Chapter 3
The Efficiency of Algorithms

INVITATION TO
Computer Science

6TH EDITION

Objectives

After studying this chapter, students will be able to:

- Describe algorithm attributes and why they are important
- Explain the purpose of efficiency analysis and apply it to new algorithms to determine the order of magnitude of their time efficiencies
- Describe, illustrate, and use the algorithms from the chapter, including: sequential and binary search, selection sort, data cleanup algorithms, pattern matching

Objectives (continued)

After studying this chapter, students will be able to:

- Explain which orders of magnitude grow faster or slower than others
- Describe what an intractable problem is, giving one or more examples, and the purpose of approximation algorithms that partially solve them

Introduction

- Many solutions to any given problem
- How can we judge and compare algorithms?
- Metaphor: Purchasing a car
  - ease of handling
  - style
  - fuel efficiency
- Evaluating an algorithm
  - ease of understanding
  - elegance
  - time/space efficiency

Attributes of Algorithms

- Attributes of interest: correctness, ease of understanding, elegance, and efficiency
- Correctness:
  - Is the problem specified correctly?
  - Does the algorithm produce the correct result?
- Example: pattern matching
  - Problem spec: “Given pattern p and text t, determine the location, if any, of pattern p occurring in text t”
  - Algorithm correct: does it always work?

Attributes of Algorithms (continued)

- Ease of understanding, useful for:
  - checking correctness
  - program maintenance
- Elegance: using a clever or non-obvious approach
  - Example: Gauss' summing of 1 + 2 + … + 100
- Attributes may conflict: Elegance often conflicts with ease of understanding
- Attributes may reinforce each other: Ease of understanding supports correctness

Attributes of Algorithms (continued)

- **Efficiency:** an algorithm's use of time and space resources
  - Timing an algorithm is not always useful
  - Confounding factors: machine speed, size of input
- **Benchmarking:** timing an algorithm on standard data sets
  - Testing hardware and operating system, etc.
  - Testing real-world performance limits

Measuring Efficiency
Sequential Search

- **Analysis of algorithms:** the study of the efficiency of algorithms
- **Searching:** the task of finding a specific value in a list of values, or deciding it is not there
- **Sequential search algorithm** (from Ch. 2):

"Given a target value and a random list of values, find the location of the target in the list, if it occurs, by checking each value in the list in turn"

FIGURE 3.1

1. Get values for *NAME*, n, $N_1, \ldots, N_n$ and $T_1, \ldots, T_n$
2. Set the value of i to 1 and set the value of *Found* to NO
3. While (*Found* = NO) and ($i \leq n$) do Steps 4 through 7
4.   If *NAME* is equal to the ith name on the list, N_i, then
5.   Print the telephone number of that person, T_i
6.   Set the value of *Found* to YES

     Else (*NAME* is not equal to N_i)

7.   Add 1 to the value of i
8. If (*Found* = NO) then
9.   Print the message 'Sorry, this name is not in the directory'
10. Stop

Sequential search algorithm

Measuring Efficiency
Sequential Search (continued)

- Central unit of work, operations most important for the task, and occurring frequently
- In sequential search, comparison of target NAME to each name in the list
- Given a big input list:
  - Best case is smallest amount of work algorithm does
  - Worst case is greatest amount of work algorithm does
  - Average case depends on likelihood of different scenarios occurring

Measuring Efficiency
Sequential Search (continued)

- Best case: target found with the first comparison
- Worst case: target never found or last value
- Average case: if each value is equally likely to be searched, work done varies from 1 to n, averages to n/2

FIGURE 3.2

Best Case	Worst Case	Average Case
1	n	$n/2$

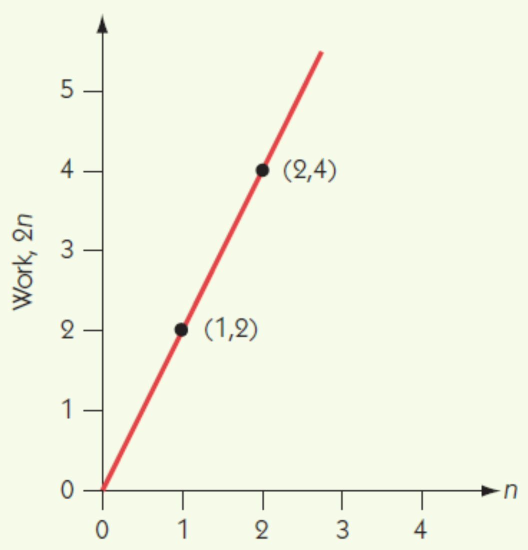
Number of comparisons to find NAME in a list of n names using sequential search

