CHAPTER

3 Algorithms

3.1 Algorithms

PROPERTIES OF ALGORITHMS

- *Input.*
- *Definiteness.*
- *Output.*
- *Correctness.*
- *Finiteness.*
- *Effectiveness.*
- *Generality.*

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

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\}

13 15 16 18 3 5 6	20 22	1 2 3

Searching Problem:

Locate an element *x* in a list of distinct elements *a1, a2, . . . , an*, or determine that it is not in the list. Find the location of the term in the list that equals *x* (that is, *i* is the solution if *x* = *ai*) and is 0 if *x* is not in the list.

The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location* {*location* is the subscript of the term that equals x, or is 0 if x is not found}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	5	6	7	8	10	12	13	15	16	18	19	20	22

X=19

X=4

The Binary Search Algorithm.

procedure *binary search* (x: integer, $a_1, \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint }
$j := n$ {j is right endpoint of search interval}
while $i < j$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x = a_i$ **then return** i
  else if $x > a_m$ **then** $i := m + 1$
   else $j := m - 1$
return 0

X=19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	5	6	7	8	10	12	13	15	16	18	19	20	22

The Binary Search Algorithm.

procedure *binary search* (x: integer, $a_1, , \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint }
$j := n$ {j is right endpoint of search interval}
while $i < j$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x = a_i$ **then return** i
  else if $x > a_m$ **then** $i := m + 1$
    else $j := m - 1$
return 0

X=4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	5	6	7	8	10	12	13	15	16	18	19	20	22

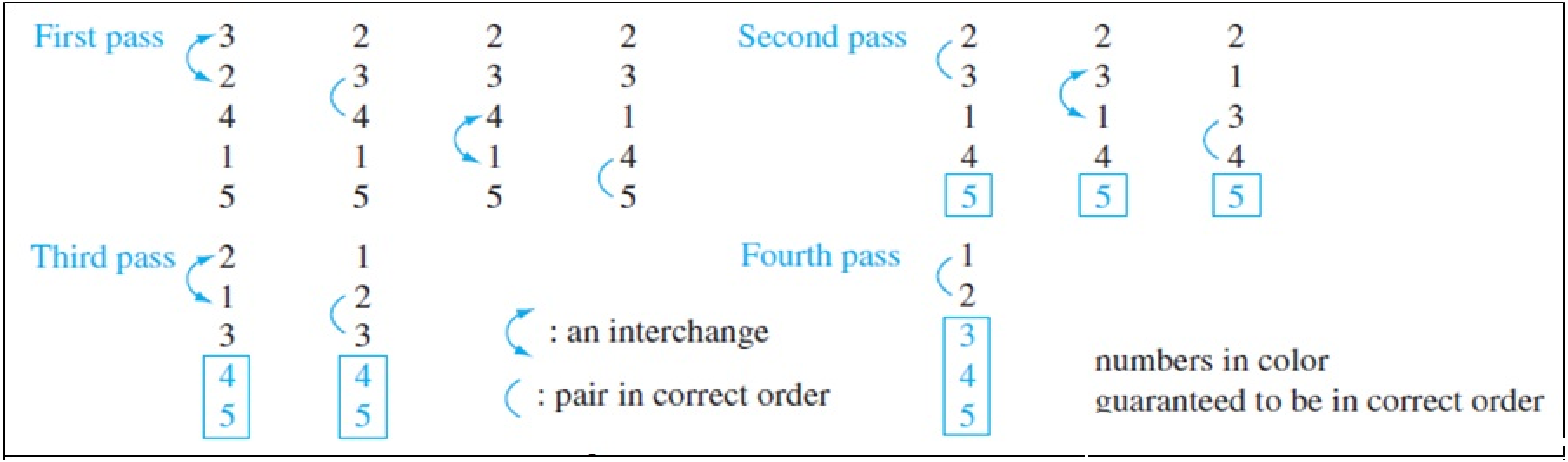
Sorting

Sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9.

There are some sorting algorithms

- Bubble sort
- Insertion sort
- Selection sort
- Heap sort
- quick sort
- Shaker sort

THE BUBBLE SORT



THE INSERTION SORT

67, 33, 21, 84, 49, 50, 75.

i=1	**33**	67	21	84	49	50	75
i=2	**21**	33	67	84	49	50	75
i=3	21	33	67	**84**	49	50	75
i=4	21	33	**49**	67	84	50	75
i=5	21	33	49	**50**	67	84	75
i=6	21	33	49	50	67	**75**	84

Greedy Algorithms

Selecting the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution.

