Mercredi 26 juillet 2023

3 Algorithms

3.1 Algorithms

When presented with a problem that arise in discrete mathematics, 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 **set, sequences and functions** (structures discussed in chap. 2) as well as such other structures as **permutations, relations, graphs, trees, networks and finite state machines** (concepts that will be discussed in later chapters).

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**.

Properties of algorithms

There are several properties that algorithms generally share:

(I.) **Input**: input values from a specified set.

(II.) **Output**: output values are the solution to the problem.

(III.) **Definiteness**: The steps must be defined precisely.

(IV.) **Correctness**: It should produce correct output values for each set of input values.

(V.) **Finiteness**: It should produce the desired output after a finite number of steps.

(VI.) **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.

(VII.) **Generality**: It should be applicable for all problems of the desired form, not just a particular set of input values.

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

In this section we will use the problem of (I.) finding the largest integer in a finite sequence of integers
(II.) locating a particular element in a finite set
to illustrate the concept of an algorithm and the properties algorithms have.
In subsequent sections, procedures for finding the greatest common divisor of two integers, for finding the shortest path between two points in a network, for multiplying matrices, and so on..., will be discussed.

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

For instance, a university may need to find the highest score on a competitive exam taken by thousands of students. Or a sports organization may want to identify the member with the highest rating each month.

Pseudocode provides an intermediate step between an English language description of an algorithm and an implementation of this algorithm in a programming language.

ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

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

i=n max<a_i exit the loop when i=n+1
Exactly 2(n-1)+1 = 2n-1 comparisons
Hence, Θ(n)

(I.) Linear search (sequential search)

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ and $x \ne a_i$)
  $i := i + 1$
if $i \le n$ **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}

if x=a_i then 2i+1 comparisons
When the element is not in the list, 2n+2 comparisons
Hence, Θ(n)

(II.) Binary Search

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}

n=2^k elements ⇔ k=log n
2k+2 = 2log n+2 comparisons
Hence, Θ(log n)

Searching Algorithms

For instance, a program that checks the spelling of words searches for them in a dictionary, which is just an ordered list of words.

Sorting Algorithms

For example, to produce a telephone directory it is necessary to alphabetize the names of subscribers. Similarly, producing a directory of songs available for downloading requires that their titles be put in alphabetic order. Putting addresses in order in an e-mail mailing list can determine whether there are duplicated addresses. Creating a useful dictionary requires that words be put in alphabetical order. Similarly, generating a parts list requires that we order them according to increasing part number.

(I.) Bubble Sort (Simplest sorting algo, but not one of the most efficient)

ALGORITHM 4 The Bubble Sort.

procedure *bubblesort*($a_1, \ldots, a_n$: real numbers with $n \ge 2$)
for $i := 1$ **to** $n - 1$
  for $j := 1$ **to** $n - i$
    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
{$a_1, \ldots, a_n$ is in increasing order}

(n-1)+(n-2)+...+2+1 = (n-1)n/2 comparisons
Hence, Θ(n^2)

(II.) Insertion Sort (simple, but not the most efficient)

ALGORITHM 5 The Insertion Sort.

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: real numbers with $n \ge 2$)
for $j := 2$ **to** n — Begin with the second element
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$
  for $k := 0$ **to** $j - i - 1$
    $a_{j-k} := a_{j-k-1}$
  $a_i := m$
{$a_1, \ldots, a_n$ is in increasing order}

2+3+...+n = n(n+1)/2 - 1
Hence, Θ(n^2)

String Matching

String matching plays an essential role in a wide variety of applications, including text editing, spam filters, systems that look for attacks in a computer network, search engines, plagiarism detection, bioinformatics, and many other important applications.