Measuring Efficiency

Order of Magnitude—Order n

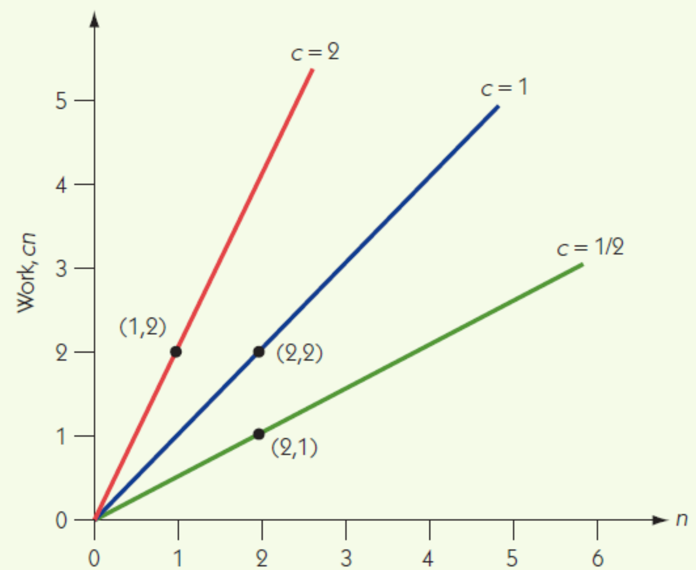
- **Order of magnitude n**, Θ(n): the set of functions that grow in a linear fashion

FIGURE 3.3



Work = 2*n*

FIGURE 3.4

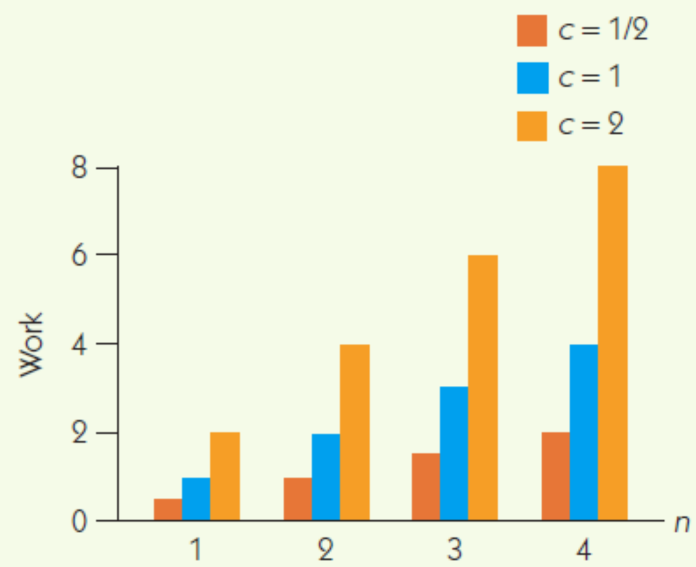


Work = *cn* for various values of *c*

Measuring Efficiency
Order of Magnitude—Order n (continued)

- Change in growth as n increases is constant size

FIGURE 3.5



Growth of work = *cn* for various values of *c*

Measuring Efficiency
Selection Sort

- **Sorting:** The task of putting a list of values into numeric or alphabetical order
- Key idea:
  - Pass repeatedly over the unsorted portion of the list
  - Each pass *select* the largest remaining value
  - Move that value to the end of the unsorted values

FIGURE 3.6

1. Get values for *n* and the *n* list items
2. Set the marker for the unsorted section at the end of the list
3. While the unsorted section of the list is not empty, do Steps 4 through 6
4. Select the largest number in the unsorted section of the list
5. Exchange this number with the last number in the unsorted section of the list
6. Move the marker for the unsorted section left one position
7. Stop

Selection sort algorithm

Measuring Efficiency
Selection Sort (continued)

Example: Selection Sort on [5, 1, 3, 9, 4]

- Pass 1:
  - Select 9 as the largest in the whole list
  - Swap with 4 to place in last slot
  - [5, 1, 3, 4, 9]
- Pass 2:
  - Select 5 as the largest in the first four values
  - Swap with 4 to place in last remaining slot
  - [4, 1, 3, 5, 9]

Measuring Efficiency
Selection Sort (continued)

Example: Selection Sort on [5, 1, 3, 9, 4]

- Pass 3:
  - Select 4 as the largest in the first three
  - Swap with 3 to place in last slot
  - [3, 1, 4, 5, 9]
- Pass 4:
  - Select 3 as the largest in the first two values
  - Swap with 1 to place in last remaining slot
  - [1, 3, 4, 5, 9]