- **Change-making**
- **Dijkstra**
- **Prim**
- **Huffman**

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

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.

We can closely approximate the time required to solve a problem of size *n* by multiplying the previous time required by a constant.

This factor will not depend on *n*.

Big-O Notation

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 $x > k$.

EXAMPLE

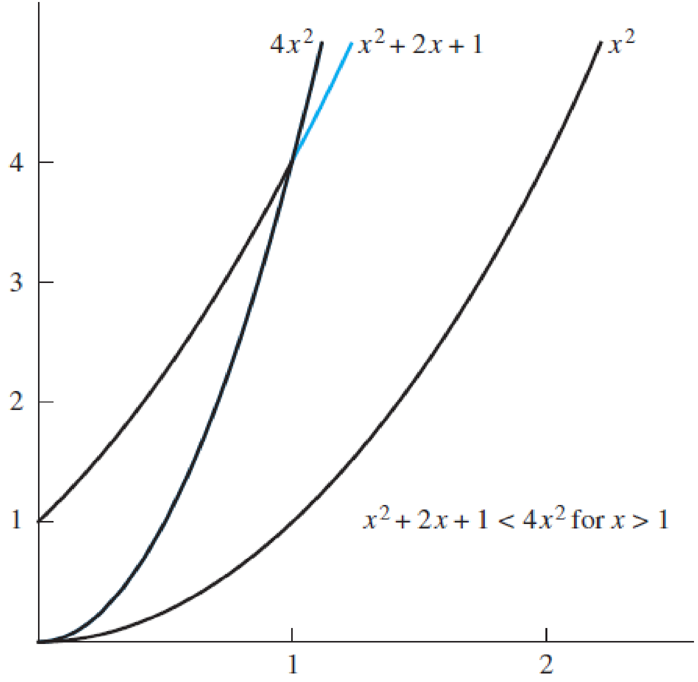
Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

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

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

$f(x) = x^2 + 2x + 1$ is $O(x^2)$.

3.2 The Growth of Functions



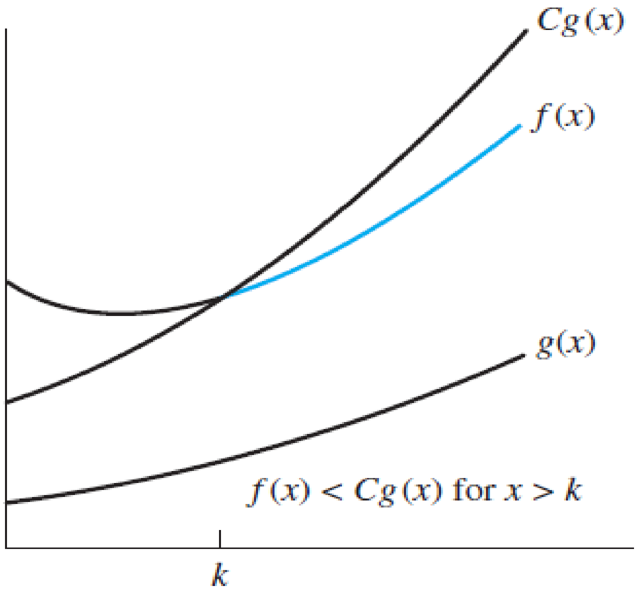
$f(x)$ is $O(x^2 + x + 7)$ $\quad f(x)$ is $O(x^3)$

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

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

$|h(x)| > |g(x)|$

$|f(x)| \le C|h(x)|$



3.2 The Growth of Functions

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n^2.$$

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

is $O(x^n)$.