(I.) Naive String Matcher (a brute force algorithm)

ALGORITHM 6 Naive String Matcher.

procedure *string match* (n, m: positive integers, $m \le n, t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m$: characters)
  for $s := 0$ **to** $n - m$
    $j := 1$
    while ($j \le m$ and $t_{s+j} = p_j$)
      $j := j + 1$
    if $j > m$ **then print** "s is a valid shift"

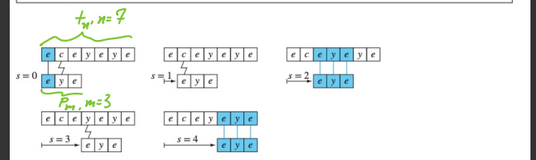

FIGURE 2 The steps of the naive string matcher with $P = eye$ in $T = eceyeye$. Matches are identified with a solid line and mismatches with a jagged line. The algorithm finds two valid shifts, $s = 2$ and $s = 4$.

Greedy Algorithms

Design to solve optimization problems. It makes the best choice at each step according to a specified criterion.

(I.) Cashier's algorithm

We display the cashier's algorithm for n cents, using any set of denominations of coins, as Algorithm 7.

ALGORITHM 7 Cashier's Algorithm.

procedure *change*($c_1, c_2, \ldots, c_r$: values of denominations of coins, where $c_1 > c_2 > \cdots > c_r$; n: a positive integer)
for $i := 1$ **to** r
  $d_i := 0$ {d_i counts the coins of denomination c_i used}
  while $n \ge c_i$
    $d_i := d_i + 1$ {add a coin of denomination c_i}
    $n := n - c_i$
{d_i is the number of coins of denomination c_i in the change for $i = 1, 2, \ldots, r$}

Consider the problem of making n cents change with quarters, dimes, nickels and pennies, and using the least total number of coins.

Proof by contradiction

Lemma 1. If n is a positive int, then n cents in change using quarters, dimes, nickels and pennies using the fewest coins possible has at most two dimes, at most one nickel, at most four pennies, and cannot have two dimes and a nickel. The amount of change in dimes, nickels and pennies cannot exceed 24 cents.

Theorem 1. The cashier's algorithm always makes changes using the fewest coins possible when change is made from quarters, dimes, nickels and pennies.

(II.) Greedy Algo for Scheduling talks

ALGORITHM 8 Greedy Algorithm for Scheduling Talks.

procedure *schedule*($s_1 \le s_2 \le \cdots \le s_n$: start times of talks,
  $e_1 \le e_2 \le \cdots \le e_n$: ending times of talks)
sort talks by finish time and reorder so that $e_1 \le e_2 \le \cdots \le e_n$
$S := \emptyset$
for $j := 1$ **to** n
  if talk j is compatible with S **then**
    $S := S \cup \{\text{talk } j\}$
return S {S is the set of talks scheduled}

3.2 The Growth of Functions

The time required to solve a problem depends 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. One of the advantages of using big-O notation, which we introduce in this section, is that we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. This means that, using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-O notation, we can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

Big-O notation is used extensively to estimate the number of operations an algorithm uses as its input grows. With help of this notation, we can determine whether it is practical to use a particular algorithm to solve a problem as the size of the input increases. Furthermore, using big-O notation, we can compare two algorithms to determine which is more efficient as the size of the input grows. For instance, if we have two algorithms for solving a problem, one using $100n^2 + 17n + 4$ operations and the other using n^3 operations, big-O notation can help us see that the first algorithm uses far fewer operations when n is large, even though it uses more operations for small values of n, such as $n = 10$.

Big-O Notation (Landau symbol)

Definition 1. 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

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

C, k are called witnesses to the relationship f(x) = O(g(x)). We only need one pair of witnesses.

Remark: Intuitively, the definition that $f(x)$ is $O(g(x))$ say that $f(x)$ grows slower than some fixed multiple of $g(x)$ as x grows without bounds

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

Working with the definition of 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 $|f(x)| \le C|g(x)|$ for $x > k$.

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

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 $0 \le x^2 + 2x + 1 \le x^2 + 2x^2 + x^2 \le 4x^2$ whenever $x > 1$.