Measuring Efficiency
Selection Sort (continued)

- Central unit of work: hidden in “find largest” step
- Work done to find largest changes as unsorted portion shrinks
- $(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2$

FIGURE 3.7

Length *n* of List to Sort	n^2	Number of Comparisons Required
10	100	45
100	10,000	4,950
1,000	1,000,000	499,500

Comparisons required by selection sort

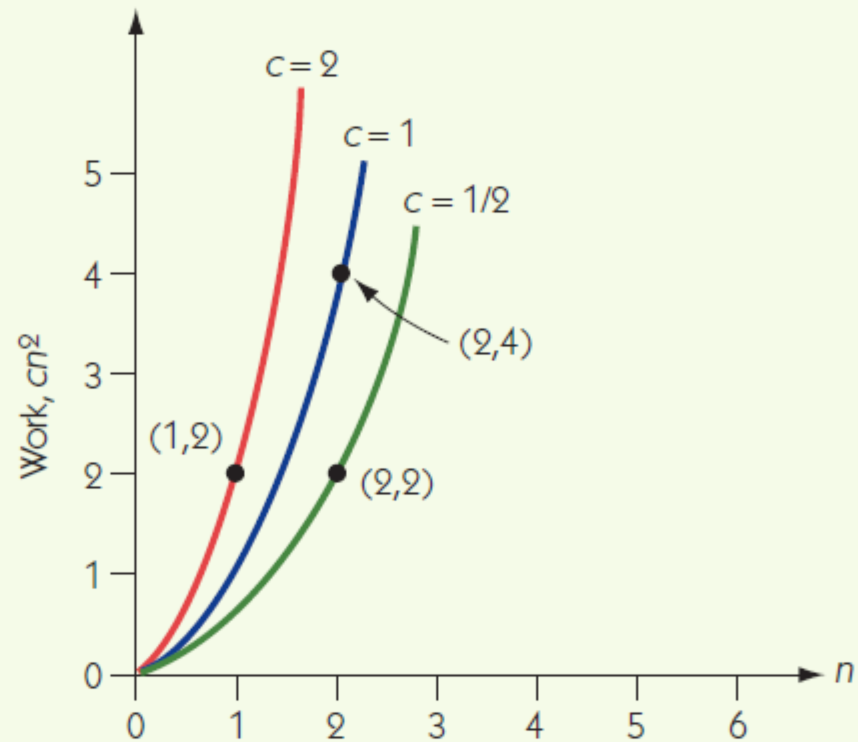
Measuring Efficiency

Order of Magnitude—Order n^2

Order n^2, $\Theta(n^2)$:

the set of functions whose growth is on the order of n^2

FIGURE 3.10

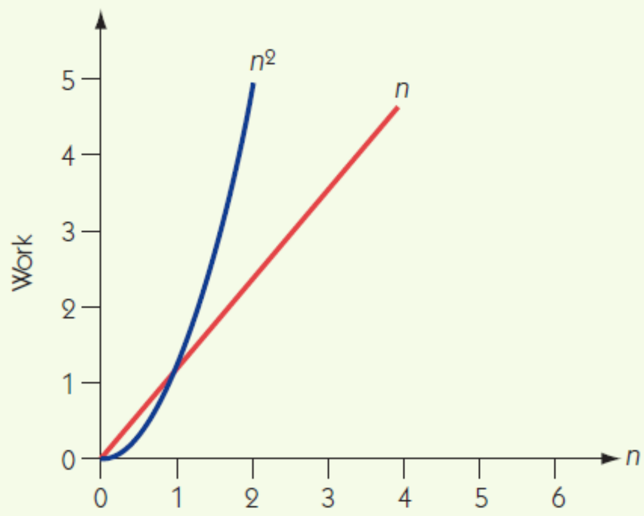


Work = cn^2 for various values of c

Measuring Efficiency
Order of Magnitude—Order n^2 (continued)

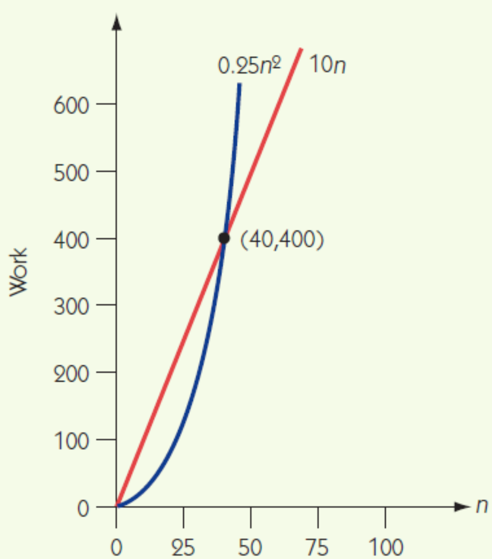
Eventually, every function with order n^2 has greater values than any function with order n

