h*. An amazingly large percentage of computing resources is devoted to sorting one thing or another.

There are many reasons why sorting algorithms interest computer scientists and mathematicians. Among these reasons are that some algorithms are easier to implement, some algorithms are more efficient, some algorithms take advantage of particular computer architectures, and some algorithms are particularly clever. In this Part of the lecture we will introduce two sorting algorithms, the bubble sort and the insertion sort.

THE BUBBLE SORT. The **bubble sort** is one of the simplest sorting algorithms, but not one of the most efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete. Pseudocode for the bubble sort is given as Algorithm 4. We can imagine the elements in the list placed in a column. In the bubble sort, the smaller elements "bubble" to the top as they are interchanged with larger elements. The larger elements "sink" to the bottom. This is illustrated in Example 19.

EXAMPLE 19. Use the bubble sort to put 3, 2, 4, 1, 5 into increasing order.

Solution: The steps of this algorithm are illustrated in Figure 1.

Begin by comparing the first two elements, 3 and 2. Because 3>2, interchange 3 and 2, producing the list 2, 3, 4, 1, 5. Because 3<4, continue by comparing 4 and 1. Because 4>1, interchange 1 and 4, producing the list 2, 3, 1, 4, 5. Because 4<5, the first pass is complete. The first pass guarantees that the largest element, 5, is in the correct position.

The second pass begins by comparing 2 and 3. Because these are in the correct order, 3 and 1 are compared. Because 3>1, these numbers are interchanged, producing 2, 1, 3, 4, 5. Because 3<4, these numbers are in the correct order. It is not necessary to do any more comparisons for this pass

because 5 is already in the correct position. The second pass guarantees that the two largest elements, 4 and 5, are in their correct positions.

The third pass begins by comparing 2 and 1. These are interchanged because 2>1, producing 1, 2, 3, 4, 5. Because 2<3, these two elements are in the correct order. It is not necessary to do any more comparisons for this pass because 4 and 5 are already in the correct positions. The third pass guarantees that the three largest elements, 3, 4, and 5, are in their correct positions.

The fourth pass consists of one comparison, namely, the comparison of 1 and 2. Because 1<2, these elements are in the correct order. This completes the bubble sort. ■

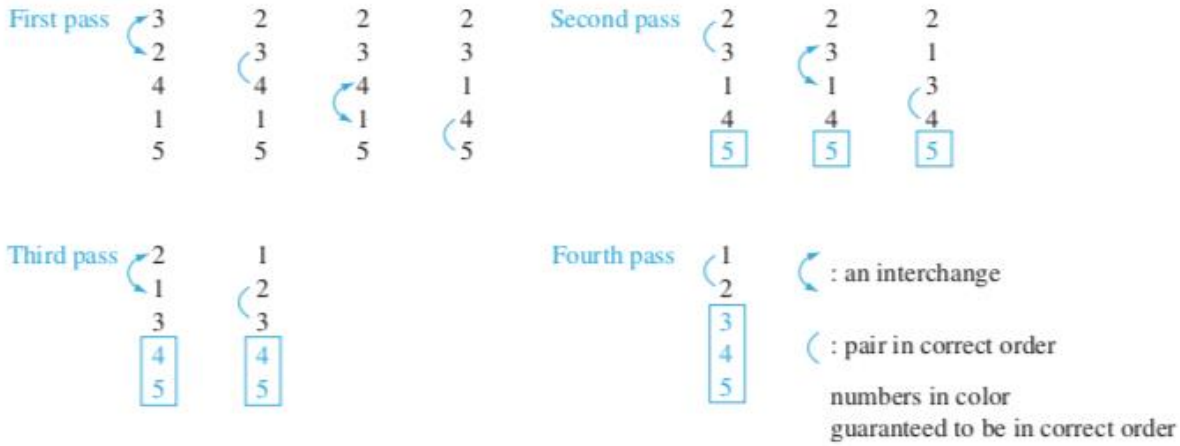

Figure 1. **The Steps of a Bubble Sort.**

ALGORITHM 4. The Bubble Sort.

procedure *bubblesort*(a_1, …, 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, ..., a_n is in increasing order} ■

Next Theorem 9 is devoted to finding the worst-case complexity of the bubble sort in terms of the number of comparisons made.