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \leq n \cdot n \cdot n \cdot \cdots \cdot n = n^n.$$

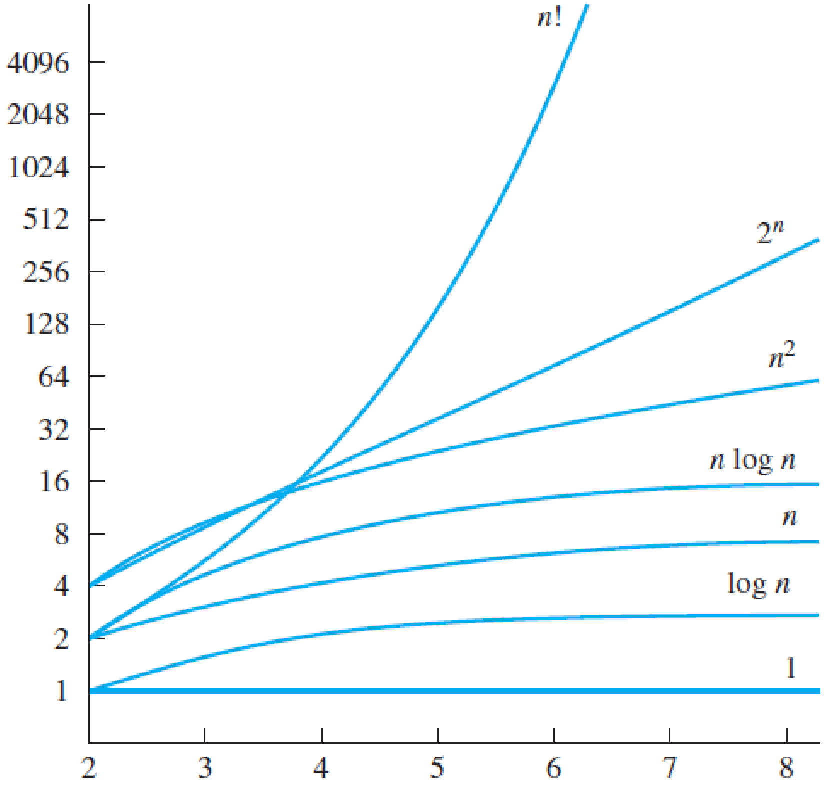
The Growth of Combinations of Functions

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 $O(\max(|g_1(x)|, |g_2(x)|))$.

Then $(f_1 f_2)(x)$ is $O(g_1(x) g_2(x))$.

Big-Omega and Big-Theta Notation



A Display of the Growth of Functions Commonly Used in Big-O Estimates.

Time Complexity

3.3 Complexity of Algorithms

How can the efficiency of an algorithm be analyzed?

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 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}

i	#comp
1	1
2	2
…	
n	n
n+1	1

Big-*O* Notation

WORST-CASE COMPLEXITY

Worst-case analysis tells us how many operations an algorithm requires to guarantee that it will produce a solution.

AVERAGE-CASE COMPLEXITY

i	#comp
1	1
2	2
…	
n	n
n+1	1

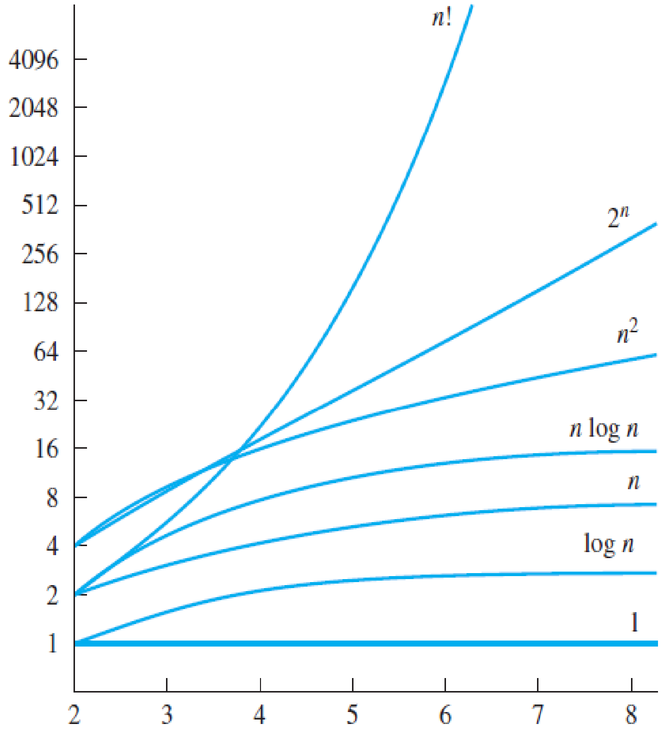
The Linear Search Algorithm.

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}

Commonly Used Terminology for the Complexity of Algorithms.

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

3.2 The Growth of Functions



A Display of the Growth of Functions Commonly Used in Big-O Estimates.