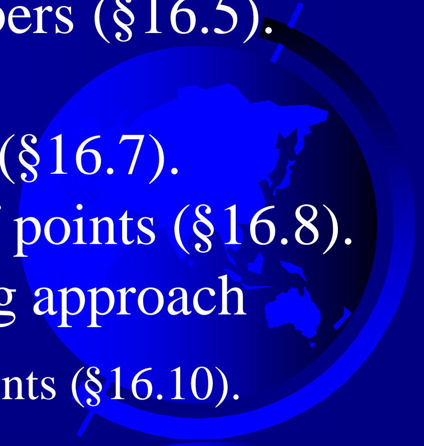
Chapter 16 Developing Efficient Algorithms

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Objectives

- To estimate algorithm efficiency using the Big *O* notation (§16.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§16.2).
- To determine the complexity of various types of algorithms (§16.3).
- To analyze the binary search algorithm (§16.4.1).
- To analyze the selection sort algorithm (§16.4.2).
- To analyze the insertion sort algorithm (§16.4.3).
- To analyze the Towers of Hanoi algorithm (§16.4.4).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, exponential) (§16.4.5).
- To design efficient algorithms for finding Fibonacci numbers (§16.5).
- To design efficient algorithms for finding gcd (§16.6).
- To design efficient algorithms for finding prime numbers (§16.7).
- To design efficient algorithms for finding a closest pair of points (§16.8).
- To solve the Eight Queens problem using the backtracking approach (§16.9).
- To design efficient algorithms for finding a convex hull for a set of points (§16.10).



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Executing Time

- Suppose two algorithms perform the same task such as search (linear search vs. binary search) and sorting (selection sort vs. insertion sort).
- Which one is better? One possible approach to answer this question is to implement these algorithms in a Programming language and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Growth Rate

- It is very difficult to compare algorithms by measuring their execution time.
- To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input.
- This approach approximates the effect of a change on the size of the input.
- In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

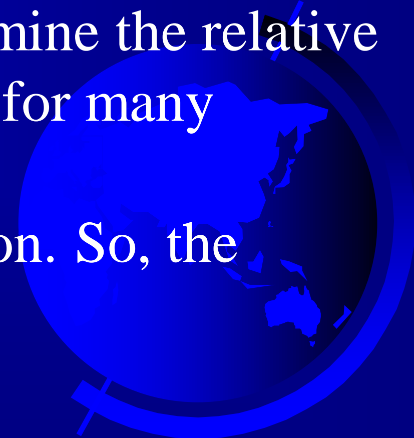
Big O Notation

- Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires *n* comparisons for an array of size *n*. If the key is in the array, it requires *n/2* comparisons on average.
- The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double.
- The algorithm grows at a linear rate. The growth rate has an order of magnitude of *n*.
- Computer scientists use the Big *O* notation to abbreviate for "order of magnitude."
- Using this notation, the complexity of the linear search algorithm is *O(n)*, pronounced as "*order of n*."

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Best, Worst, and Average Cases

- For the same input size, an algorithm's execution time may vary, depending on the input.
  - An input that results in the shortest execution time is called the *best-case* input and
  - an input that results in the longest execution time is called the *worst-case* input.
- Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.
- An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems.
- Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Ignoring Multiplicative Constants

- The linear search algorithm requires n comparisons in the worst-case and $n/2$ comparisons in the average-case.
- Using the Big O notation, both cases require $O(n)$ time.
- The multiplicative constant (1/2) can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for n/2 or 100n is the same as n, i.e., O(n) = O(n/2) = O(100n).

n \ $f(n)$	n	$n/2$	$100n$
100	100	50	10000
200	200	100	20000
$f(200) / f(100)$	2	2	2

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number.
- In general, it takes $n-1$ times of comparisons to find maximum number in a list of n elements.
- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency.
- As n grows larger, the n part in the expression $n-1$ dominates the complexity.
- The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression $n-1$) and highlight the important part (e.g., n in the expression $n-1$). So, the complexity of this algorithm is $O(n)$.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Useful Mathematic Summations

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

$$a^{0}+a^{1}+a^{2}+a^{3}+....+a^{(n-1)}+a^{n}=\frac{a^{n+1}-1}{a-1}$$

$$2^{0}+2^{1}+2^{2}+2^{3}+....+2^{(n-1)}+2^{n}=\frac{2^{n+1}-1}{2-1}$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Examples: Determining Big-O

- Repetition
- Sequence
- Selection
- Logarithm

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Repetition: Simple Loops

executed
n times

```
for i in range(n):
        k = k + 5
```

constant time

Time Complexity

$T(n) = (\text{a constant } c) * n = cn = \mathbf{O(n)}$

Ignore multiplicative constants (e.g., "c").