Consequently, we 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 $2x < x^2$ and $1 < x^2$.

Consequently, if $x > 2$, we have $0 \le x^2 + 2x + 1 \le x^2 + x^2 + x^2 \le 3x^2$. It follows that $C = 3$ and $k = 2$ are also witnesses to the relation $f(x)$ is $O(x^2)$.

Observe that in the relationship "$f(x)$ is $O(x^2)$", x^2 can be replaced by any function that has larger values than x^2 $\forall x \ge k \in \mathbb{R}^+$. 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 $O(x^2 + 2x + 1)$.

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

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 $C = 7$ and $k = 1$ are also witnesses to the relationship $7x^2$ is $O(x^3)$.

Example 3 | Show that n^2 is not $O(n)$.

To show that n^2 is not $O(n)$, we must show that no pair of witnesses C and k exist such that $n^2 \le Cn$ whenever $n > k$.

We will use a **proof by contradiction** to show this.

Suppose that there are constants C and k for which $n^2 \le Cn$ whenever $n > k$.

Observe that when $n > 0$ we can divide both sides of the inequality $n^2 \le Cn$ by n to obtain the equivalent inequality $n \le C$.

However, no matter what C and k are, the inequality $n \le C$ 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 $n \le C$ even though $n > k$.

This contradiction shows that n^2 is not $O(n)$. ∎

Example 4 | Example 2 shows that $7x^2$ is $O(x^3)$. Is it also true that x^3 is $O(7x^2)$?

To determine whether x^3 is $O(7x^2)$, we need to determine whether witnesses C and k exist, so that $x^3 \le C \cdot 7x^2$ whenever $x > k$.

We will show that no such witnesses exist using a proof by contraction.

If C and k are witnesses, the inequality $x^3 \le C \cdot 7x^2$ holds for all $x > k$.

Observe that the inequality $x^3 \le C \cdot 7x^2$ is equivalent to the inequality $x \le 7C$, which follows by dividing both sides by the positive quantity x^2.

However, no matter what C is, it is not the case that $x \le 7C$ 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 $O(7x^2)$. ∎

(I.) $f(x) = x^2 + 2x + 1 \le 4 \cdot x^2 \le 4 \cdot g(x)$, $\begin{cases} C = 4 \\ k = 1 \end{cases}$

(II.) $g(x) = x^2 \le x^2 + 2x + 1 \le 1 \cdot f(x)$, $\begin{cases} C = 1 \\ k = 1 \end{cases}$

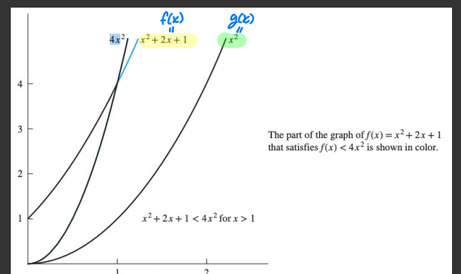

FIGURE 1 **The function $x^2 + 2x + 1$ is $O(x^2)$.**

Theorem 1 | Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, where $a_0, a_1, \dots, a_n \in \mathbb{R}$. Then $f(x)$ is $O(x^n)$.

Proof using the triangle inequality.

If $x > 1$ we have,

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

$\le |a_n| x^n + |a_{n-1}| x^{n-1} + \dots + |a_1| x + a_0$, $|x+y| \le |x| + |y|$

$\le x^n \left(|a_n| + \frac{|a_{n-1}|}{x} + \dots + \frac{|a_1|}{x^{n-1}} + \frac{|a_0|}{x^n} \right)$

$\le x^n \left(|a_n| + |a_{n-1}| + \dots + |a_1| + |a_0| \right)$

This shows that $|f(x)| \le C \cdot x^n$ where $C = |a_n| + |a_{n-1}| + \dots + |a_0|$ whenever $x > 1$.