FIGURE 3.11



A comparison of n and n^2

FIGURE 3.12



For large enough n, $0.25n^2$ has larger values than $10n$

FIGURE 3.13

	Number of Work Units Required	
	Algorithm A	Algorithm B
n	$0.0001n^2$	$100n$
1,000	100	100,000
10,000	10,000	1,000,000
100,000	1,000,000	10,000,000
1,000,000	100,000,000	100,000,000
10,000,000	10,000,000,000	1,000,000,000

A comparison of two extreme $\Theta(n^2)$ and $\Theta(n)$ algorithms

Analysis of Algorithms
Data Cleanup Algorithms

"Given a collection of age data, where erroneous zeros occur, find and remove all the zeros from the data, reporting the number of legitimate age values that remain"

- Illustrates multiple solutions to a single problem
- Use of analysis to compare algorithms

Analysis of Algorithms

Data Cleanup Algorithms (continued)

- Shuffle-left algorithm:
  - Search for zeros from left to right
  - When a zero is found, shift all values to its right one cell to the left IS THIS RIGHT?
- Example: [55, 0, 32, 19, 0, 27]
  - Finds 0 at position 2: [55, 32, 19, 0, 27, 27]
  - Finds 0 at position 4: [55, 32, 19, 27, 27, 27]

FIGURE 3.14

1. Get values for *n* and the *n* data items
2. Set the value of *legit* to *n*
3. Set the value of *left* to *1*
4. Set the value of *right* to 2
5. While *left* is less than or equal to *legit* do Steps 6 through 14
6.   If the item at position *left* is not 0 then do Steps 7 and 8
7.   Increase *left* by 1
8.   Increase *right* by 1
9.   Else (the item at position *left* is 0) do Steps 10 through 14
10.   Reduce *legit* by 1
11.   While *right* is less than or equal to *n* do Steps 12 and 13
12.   Copy the item at position *right* into position (*right* – 1)
13.   Increase *right* by 1
14.   Set the value of *right* to (*left* + 1)
15. Stop

The shuffle-left algorithm for data cleanup

Analysis of Algorithms

Data Cleanup Algorithms (continued)

- Analysis of shuffle-left for time efficiency
  - Count comparisons looking for zero AND movements of values
  - Best case: no zeros occur, check each value and nothing more: $\Theta(n)$
  - Worst case: every value is a zero, move n-1 values, then n-2 values, etc.: $\Theta(n^2)$
- Analysis of shuffle-left for space efficiency
  - Uses no significant space beyond input

Analysis of Algorithms
Data Cleanup Algorithms (continued)

- Copy-over algorithm:
  - Create a second, initially empty, list
  - Look at each value in the original
  - If it is non-zero, copy it to the second list
- Example: [55, 0, 32, 19, 0, 27]

1. answer = [55]
2. answer = [55]
3. answer = [55, 32]
4. answer = [55, 32, 19]
5. answer = [55, 32, 19]
6. answer = [55, 32, 19, 27]

FIGURE 3.15

1. Get values for *n* and the *n* data items
2. Set the value of *left* to 1
3. Set the value of *newposition* to 1
4. While *left* is less than or equal to *n* do Steps 5 through 8
5.   If the item at position *left* is not 0 then do Steps 6 and 7
6.   Copy the item at position *left* into position *newposition* in new list
7.   Increase *newposition* by 1
8.   Increase *left* by 1
9. Stop

The copy-over algorithm for data cleanup

Analysis of Algorithms

Data Cleanup Algorithms (continued)

- Time efficiency for copy-over
  - Best case: all zeros, checks each value but doesn't copy it: $\Theta(n)$
  - Worst case: no zeros, checks each value and copies it: $\Theta(n)$
- Space efficiency for copy-over
  - Best case: all zeros, uses no extra space
  - Worst case: no zeros, uses n extra spaces

Analysis of Algorithms
Data Cleanup Algorithms (continued)

- Converging-pointers algorithm:
  - Keep track of two pointers at the data
  - Left pointer moves left to right and stops when it sees a zero value
  - Right pointer stays put until a zero is found
  - Then its value is copied on top of the zero, and it moves one cell to the left
  - Stop when the left crosses the right