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Repetition: Nested Loops

executed
n times

```
for i in range(n):
    for j in range(n):
        k = k + i + j
```

inner loop
executed
n times

constant time

Time Complexity

$$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$$

Ignore multiplicative constants (e.g., "c").

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Repetition: Nested Loops

executed n times

```
for i in range(n):
    for j in range(i):
        k = k + i + j
```

inner loop executed i times

constant time

Time Complexity

$T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$

Ignore non-dominating terms

Ignore multiplicative constants

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Repetition: Nested Loops

executed
n times

```
for i in range(n):
    for i in range(20):
        k = k + i + j
```

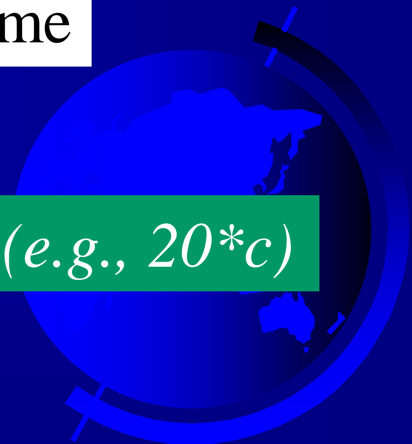
inner loop
executed
20 times

constant time

Time Complexity

$T(n) = 20 * c * n = O(n)$

*Ignore multiplicative constants (e.g., 20*c)*



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Sequence

executed *10* times

```
for i in range(9):
    k = k + 4
```

executed *n* times

```
for i in range(n):
    for i in range(20):
        k = k + i + j
```

inner loop executed *20* times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Selection

O(n)

```
if (list.contains(e)) {
  System.out.println(e);
}
else
  for (Object t: list) {
    System.out.println(t);
  }
```

Let n be list.size(). Executed n times.

Time Complexity

$$T(n) = \text{test time} + \text{worst-case (if, else)}$$
$$= O(n) + O(n)$$
$$= O(n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Constant Time

- The Big *O* notation estimates the execution time of an algorithm in relation to the input size.
- If the time is not related to the input size, the algorithm is said to take *constant time* with the notation *O(1)*.
- For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

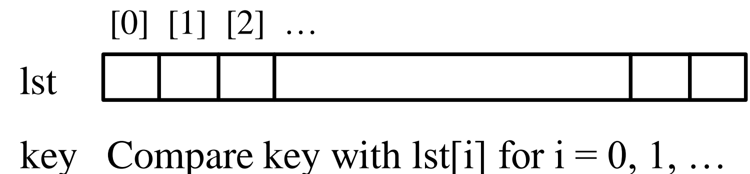
© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Searching Lists

Searching is the process of looking for a specific element in a list; for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching. In this section, two commonly used approaches are discussed, *linear search* and *binary search*.

```
# The function for finding a key in the list
def linearSearch(lst, key):
    for i in range(0, len(lst)):
        if key == lst[i]:
            return i

    return -1
```



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Linear Search

- The linear search approach compares the key element, key, *sequentially* with each element in list. The method continues to do so until the key matches an element in the list or the list is exhausted without a match being found.
- If a match is made, the linear search returns the index of the element in the list that matches the key.
- If no match is found, the search returns -1.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Linear Search Animation

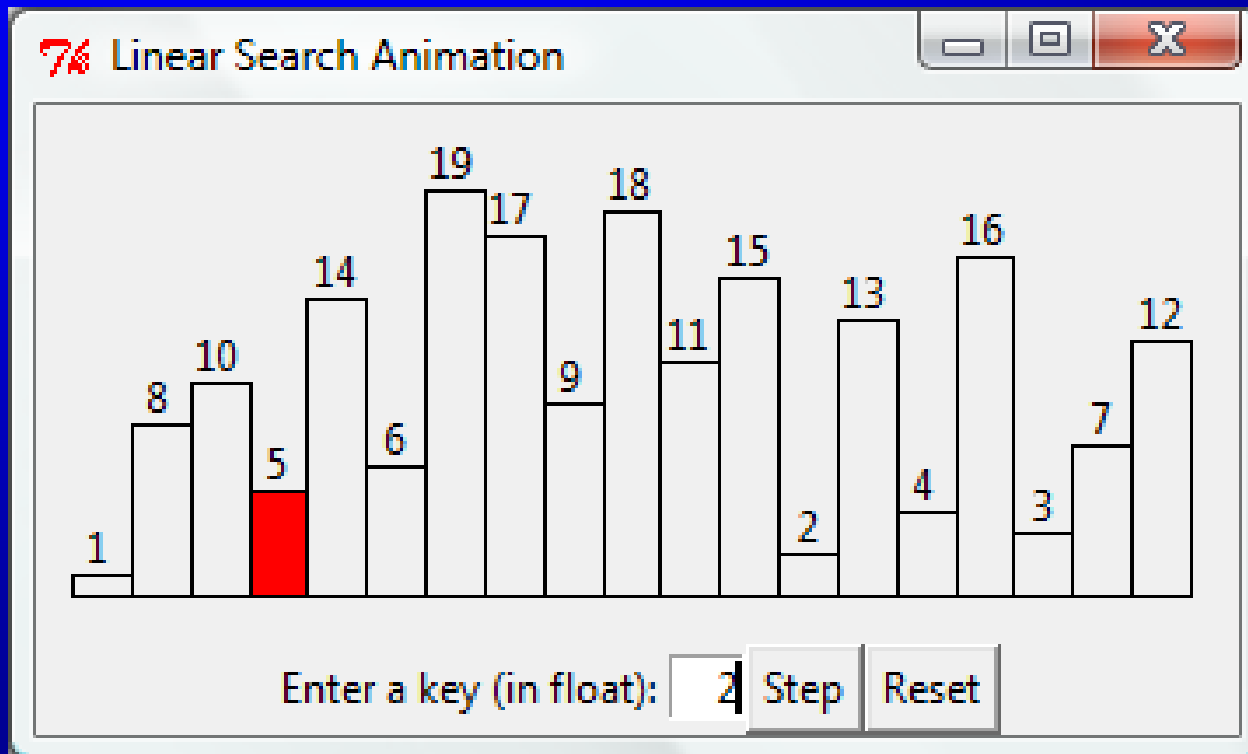
Key	List							
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8
3	6	4	1	9	7	3	2	8