Hence, the witnesses $\begin{cases} C = |a_n| + |a_{n-1}| + \dots + a_0 \\ k = 1 \end{cases}$ show that $f(x)$ is $O(x^n)$.

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

and if $|g(x)| < |h(x)|$ $\forall x > k$

then $|f(x)| \le C|h(x)|$ 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 often chosen to have the smallest growth rate of the functions belonging to a set of reference functions, such as functions of the form x^n, where n is a positive real number.

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

Example 5 | How can big-O notation be used to estimate the sum of the first n positive integers?

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

$\sum_{k=1}^{n} k = 1 + 2 + \dots + (n-1) + n$

$\le n + n + \dots + n$

$\le n \cdot n$

$\le n^2$ $\begin{cases} C = 1 \\ k = 1 \end{cases}$

From this inequality it follows that $1 + 2 + 3 + \dots + n$ is $O(n^2)$ taking $C = 1$ and $k = 1$ as witnesses.

Example 6 | Give big-O estimates for the factorial function and the logarithm of the factorial function, where the factorial function $f(n) = n!$ is defined by $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ whenever n is a positive integer, and $0! = 1$.

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

Hence, $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$

$\le n \cdot n \cdot n \cdot \dots \cdot n$

$\le n^n$

This inequality shows that $n!$ is $O(n^n)$ taking $C = 1$ and $k = 1$ as witnesses.

Taking logarithms of both sides of the inequality established for $n!$ we obtain $\log n! \le \log n^n \le n \cdot \log n$

This implies that $\log n!$ is $O(n \cdot \log n)$ again taking $C = 1$ and $k = 1$ as witnesses.

Example 7 | $n < 2^n$ whenever n is a positive integer. Show that this inequality implies that n is $O(2^n)$ and use this inequality to show that $\log n$ is $O(n)$.

Using the inequality $n < 2^n$, we quickly can conclude that n is $O(2^n)$ by taking $k = C = 1$ as witnesses.

Note that because the logarithm function is increasing, taking logarithm (base 2) of both sides of this inequality shows that $\log n < n$. It follows that $\log n$ is $O(n)$, $C = k = 1$ as witnesses. ∎

If we have logarithms to a base b, where b is different from 2, we still have $\log_b n$ is $O(n)$ because $\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$ whenever n is a positive integer. We take $C = \frac{1}{\log b}$ and $k = 1$ as witnesses.

Function often include the following: 1, $\log n$, n, $n \cdot \log n$, n^2, 2^n, $n!$

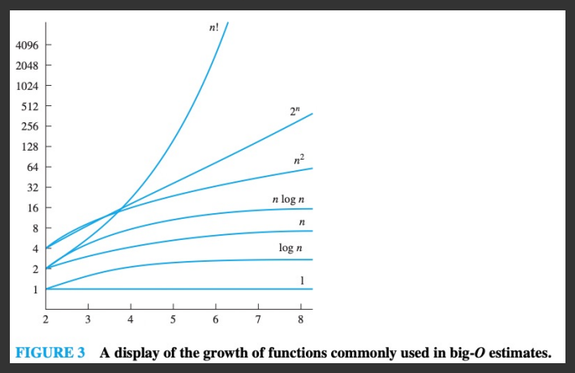

FIGURE 3 **A display of the growth of functions commonly used in big-O estimates.**

Useful big-O estimates involving logarithms, powers, and exponential functions

if $d > c > 1$, $b > 1$: $(\log_b n)^c$ is $O(n^d)$ but n^d is not $O((\log_b n)^c)$

n^d is $O(b^n)$, but b^n is not $O(n^d)$

$c > b > 1$: b^n is $O(c^n)$, but c^n is not $O(b^n)$

$c > 1$: c^n is $O(n!)$, but $n!$ is not $O(c^n)$

Example 8 | Arrange the functions $f_1(n) = 8^{f(n)}$; $f_4(n) = n!$ in a list so that each function is big-O of the next function.