FIGURE 3.16

1. Get values for *n* and the *n* data items
2. Set the value of *legit* to *n*
3. Set the value of *left* to 1
4. Set the value of *right* to *n*
5. While *left* is less than *right* do Steps 6 through 10
6.   If the item at position *left* is not 0 then increase *left* by 1
7.   Else (the item at position *left* is 0) do Steps 8 through 10
8.   Reduce *legit* by 1
9.   Copy the item at position *right* into position *left*
10.   Reduce *right* by 1
11. If the item at position *left* is 0, then reduce *legit* by 1
12. Stop

The converging-pointers algorithm for data cleanup

Analysis of Algorithms

Data Cleanup Algorithms (continued)

Example: [55, 0, 32, 19, 0, 27]

[55, 0, 32, 19, 0, 27]

L R

[55, 0, 32, 19, 0, 27]

L R

[55, 27, 32, 19, 0, 27]

L R

[55, 27, 32, 19, 0, 27]

LR

Analysis of Algorithms

Data Cleanup Algorithms (continued)

- Time efficiency for converging-pointers
  - Best case: no zeros, left pointer just moves across to pass the right pointers, examines each value: $\Theta(n)$
  - Worst case: all zeros, examines each value and copies a value over it, right pointer moves left towards left pointer: $\Theta(n)$
- Space efficiency for converging-pointers
  - No significant extra space needed

FIGURE 3.17

	1. Shuffle-left		2. Copy-over		3. Converging-pointers	
	Time	Space	Time	Space	Time	Space
Best case	$\Theta(n)$	n	$\Theta(n)$	n	$\Theta(n)$	n
Worst case	$\Theta(n^2)$	n	$\Theta(n)$	$2n$	$\Theta(n)$	n
Average case	$\Theta(n^2)$	n	$\Theta(n)$	$n \leq x \leq 2n$	$\Theta(n)$	n

Analysis of three data cleanup algorithms

Analysis of Algorithms

Binary Search

Binary Search Algorithm:

"Given a target value and an *ordered* list of values, find the location of the target in the list, if it occurs, by starting in the middle and splitting the range in two with each comparison"

FIGURE 3.18

1. Get values for *NAME*, *n*, $N_1, \ldots, N_n$ and $T_1, \ldots, T_n$
2. Set the value of *beginning* to 1 and set the value of *Found* to NO
3. Set the value of *end* to *n*
4. While *Found* = NO and *beginning* is less than or equal to *end* do Steps 5 through 10
5. Set the value of *m* to the middle value between *beginning* and *end*
6. If *NAME* is equal to N_m, the name found at the midpoint between *beginning* and *end*, then do Steps 7 and 8
7. Print the telephone number of that person, T_m
8. Set the value of *Found* to YES
9. Else if *NAME* precedes N_m alphabetically, then set $end = m - 1$
10. Else (*NAME* follows N_m alphabetically) set $beginning = m + 1$
11. If (*Found* = NO) then print the message 'I am sorry but that name is not in the directory'
12. Stop

Binary search algorithm (list must be sorted)

Analysis of Algorithms

Binary Search (continued)

Example: target = 10, list = [1, 4, 5, 7, 10, 12, 14, 22]

mid = 7, eliminate lower half:

[1, 4, 5, 7, 10, 12, 14, 22]

mid = 12, eliminate upper half:

[1, 4, 5, 7, 10, 12, 14, 22]

mid = 10, value found!

Analysis of Algorithms
Binary Search (continued)

- Central unit of work: comparisons against target
- Best case efficiency:
  - Value happens to be the first middle value: 1 comparison
- Worst case efficiency:
  - Value does not appear, repeats as many times as we can divide the list before running out of values: $\Theta(\lg n)$

Analysis of Algorithms
Binary Search (continued)

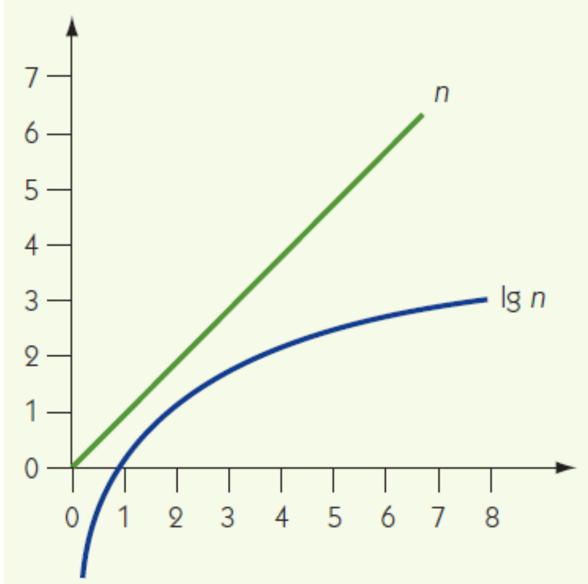
- **Order of magnitude lg n**, Θ(lg n), grows very slowly