Theorem 9. The worst-case complexity of the bubble sort in terms of the number of comparisons made is $\Theta(n^2)$.

Proof: The bubble sort sorts a list by performing a sequence of passes through the list. During each pass the bubble sort successively compares adjacent elements, interchanging them if necessary. When the i-th pass begins, the (i−1) largest elements are guaranteed to be in the correct positions. During this pass, (n−i) comparisons are used. Consequently, the total number of comparisons used by the bubble sort to order a list of n elements is

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

Note that the bubble sort always uses this many comparisons, because it continues even if the list becomes completely sorted at some intermediate step. Consequently, the bubble sort uses (n−1)n/2 comparisons, so it has $\Theta(n^2)$ worst-case complexity in terms of the number of comparisons used. ■

THE INSERTION SORT. The **insertion sort** is a simple sorting algorithm, but it is usually not the most efficient. To sort a list with n elements, the insertion sort begins with the second element. The insertion sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element and after the first element if it exceeds the first element. At this point, the first two elements are in the correct order. The third element is then compared with

the first element, and if it is larger than the first element, it is compared with the second element; it is inserted into the correct position among the first three elements an so on.

EXAMPLE 20. Use the insertion sort to put the elements of the list 3, 2, 4, 1, 5 in increasing order.

Solution: The insertion sort first compares 2 and 3. Because 3>2, it places 2 in the first position, producing the list 2, 3, 4, 1, 5 (the sorted part of the list is shown in color). At this point, 2 and 3 are in the correct order. Next, it inserts the third element, 4, into the already sorted part of the list by making the comparisons 4>2 and 4>3. Because 4>3, 4 remains in the third position. At this point, the list is 2, 3, 4, 1, 5 and we know that the ordering of the first three elements is correct. Next, we find the correct place for the fourth element, 1, among the already sorted elements, 2, 3, 4. Because 1<2, we obtain the list 1, 2, 3, 4, 5. Finally, we insert 5 into the correct position by successively comparing it to 1, 2, 3, and 4. Because 5>4, it stays at the end of the list, producing the correct order for the entire list. ■

In general, in the jth step of the insertion sort, the jth element of the list is inserted into the correct position in the list of the previously sorted (j-1) elements. To insert the j th element in the list, a linear search technique is used; the jth element is successively compared with the already sorted (j–1) elements at the start of the list until the first element that is not less than this element is found or until it has been compared with all (j–1) elements; the j th element is inserted in the correct position so that the first j elements are sorted. The algorithm continues until the last element is placed in the correct position relative to the already sorted list of the first (n–1) elements. ■

The insertion sort is described in pseudocode in Algorithm 5.

ALGORITHM 5. The Insertion Sort.

procedure *insertion sort*(a_1, a_2, …, a_n:

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

{a_1, ..., a_n is in increasing order} ■

Theorem 10. The worst-case complexity of the insertion sort in terms of the number of comparisons made is $\Theta(n^2)$.

Proof: The insertion sort inserts the jth element into the correct position among the first (j−1) elements that have already been put into the correct order. It does this by using a linear search technique, successively comparing the j th element with successive terms until a term that is greater than or equal to it is found or it compares a_j with itself and stops because a_j is not less than itself. Consequently, in the worst case, j comparisons are required to insert the jth element into the correct position. Therefore, the total number of comparisons used by the insertion sort to sort a list of n elements is

$$2+3+\ldots+n=\frac{(n+2)(n-1)}{2}=\frac{(n+2)(n-1)}{2}=\frac{n^2+n-2}{2}=\frac{n(n+1)}{2}-1$$