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Linear Search Animation

http://www.cs.armstrong.edu/liang/animation/LinearSearchAnimation.html



Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Binary Search

- For binary search to work, the elements in the list must already be ordered. Without loss of generality, assume that the list is in ascending order.

  e.g., 2 4 7 10 11 45 50 59 60 66 69 70 79

- The binary search first compares the key with the element in the middle of the list.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Binary Search, cont.

Consider the following three cases:

- If the key is less than the middle element, you only need to search the key in the first half of the list.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you only need to search the key in the second half of the list.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Binary Search

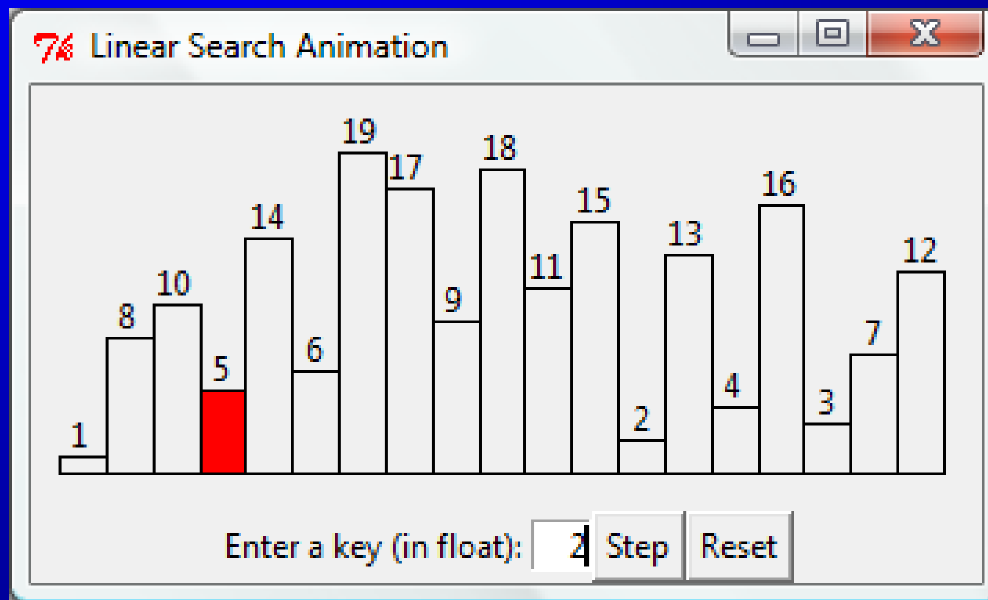
Key	List							
8	1	2	3	4	6	7	8	9
8	1	2	3	4	6	7	8	9
8	1	2	3	4	6	7	8	9