$f_2(n) = (\log n)^2$; $f_5(n) = (1.1)^n$

$f_3(n) = 2 \cdot n \log n$; $f_6(n) = n^2$

$8^{f(n)}$, $2n \log n$, $(\log n)^2$, n^2, $(1.1)^n$, $n!$

The Growth of Combination of Functions

Theorem 2. Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$.
Then $(f_1+f_2)(x)$ is $O(g(x))$, where $g(x) = \max(|g_1(x)|, |g_2(x)|)$ $\forall x$

Corollary 1. Suppose that $f_1(x)$ and $f_2(x)$ are both $O(g(x))$. Then $(f_1+f_2)(x)$ is $O(g(x))$

Theorem 3. Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

Example 9 | Give a big-O estimate for $f(n) = 3n\log(n!) + (n^2+3)\log n$, where n is a positive integer.

First, the product $3n \cdot \log n!$ will be estimated. We know that $\log(n!)$ is $O(n \log n)$.
Using this estimate and the fact that $3n$ is $O(n)$, Th. 3 gives the estimate that
$3n \cdot \log n!$ is $O(n^2 \log n)$.
Next, the product $(n^2+3)\log n$ will be estimated. Because $(n^2+3) < n^2$ when $n > 2$,
it follows that (n^2+3) is $O(n^2)$. Thus, from Th. 3 it follows that $(n^2+3)\log n$ is $O(n^2 \log n)$
Using Th. 2 to combine the two big-O estimates for the products shows that
$f(n) = 3n\log n! + (n^2+3)\log n$ is $O(n^2 \log n)$

Example 10 | Give a big-O estimate for $f(x) = (x+1)\log(x^2+1) + 3x^2$

First, a big-O estimate for $(x+1)\log(x^2+1)$ will be found. Note that $(x+1)$ is $O(x)$.
Furthermore, $x^2+1 \le 2x^2$ when $x > 1$. Hence $\log(x^2+1) \le \log(2x^2)$
$\le \log(2) + \log(x^2)$
$\le \log(2) + 2\log(x)$
$\le 3\log(x)$ if $x > 2$.
This shows that $\log(x^2+1)$ is $O(\log x)$
From Th. 3 it follows that $(x+1)\log(x^2+1)$ is $O(x \log x)$.
Because $3x^2$ is $O(x^2)$, Th. 2 tells us that $f(x)$ is $O(\max(x\log x, x^2))$
Because $x \log x \le x^2$ for $x > 1$, it follows that $f(x)$ is $O(x^2)$

Big-Omega and Big-Theta Notation

Big-O notation is used extensively to describe the growth of functions, but it has limitations.
Big-O 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.

Definition 2. Let f and g: $\mathbb{Z} \cup \mathbb{R} \to \mathbb{R}$
We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k with C positive such that $|f(x)| \ge C|g(x)|$ whenever $x > k$.

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

Definition 3. Let f and g: $\mathbb{Z} \cup \mathbb{R} \to \mathbb{R}$.
We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.
($f(x)$ and $g(x)$ are the same order.)
When $f(x)$ is $\Theta(g(x))$, it is also the case that $g(x)$ is $\Theta(f(x))$.
Also note that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.
Furthermore, note that $f(x)$ is $\Theta(g(x))$ if and only if there are positive real numbers
C_1 and C_2 and a positive real number k such that $C_1|g(x)| \le f(x) \le C_2|g(x)|$ whenever $x > k$.
The existence of the constants C_1, C_2 and k tells us that $f(x)$ is $\Omega(g(x))$ and that $f(x)$ is $O(g(x))$, respectively.
Usually, when big-Θ notation is used, the function $g(x)$ in $\Theta(g(x))$ is a relatively simple reference function, such as x^n, c^x, $\log x$, and so on,
while $f(x)$ can be relatively complicated.

Example 12 | Determine whether this sum is order n^2 without using the summation formula for this sum.