Note that the insertion sort may use considerably fewer comparisons if the smaller elements started out at the end of the list. We conclude that the insertion sort has worst-case complexity $\Theta(n^2)$. ■

In Examples 19 and 20 we demonstrated that both the bubble sort and the insertion sort have

worst-case time complexity $\Theta(n^2)$. However, the most efficient sorting algorithms can sort n items in O(nlogn) time. From this point on, we will assume that sorting n items can be done in O(nlogn). ■

Complexity of Dijkstra's Algorithm

We can now estimate the computational complexity of Dijkstra's algorithm in terms of additions and comparisons. Here we remind the Dijkstra's algorithm from one of previous lectures for the completeness and convenience.

ALGORITHM 1. Dijkstra's Algorithm.

procedure *Dijkstra*(G: weighted connected simple graph, with all weights positive)
{G has vertices a=v_0, v_1, ... , v_n=z and lengths $w(v_i, v_j)$ where $w(v_i, v_j)=\infty$ if $\{v_i, v_j\}$ is not an edge in G}
for i:=1 **to** n
    L(v_i):=∞
L(*a*):= 0
S:= ∅
{the labels are now initialized so that the label of *a* is 0 and all other labels are ∞, and S={∅} }
while z∉S
    u:= a vertex not in S with L(u) minimal
    S:= S∪{u}
    for all vertices *v* not in S
        if L(u) + *w*(u, *v*) < L(*v*) **then** L(*v*) := L(u) + *w*(u, *v*)
        {this adds a vertex to S with minimal label and updates the labels of vertices not in S}
return L(z) {L(z) = length of a shortest path from a to z} ■

Theorem 11: Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) to find the length of a shortest path between two vertices in a connected simple undirected weighted graph with n vertices.

Proof. The algorithm uses no more than (n–1) iterations where n is the number of vertices in the graph, because one vertex is added to the distinguished set at each iteration. We are done if we can estimate the number of operations used for each iteration. We can identify the vertex not in S_k with the smallest label using no more than (n–1) comparisons. Then we use an addition and a comparison to update the label of each vertex not in S_k. It follows that no more than 2(n−1) operations are used at each iteration, because there are no more than (n-1) labels to update at each iteration. Because we use no more than (n–1) iterations, each using no more than 2(n−1) operations. ■

CONCLUDING REMARKS

Note that a big-Θ estimate of the time complexity of an algorithm expresses how the time required to solve the problem increases as the input grows in size.

In practice, the best estimate (that is, with the smallest reference function) that can be shown is used. However, big-Θ estimates of time complexity cannot be directly translated into the actual amount of computer time used. One reason is that a big-Θ estimate f(n) is Θ(g(n)), where f(n) is the time complexity of an algorithm and g(n) is a reference function, means that $C_1g(n) \le f(n) \le C_2g(n)$ when n>k, where C_1, C_2, and k are constants. Thus, without knowing the constants C_1, C_2, and k in the inequality, this estimate cannot be used to determine a lower bound and an upper bound on the number of operations used in the worst case. As remarked before, the time required for an operation depends on the type of operation and the computer being used. Often, instead of a big-Θ estimate on the worst-case time complexity of an algorithm, we have only a big-O estimate. Note that a big-O estimate on the time complexity of an algorithm provides an upper, but not a lower, bound on the worst-case time required for the algorithm as a function of the input size. Nevertheless, for simplicity,

we will often use big-O estimates when describing the time complexity of algorithms, with the understanding that big-Θ estimates would provide more information.

Table 2 below displays the time needed to solve problems of various sizes with an algorithm using the indicated number n of bit operations, if each bit operation takes 10^{-11} seconds, a reasonable estimate of the time required for a bit operation using the fastest computers available today. Times of more than 10^{100} years are indicated with an asterisk. In the future, these times will decrease as faster computers are developed. We can use the times shown in Table 2 to see whether it is reasonable to expect a solution to a problem of a specified size using an algorithm with known worst-case time complexity when we run this algorithm on a modern computer. Note that we cannot determine the exact time a computer uses to solve a problem with input of a particular size because of a myriad of issues involving computer hardware and the particular software implementation of the algorithm.