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Binary Search Animation

http://www.cs.armstrong.edu/liang/animation/BinarySearchAnimation.html



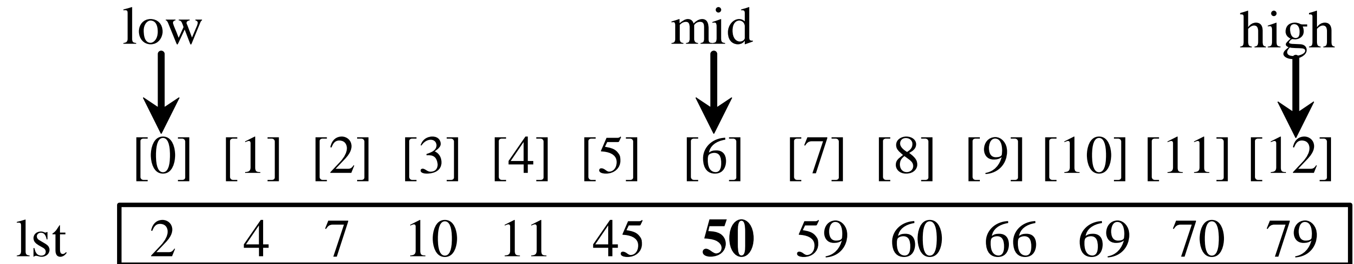
Run

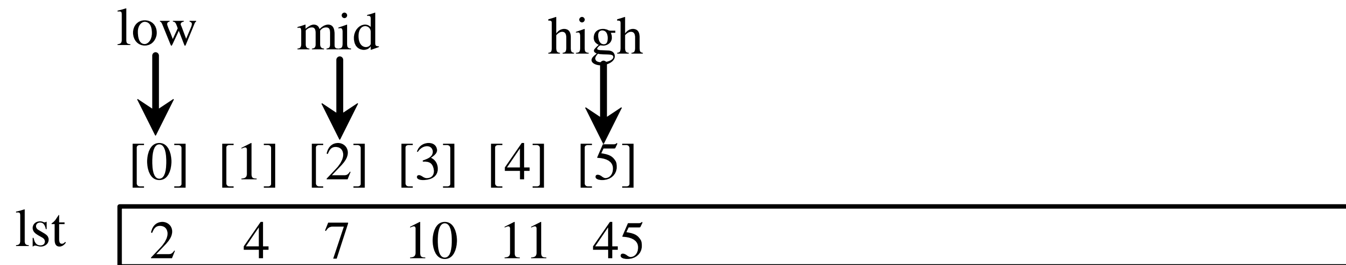
© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Binary Search, cont.