FIGURE 3.20

n	lg *n*
8	3
16	4
32	5
64	6
128	7

Values for *n* and lg *n*

FIGURE 3.21



A comparison of *n* and lg *n*

Analysis of Algorithms
Pattern Matching

- Algorithm from chapter 2
- Best case: when first symbol of pattern does not appear in text
- Worst case: when all but last symbol of pattern make up the text

FIGURE 2.16

```
Get values for n and m, the size of the text and the pattern, respectively
Get values for both the text T1 T2 ... Tn and the pattern P1 P2 ... Pm
Set k, the starting location for the attempted match, to 1
While (k ≤ (n − m + 1)) do
    Set the value of i to 1
    Set the value of Mismatch to NO
    While both (i ≤ m) and (Mismatch = NO) do
        If Pi ≠ Tk+(i−1) then
            Set Mismatch to YES
        Else
            Increment i by 1 (to move to the next character)
    End of the loop
    If Mismatch = NO then
        Print the message 'There is a match at position'
        Print the value of k
    Increment k by 1
End of the loop
Stop, we are finished
```

Final draft of the pattern-matching algorithm

Analysis of Algorithms
Pattern Matching (continued)

- Best case example:
  - pattern = “xyz”   text = “aaaaaaaaaaaaaaa”
  - At each step, compare ‘x’ to ‘a’ and then move on
  - $\Theta(n)$ comparisons
- Worst case example:
  - pattern = “aab”   text = “aaaaaaaaaaaaaaa”
  - At each step, compare m symbols from pattern against text before moving on
  - $\Theta(mn)$ comparisons

FIGURE 3.22

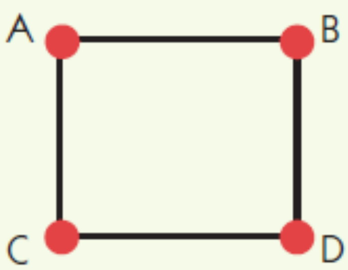
Problem	Unit of Work	Algorithm	Best Case	Worst Case	Average Case
Searching	Comparisons	Sequential search	1	$\Theta(n)$	$\Theta(n)$
		Binary search	1	$\Theta(\lg n)$	$\Theta(\lg n)$
Sorting	Comparisons and exchanges	Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Data cleanup	Examinations and copies	Shuffle-left	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
		Copy-over	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
		Converging-pointers	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Pattern matching	Character comparisons	Forward march	$\Theta(n)$	$\Theta(m \times n)$	