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

It is important to have a reasonable estimate for how long it will take a computer to solve a problem. For instance, if an algorithm requires approximately 10 hours, it may be worthwhile to spend the computer time (and money) required to solve this problem. But, if an algorithm requires approximately 10 billion years to solve a problem, it would be unreasonable to use resources to implement this algorithm. One of the most interesting phenomena of modern technology is the tremendous increase in the speed and memory space of computers. Another important factor that decreases the time needed to solve problems on computers is **parallel processing**, which is the technique of performing sequences of operations simultaneously.

Efficient algorithms, including most algorithms with polynomial time complexity, benefit most from significant technology improvements. However, these technology improvements offer little help in overcoming the complexity of algorithms of exponential or factorial time complexity. Because of the increased speed of computation, increases in computer memory, and the use of algorithms that take advantage of parallel processing, many problems that were considered impossible to solve 2-3 years ago are now routinely solved, and certainly 2-3 years from now this statement will still be true.

EXERCISES

1. Give a big-O estimate for the number of operations (where an operation is an addition or a multiplication) used in this segment of an algorithm.

```
t := 0
for i := 1 to 3
        for j :=1 to 4
        t :=t+ij
```

2. Give a big-O estimate for the number of operations, where an operation is a comparison or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the **for** loops, where $a_1, a_2, ..., a_n$ are positive real numbers).

m := 0

for i := 1 **to** n

for j :=i+1 **to** n

m := max(a_i a_j , m)

3. Suppose that an element is known to be among the first four elements in a list of 32 elements. Would a linear search or a binary search locate this element more rapidly?

4. What is the effect in the time required to solve a problem when you increase the size of the input from *n* to (*n*+1), assuming that the number of milliseconds the algorithm uses to solve the problem with input size *n* is the following function? [Express your answer in the simplest form possible, either as a ratio or a difference. Your answer may be a function of n or a constant.]

a) logn **b)** 100n **c)** n^2

d) n^3 **e)** 2^n **f)** n!

g) 2^{n^2}

5. Analyze the worst-case time complexity of the algorithm you devised in Exercise 14 of Part 1 for locating a mode in a list of nondecreasing integers.

6. Analyze the worst-case time complexity of the algorithm you devised in Exercise 15 of Part 1 for finding the first term of a sequence of integers equal to some previous term.

7. Analyze the worst-case time complexity of the algorithm you devised in Exercise 16 of Part 1 for finding the first term of a sequence less than the immediately preceding term.

8. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 5 in Part 1 for determining whether a string of n characters is a palindrome.

Answers

1. O(1)

2. $O(n^2)$

3. Linear

4. a) Less than 1 millisecond more **b)** 100 milliseconds more

c) 2n+1 milliseconds more **d)** $3n^2+3n+1$ milliseconds more

e) Twice as much time **f) (**n+1) times as many milliseconds

g) 2^{2n+1} times as many milliseconds

5. O(n)

6. $O(n^2)$

7. O(n)

8. O(n)

Supplementary Exercises.

1. a) Describe an algorithm for locating the last occurrence of the largest number in a list of integers.

b) Estimate the number of comparisons used.

2. a) Give an algorithm to determine whether a bit string contains a pair of consecutive zeros.

b) How many comparisons does the algorithm use?

3. a) Adapt Algorithm 1 in Part 1 to find the maximum and the minimum of a sequence of n elements by employing a temporary maximum and a temporary minimum that is updated as each successive element is examined.

b) Describe the algorithm from part (a) in pseudocode.

c) How many comparisons of elements in the sequence are carried out by this algorithm? (Do not count comparisons used to determine whether the end of the sequence has been reached.)