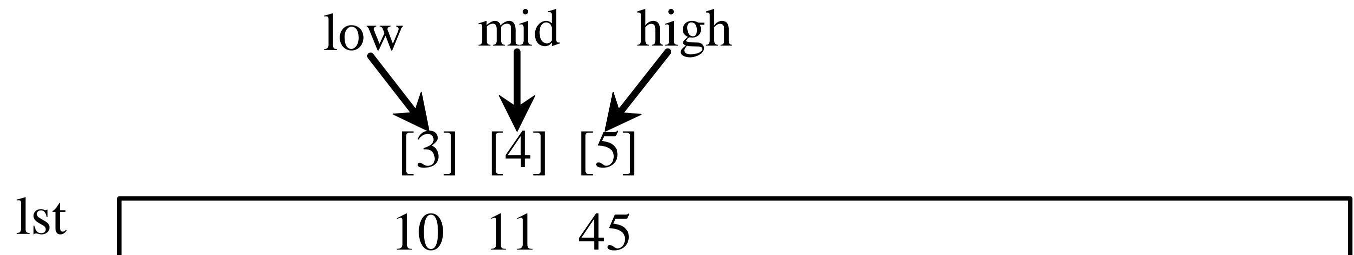
key is 11

key < 50





key > 7

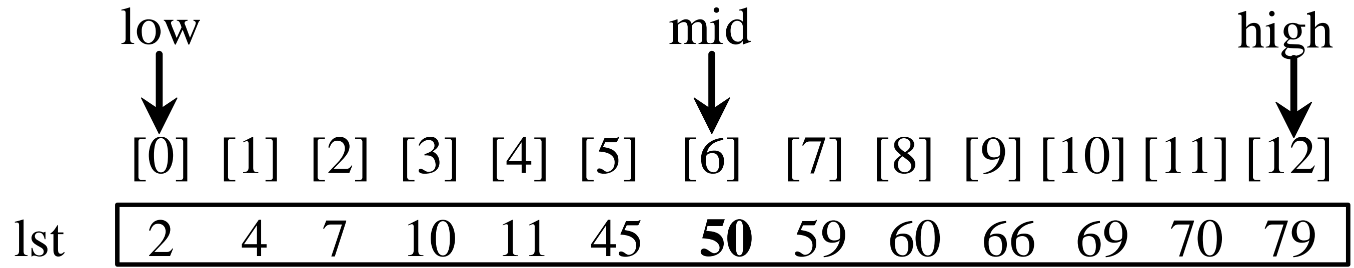


key == 11

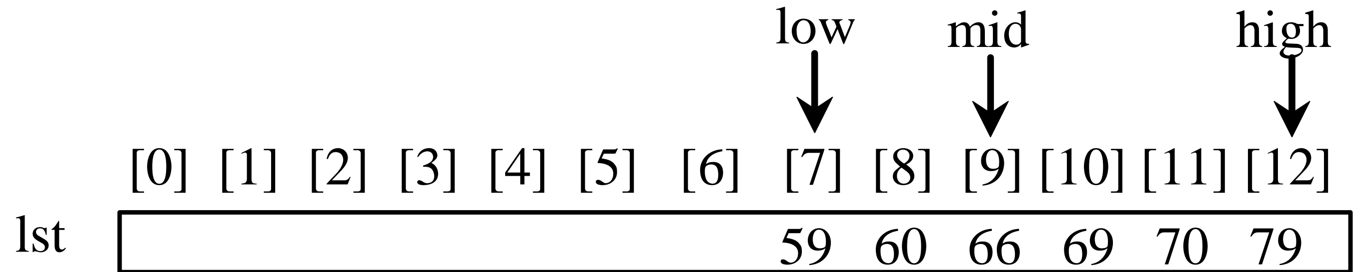
© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

key is 54

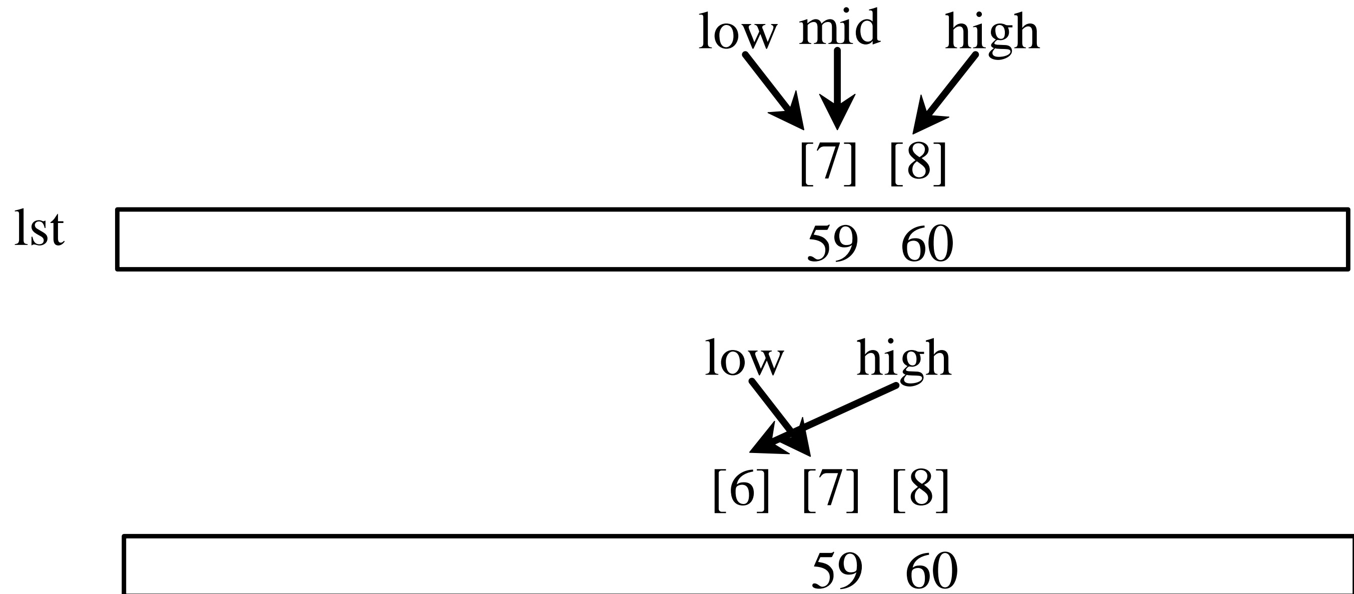
key > 50



key < 66



key < 59



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Binary Search, cont.

- The binarySearch method returns the index of the element in the list that matches the search key if it is contained in the list.
- Otherwise, it returns

  -insertion point - 1.

- The insertion point is the point at which the key would be inserted into the list.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