Order-of-magnitude time efficiency summary

When Things Get Out of Hand

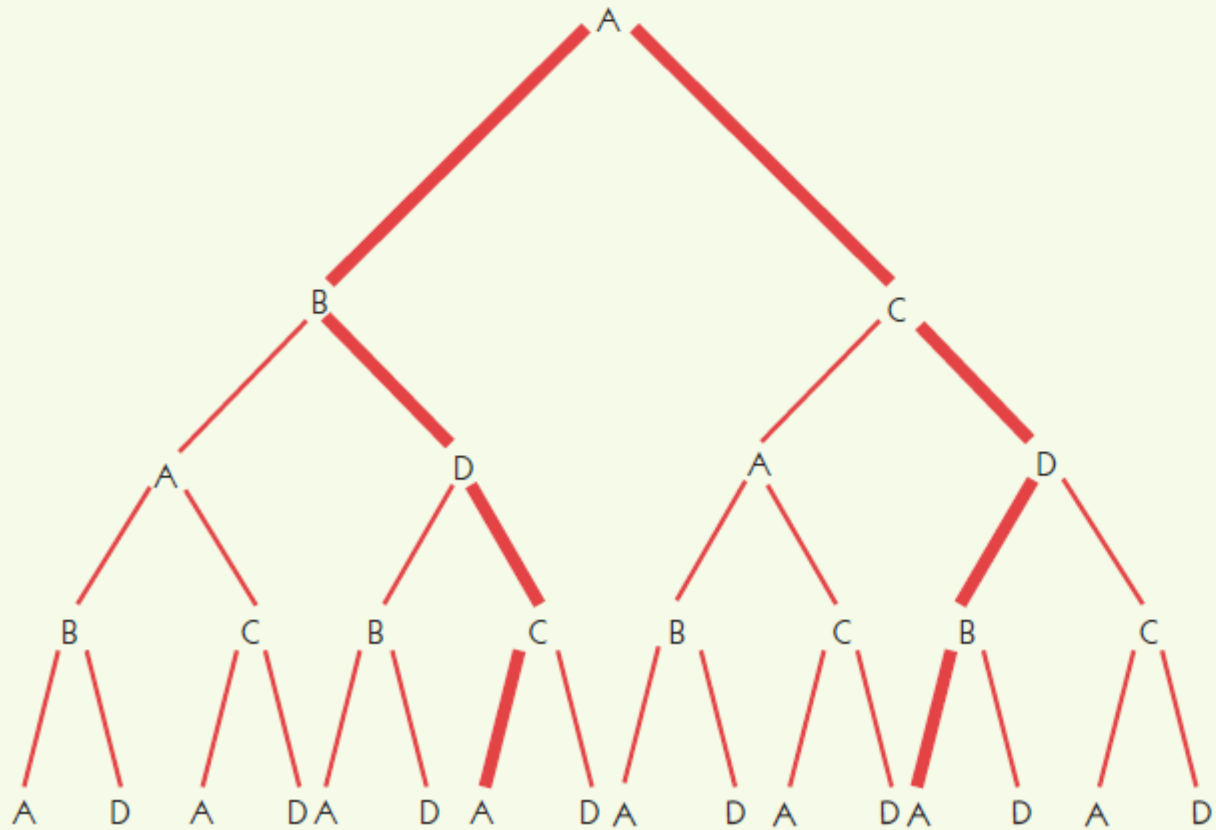
- **Polynomially bounded:** an algorithm that does work on the order of $\Theta(n^k)$
- Most common problems are polynomially bounded
- Hamiltonian circuit is NOT
  - Given a graph, find a path that passes through each vertex exactly once and returns to its starting point

FIGURE 3.23



Four connected cities

FIGURE 3.24



Hamiltonian circuits among all paths from A in Figure 3.23 with four links

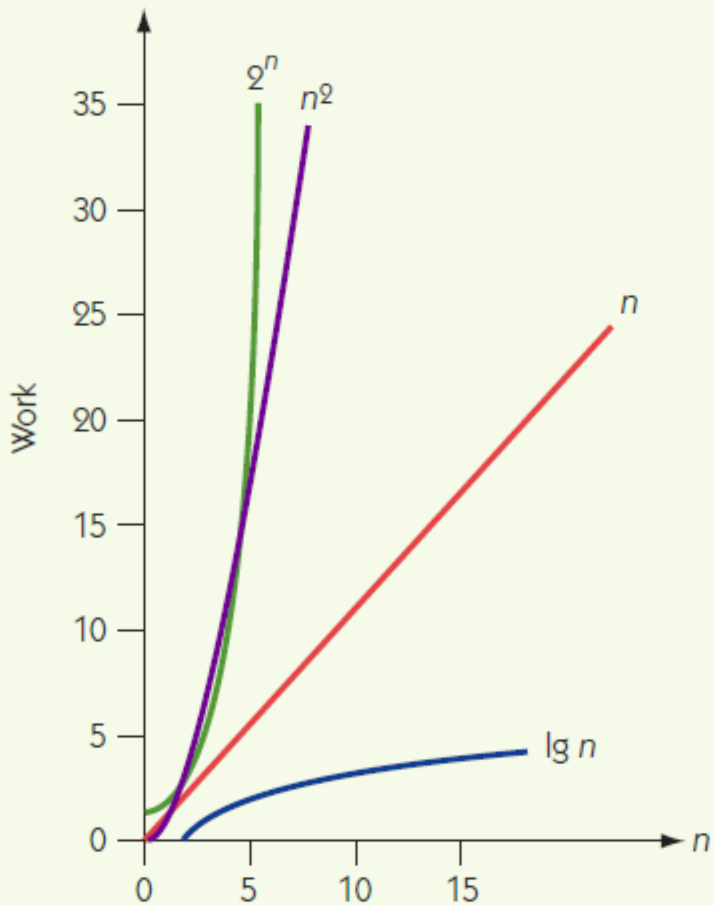
When Things Get Out of Hand (continued)

- Possible paths in the graph are paths through a tree of choices
- Most simple case has exactly two choices per vertex
- Number of paths to examine = number of leaves in the tree
- Height of the tree = n+1 (n is the number of vertices in the graph)
- Number of leaves = 2^n

When Things Get Out of Hand (continued)