4. Show that the worst-case complexity in terms of comparisons of an algorithm that finds the maximum and minimum of n elements is at least $\lceil 3n/2 \rceil - 2$.

5. Devise an algorithm that finds all equal pairs of sums of two terms of a sequence of *n* numbers and determine the worst-case complexity of your algorithm.

6. Show that $(n \log n + n^2)^3$ is $O(n^6)$.

7. Give a big-O estimate for $(x^2 + x(\log x)^3) \cdot (2^x + x^3)$.

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

9. Find all pairs of functions of the same order in this list of functions:

$(n^2 + (\log n)^2)$, $(n^2 + n)$, $(n^2 + \log 2^n + 1)$, $((n+1)^3 - (n-1)^3)$, and $(n + \log n)^2$.

10. Find an integer n with $n > 2$ for which $n^{2^{100}} < 2^n$.

11. Arrange the functions n^n, $(\log n)^2$, $n^{1.0001}$, $(1.0001)^n$, and $n(\log n)^{1001}$ in a list so that each function is big-O of the next function. [*Hint:* To determine the relative size of some of these functions, take logarithms.]

Answers for Supplementary Exercises

1. a) procedure *last max*($a_1, \ldots, a_n$: integers)

```
max := a1
last := 1
i := 2
while i ≤ n
    if ai ≥ max then
        max:= ai
        last:= i
    i := i + 1
return last
```

b) $2n - 1 = O(n)$ comparisons

2. a) procedure *pair zeros*($b_1\, b_2 \ldots b_n$: bit string, $n \geq 2$)

```
x := b1
y := b2
k := 2
    while k < n and (x≠0 or y≠0)
        k := k + 1
        x := y
        y := bk
if x = 0 and y = 0 then print "YES"
else print "NO"
```

b) $O(n)$

3. a) and **b)**

procedure *smallest and largest*(a_1, a_2, . . . , a_n: integers)

min := a_1

max := a_1

for i := 2 **to** n

if a <*min* **then** *min*:= a_i

if a_i >*max* **then** *max*:=a_i

{*min* is the smallest integer among the input, and *max* is the largest}

c) 2n−2

4. Before any comparisons are done, there is a possibility that each element could be the maximum and a possibility that it could be the minimum. This means that there are 2n different possibilities, and 2n−2 of them have to be eliminated through comparisons of elements, because we need to find the unique maximum and the unique minimum. We classify comparisons of two elements as "virgin" or "nonvirgin," depending on whether or not both elements being compared have been in any previous comparison. A virgin comparison eliminates the possibility that the larger one is the minimum and that the smaller one is the maximum; thus each virgin comparison eliminates two possibilities, but it clearly cannot do more. A nonvirgin comparison must be between two elements that are still in the running to be the maximum or two elements that are still in the running to be the minimum, and at least one of these elements must *not* be in the running for the other category. For example, we might be comparing x and y, where all we know is that x has been eliminated as the minimum. If we find that x>y in this case, then only one possibility has been ruled out—we now know that y is not the maximum. Thus in the worst case, a nonvirgin comparison eliminates only one possibility. (The cases of other nonvirgin comparisons are similar.) Now there are at most ⌊n/2⌋ comparisons of elements that have not been compared before, each removing two possibilities; they remove 2⌊n/2⌋ possibilities altogether. Therefore we need 2n−2−2⌊n/2⌋ more comparisons that, as we have argued, can remove only one possibility each, in order to find the answers in the worst case, because 2n−2 possibilities have to be eliminated. This gives us a total of 2n−2−2⌊n/2⌋+⌊n/2⌋ comparisons in all. But 2n−2−2⌊n/2⌋+⌊n/2⌋=2n−2−⌊n/2⌋=2n−2+⌈−n/2⌉**=** =⌈2n−n/2⌉−2=⌈3n/2⌉−2, as desired.