From Idea to Soluton

```
# Use binary search to find the key in the list
def binarySearch(lst, key):
    low = 0
    high = len(lst) - 1

    while high >= low:
        mid = (low + high) // 2
        if key < lst[mid]:
            high = mid - 1
        elif key == lst[mid]:
            return mid
        else:
            low = mid + 1

    return –low - 1 # Now high < low, key not found
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Logarithm: Analyzing Binary Search

$$n = 2^k$$

The binary search algorithm presented searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by c. Let $T(n)$ denote the time complexity for a binary search on a list of n elements. Without loss of generality, assume n is a power of 2 and $k=\log_2(n)$. Since binary search eliminates half of the input after two comparisons,

$$T(n) = T(\frac{n}{2}) + c = T(\frac{n}{2^2}) + c + c = ... = T(\frac{n}{2^k}) + ck = T(1) + c\log n = 1 + c\log n$$

$$= O(\log n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Logarithmic Time

- Ignoring constants and smaller terms, the complexity of the binary search algorithm is *O(logn)*.
- An algorithm with the *O(logn)* time complexity is called a *logarithmic algorithm*.
- The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted.
- The logarithmic algorithm grows slowly as the problem size increases.
- If you square the input size, you only double the time for the algorithm.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Hashing

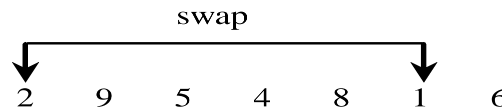
- In previous sections we were able to make improvements in our search algorithms by taking advantage of information about where items are stored in the collection with respect to one another.
- For example, by knowing that a list was ordered, we could search in logarithmic time using a binary search.
- In this section we will attempt to go one step further by building a data structure that can be searched in O(1) time. This concept is referred to as hashing.
- Study: https://runestone.academy/runestone/books/published/pythonds/SortSearch/Hashing.html

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

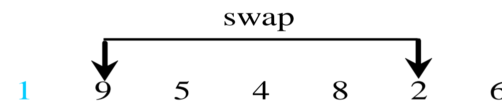
Selection Sort

Selection sort finds the smallest number in the list and places it at the beginning. It then finds the next smallest number in the remaining part and places it next to first, and so on until the list contains only a single number. Figure 16.17 shows how to sort the list {2, 9, 5, 4, 8, 1, 6} using selection sort.

Select 1 (the smallest) and swap it with 2 (the first) in the list

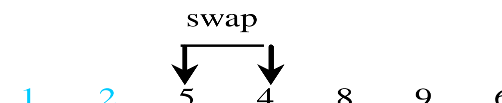


Select 2 (the smallest) and swap it with 9 (the first) in the remaining list



The number 1 is now in the correct position and thus no longer needs to be considered.

Select 4 (the smallest) and swap it with 5 (the first) in the remaining list



The number 2 is now in the correct position and thus no longer needs to be considered.

5 is the smallest and in the right position. No swap is necessary

The number 6 is now in the correct position and thus no longer needs to be considered.

Select 6 (the smallest) and swap it with 8 (the first) in the remaining list

The number 5 is now in the correct position and thus no longer needs to be considered.

Select 8 (the smallest) and swap it with 9 (the first) in the remaining list

The number 6 is now in the correct position and thus no longer needs to be considered.

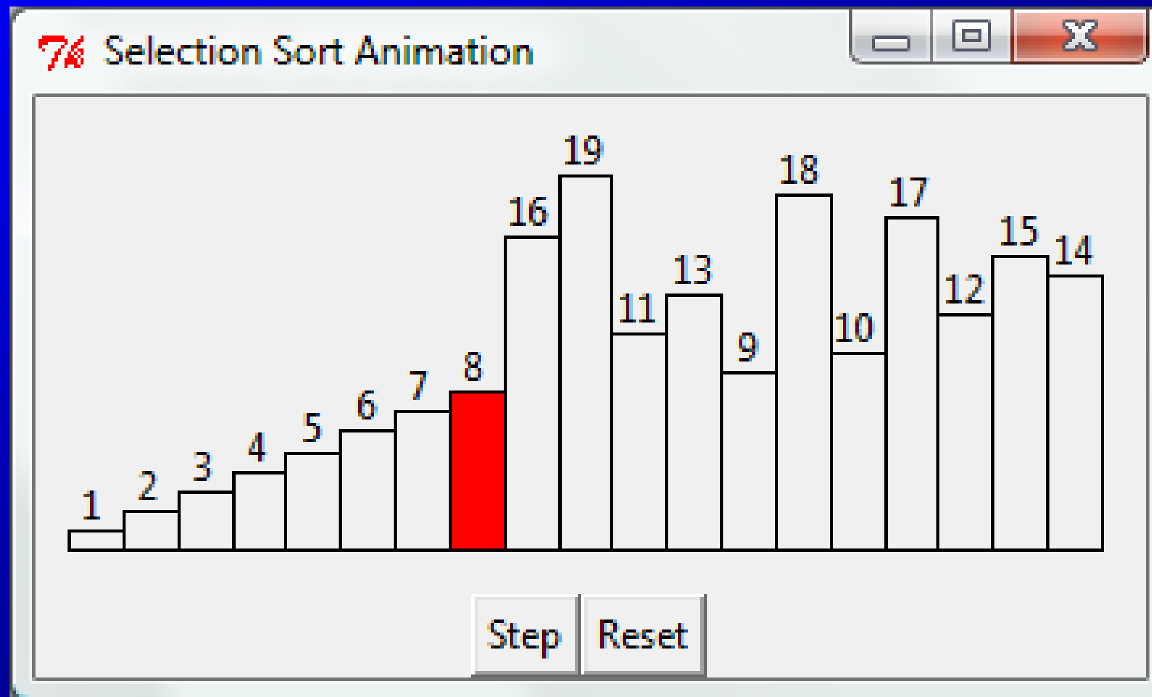
Since there is only one element remaining in the list, sort is completed

The number 8 is now in the correct position and thus no longer needs to be considered.

animation

Selection Sort Animation

http://www.cs.armstrong.edu/liang/animation/SelectionSortAnimation.html



Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

From Idea to Solution