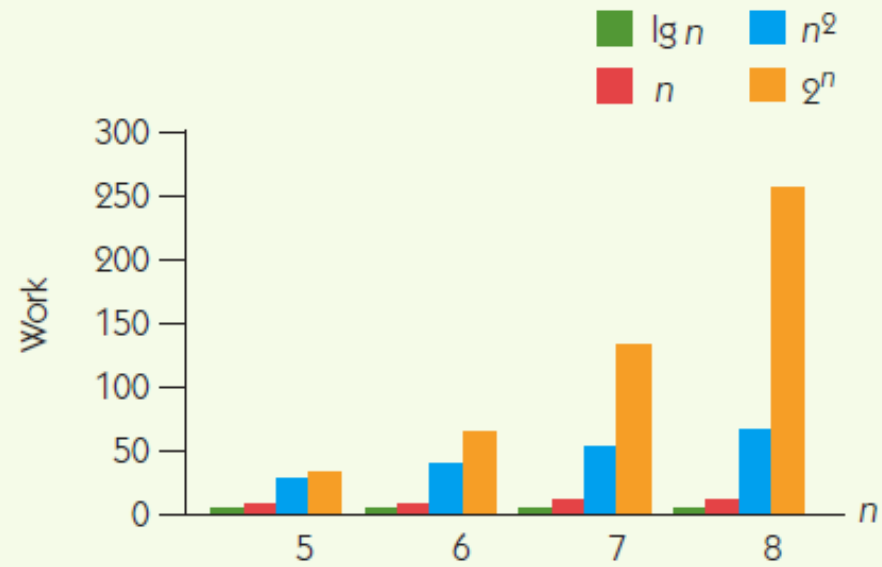
- **Exponential algorithm:** an algorithm whose order of growth is $\Theta(k^n)$
- **Intractable:** problems with no polynomially-bounded solutions
  - Hamiltonian circuit
  - Traveling Salesperson
  - Bin packing
  - Chess

FIGURE 3.25



Comparison of lg n, n, n^2, and 2^n

FIGURE 3.26



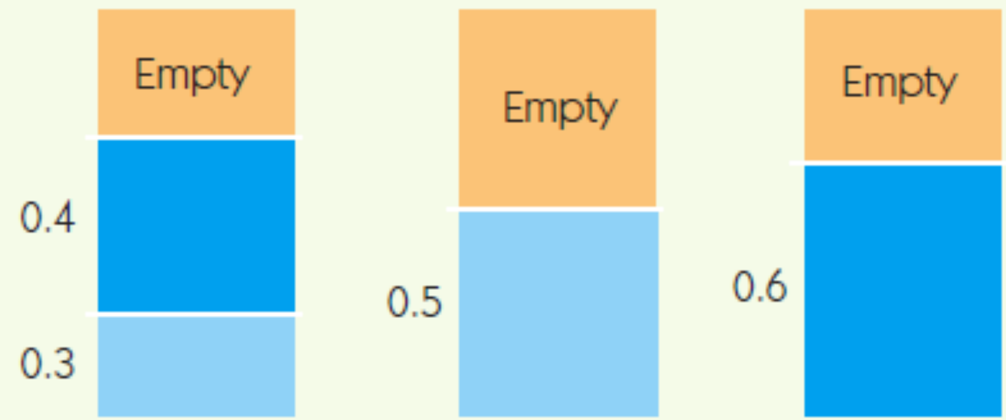
Comparisons of lg n, n, n^2, and 2^n for larger values of n

FIGURE 3.27

Order	n: 10	50	100	1,000
lg *n*	0.0003 sec	0.0006 sec	0.0007 sec	0.001 sec
n	0.001 sec	0.005 sec	0.01 sec	0.1 sec
n^2	0.01 sec	0.25 sec	1 sec	1.67 min
2^n	0.1024 sec	3,570 years	4×10^{16} centuries	*Too big to compute!!*

A comparison of four orders of magnitude

FIGURE 3.28



A first-fit solution to a bin-packing problem

When Things Get Out of Hand (continued)

- **Approximation algorithms:** algorithms that partially solve, or provide sub-optimal solutions to, intractable problems
- Example: bin packing
- For each box to be packed
  - check each current bin
    - if new box fits in the bin, place it there
  - if no bin can hold the new box, add a new bin

Summary

- We must evaluate the quality of algorithms, and compare competing algorithms to each other
- Attributes: correctness, efficiency, elegance, and ease of understanding
- Compare competing algorithms for time and space efficiency (time/space tradeoffs are common)
- Orders of magnitude capture work as a function of input size: $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(2^n)$
- Problems with only exponential algorithms are intractable