Let $f(n) = 1+2+3+\dots+n$. Because we already know that $f(n)$ is $O(n^2)$, 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 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 $1+2+\dots+n \ge \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \dots + n$
$\ge \lceil n/2 \rceil$ ... $\ge \lceil n/2 \rceil$
$\ge \underbrace{\lceil n/2 \rceil + \lceil n/2 \rceil + \dots + \lceil n/2 \rceil}_{(n - \lceil n/2 \rceil + 1) \text{ fois}}$
$\ge (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil$
$\ge (n/2)(n/2)$
$\ge \frac{1}{4}n^2$
This shows that $f(n)$ is $\Omega(n^2)$
We conclude that $f(n)$ is of order n^2, or in symbols, $f(n)$ is $\Theta(n^2)$.
Remark: Note that we can also show that $f(n) = \sum_{i=1}^{n} i$ is $\Theta(n^2)$ using the closed formula $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Example 13 | Show that $3x^2 + 8x\log x$ is $\Theta(x^2)$

Because $0 \le 8x\log x \le 8x^2$, it follows that $3x^2 + 8x\log x \le 11x^2$ for $x > 1$.
Consequently, $3x^2 + 8x\log x$ is $O(x^2)$. Clearly, x^2 is $O(3x^2 + 8x\log x)$. Consequently, $3x^2 + 8x\log x$ is $\Theta(x^2)$.

Theorem 4. Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_1 x + a_0$, where $a_0, a_1, \dots, a_n \in \mathbb{R}$ with $a_n \neq 0$
Then $f(x)$ is of order x^n ($f(x) \in \Theta(x^n)$)

Brute-Force Algorithms

In general, brute-force algorithms are naive approaches for solving problems that do not take advantage of any special structures of the problem or clever ideas.
Although brute-force algorithms are often inefficient, they are often quite useful. A brute-force algorithm may be able to solve practical instances of problems, particularly when the input is not too large, even if it is impractical to use this algorithm for larger inputs. Furthermore, when designing new algorithms to solve a problem, the goal is often to find a new algorithm that is more efficient than a brute-force algorithm.

3.3 Complexity of Algorithms.

Considerations of space complexity are tied in with the particular data structures used to implement the algorithm.
Because data structures are not dealt with in detail in this book, space complexity will not be considered.
We will restrict our attention to time complexity.

Time Complexity

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 operation.
The fastest computers in existence can perform basic bit operations (adding, multiplying, comparing, exchanging two bits) in 10^{-11} second (10 picoseconds), but personnal computers may require 10^{-8} second (10 nanoseconds), which is 1000 times as long, to do the same operations.

Matrix-Chain Multiplication

How should the matrix-chain $A_1 A_2 \cdots A_n$ be computed using the fewest multiplications of integers?
Because matrix multiplication is associative, the order of the multiplication used does not change the product.

Example 9 | In which order should the matrice A_1, A_2 and A_3, where A_1 is 30×20, A_2 is 20×40, and A_3 is 40×10, all with integers entries be multiplied to use the least number of multiplications of integers.

There are two possible ways to compute $A_1A_2A_3$. These are $\underbrace{A_1(A_2A_3)}_{(I_e)}$ and $\underbrace{(A_1A_2)A_3}_{(II_e)}$.
(I_e) If A_2 and A_3 are first multiplied,
a total of $20 \cdot 40 \cdot 10 = 8000$ multiplications of integers are used to obtain the 20×10 matrix (A_2A_3).
Then, to multiply A_1 and A_2A_3 requires $30 \cdot 20 \cdot 10 = 6000$ multiplications.
Hence, a total of $8000 + 6000 = 14\,000$ multiplication are used.
(II_e) On the other hand, $A_1A_2 \to 30 \cdot 20 \cdot 40 = 24\,000$
$(A_1A_2)A_3 \to 30 \cdot 40 \cdot 10 = 12000$
36 000 multiplications are used.
Clearly, the first method is more efficient.