```
for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]
```

```
lst[0] lst[1] lst[2] lst[3] ...                    lst[10]

lst[0] lst[1] lst[2] lst[3] ...                    lst[10]

lst[0] lst[1] lst[2] lst[3] ...                    lst[10]

lst[0] lst[1] lst[2] lst[3] ...                    lst[10]

lst[0] lst[1] lst[2] lst[3] ...                    lst[10]

                            ...

lst[0] lst[1] lst[2] lst[3] ...                    lst[10]
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

```
for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]
```

Expand

```
currentMin = lst[i]

     for j in range(i + 1, len(lst)):
        if currentMin > lst[j]:
           currentMin = lst[j]
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

```
for i in range(0, len(lst)):
    select the smallest element in lst[i.. len(lst)-1]
    swap the smallest with lst[i], if necessary
    # lst[i] is in its correct position.
    # The next iteration apply on lst[i+1..len(lst)-1]
```

Expand

```
# Find the minimum in the lst[i..len(lst)-1]
    currentMin = lst[i]
    currentMinIndex = i
    for j in range(i + 1, len(lst)):
        if currentMin > lst[j]:
            currentMin = lst[j]
            currentMinIndex = j
    # Swap lst[i] with lst[currentMinIndex] if necessary
    if currentMinIndex != i:
        lst[currentMinIndex] = lst[i]
        lst[i] = currentMin
```

Wrap it in a Function

```
# The function for sorting the numbers

def selectionSort(lst):
    for i in range(0, len(lst) - 1):
        # Find the minimum in the lst[i..len(lst)-1]
        currentMin = lst[i]
        currentMinIndex = i
        for j in range(i + 1, len(lst)):
            if currentMin > lst[j]:
                currentMin = lst[j]
                currentMinIndex = j
        # Swap lst[i] with lst[currentMinIndex] if necessary
        if currentMinIndex != i:
            lst[currentMinIndex] = lst[i]
            lst[i] = currentMin
```

Invoke (Call) it by selectionSort(yourList)

Analyzing Selection Sort

- The selection sort algorithm presented finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it next to first, and so on until the list contains only a single number.
- The number of *comparisons* is $n-1$ for the first iteration, $n-2$ for the second iteration, and so on.
- Let $T(n)$ denote the complexity for selection sort and c denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c ... + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

- Ignoring constants and smaller terms, the complexity of the selection sort algorithm is $O(n^2)$.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Quadratic Time

- An algorithm with the $O(n^2)$ time complexity is called a *quadratic algorithm*.
- The quadratic algorithm grows quickly as the problem size increases.
- If you double the input size, the time for the algorithm is quadrupled.
- Algorithms with a nested loop are often quadratic.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Insertion Sort

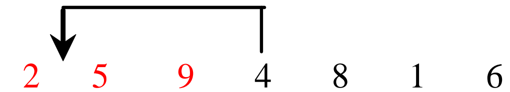
myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

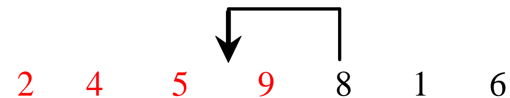
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: The sorted sublist is [2, 9]. Insert 5 into the sublist.

Step 3: The sorted sublist is [2, 5, 9]. Insert 4 into the sublist.



Step 4: The sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist.



Step 5: The sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist.

Step 6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist.

Step 7: The entire list is now sorted

animation

Insertion Sort

myList = [2, 9, 5, 4, 8, 1, 6] # Unsorted

2	9	5	4	8	1	6

2	9	5	4	8	1	6

2	5	9	4	8	1	6

2	4	5	9	8	1	6

2	4	5	8	9	1	6

1	2	4	5	8	9	6

1	2	4	5	6	8	9

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

How to Insert?

The insertion sort algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
list	2	5	9	4				Step 1: Save 4 to a temporary variable currentElement

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
list	2	5		9				Step 2: Move list[2] to list[3]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
list	2		5	9				Step 3: Move list[1] to list[2]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	
list	2	4	5	9				Step 4: Assign currentElement to list[1]

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