5. The following algorithm has worst-case complexity $O(n^4)$.

procedure *equal sums*(a_1, a_2, . . . , a_n)

for i :=1 to n

for j :=i+1 **to** n {since we want i < j}

for k :=1 **to** n

for l :=k+1 **to** n {since we want k < l}

if $a_i + a_j = a_k + a_l$ and (i,j)≠(k,l)

then output these pairs

6. Because log n < n, we have $(n\log n + n^2)^3 \le (n^2 + n^2)^3 \le (2n^2)^3 = 8n^6$ for all n > 0. This proves that $(n \log n + n^2)^3$ is $O(n^6)$, with witnesses C = 8 and k = 0.

7. $O(x^2 2^x)$

8. Note that $\frac{n!}{2^n} = \frac{n}{2}\frac{n-1}{2} \dots \frac{3}{2}\frac{2}{2}\frac{1}{2} > \frac{n}{2} 1 \cdot 1 \cdot 1 \cdot \dots \cdot 1 \cdot \frac{1}{2} = \frac{n}{4}$

9. All of these functions are of the same order.

10. 2^{107}

11. $(\log n)^2$, $n(\log n)^{1001}$, $n^{1.0001}$, 1.0001^n , n^n

Key Terms and Results

Terms

- **algorithm:** a finite sequence of precise instructions for performing a computation or solving a problem
- **searching algorithm:** the problem of locating an element in a list
- **linear search algorithm:** a procedure for searching a list element by element
- **binary search algorithm:** a procedure for searching an ordered list by successively splitting the list in half
- **sorting:** the reordering of the elements of a list into prescribed order
- ***f* (*x*) is O(*g*(*x*)):** the fact that $|f(x)| \leq C|g(x)|$ for all $x>k$ for some constants C and k
- **witness to the relationship *f*(*x*) is O(*g*(*x*)):** a pair C and k such that $|f(x)| \leq C\,|g(x)|$ whenever $x>k$
- ***f* (*x*) is Ω(*g*(*x*)):** the fact that $|f(x)| \geq C|g(x)|$ for all $x>k$ for some positive constants C and k
- ***f* (*x*) is Θ(*g*(*x*)):** the fact that f(x) is both O(g(x)) and Ω(g(x))
- **time complexity:** the amount of time required for an algorithm to solve a problem
- **space complexity:** the amount of space in computer memory required for an algorithm to solve a problem
- **worst-case time complexity:** the greatest amount of time required for an algorithm to solve a problem of a given size
- **average-case time complexity:** the average amount of time required for an algorithm to solve a problem of a given size
- **algorithmic paradigm:** a general approach for constructing algorithms based on a particular concept
- **brute force:** the algorithmic paradigm based on constructing algorithms for solving problems in a naive manner from the statement of the problem and definitions
- **greedy algorithm:** an algorithm that makes the best choice at each step according to some specified condition
- **tractable problem:** a problem for which there is a worst-case polynomial-time algorithm that solves it
- **intractable problem:** a problem for which no worst-case polynomial-time algorithm exists for solving it
- **solvable problem:** a problem that can be solved by an algorithm
- **unsolvable problem:** a problem that cannot be solved by an algorithm

Results

- **linear and binary search algorithms:**
- **bubble sort:** a sorting that uses passes where successive items are interchanged if they in the wrong order
- **insertion sort:** a sorting that at the jth step inserts the jth element into the correct position in in the list, when the first $j-1$ elements of the list are already sorted
- **The linear search has O(n)** worst case time complexity.
- **The binary search has O(log n)** worst case time complexity.
- **The bubble and insertion sorts have $O(n^2)$** worst case time complexity.
- **log n! is O(n log n).**
- **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 $O(\max(g_1(x), g_2(x)))$ and $(f_1f_2)(x)$ is $O((g_1g_2(x))$**
- **If $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$, and $P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$ then $P_n(x)$ is $\Theta(x^n)$, and hence O(n) and Ω(n).**