```
lst[0]
```

```
lst[0] lst[1]
```

```
lst[0] lst[1] lst[2]
```

```
lst[0] lst[1] lst[2] lst[3]
```

```
lst[0] lst[1] lst[2] lst[3] ...
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

From Idea to Solution

```
for i in range(1, len(lst)):
    insert lst[i] into a sorted sublist lst[0..i-1] so that
    lst[0..i] is sorted.
```

Expand

```
k = i - 1
while k >= 0 and lst[k] > currentElement:
           lst[k + 1] = lst[k]
           k -= 1
# Insert the current element into lst[k + 1]
lst[k + 1] = currentElement
```

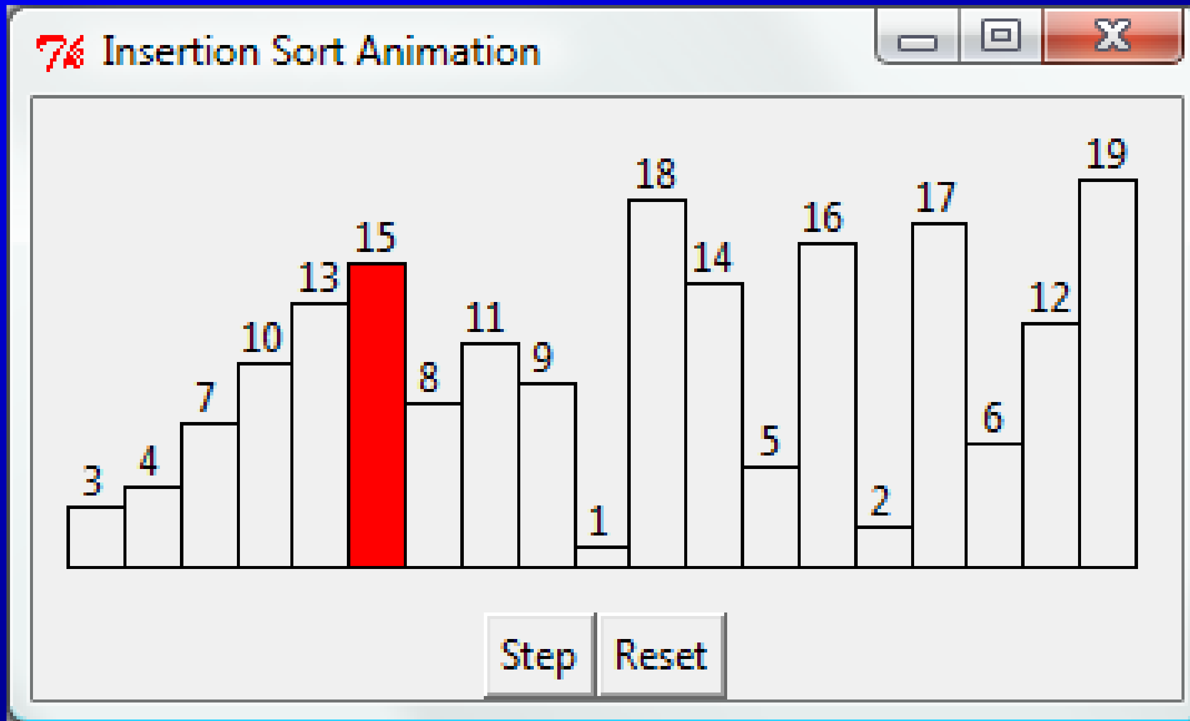
InsertSort

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Insertion Sort Animation

http://www.cs.armstrong.edu/liang/animation/InsertionSortAnimation.html



Run

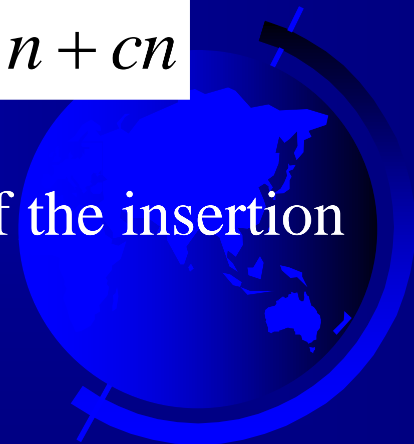
© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Analyzing Insertion Sort

The insertion sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted partial array until the whole array is sorted. At the *kth* iteration, to insert an element to a array of size *k*, it may take *k* comparisons to find the insertion position, and *k* moves to insert the element. Let *T(n)* denote the complexity for insertion sort and *c* denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = 2 + c + 2 \times 2 + c \ldots + 2 \times (n-1) + c = n^2 - n + cn$$

Ignoring constants and smaller terms, the complexity of the insertion sort algorithm is $O(n^2)$.



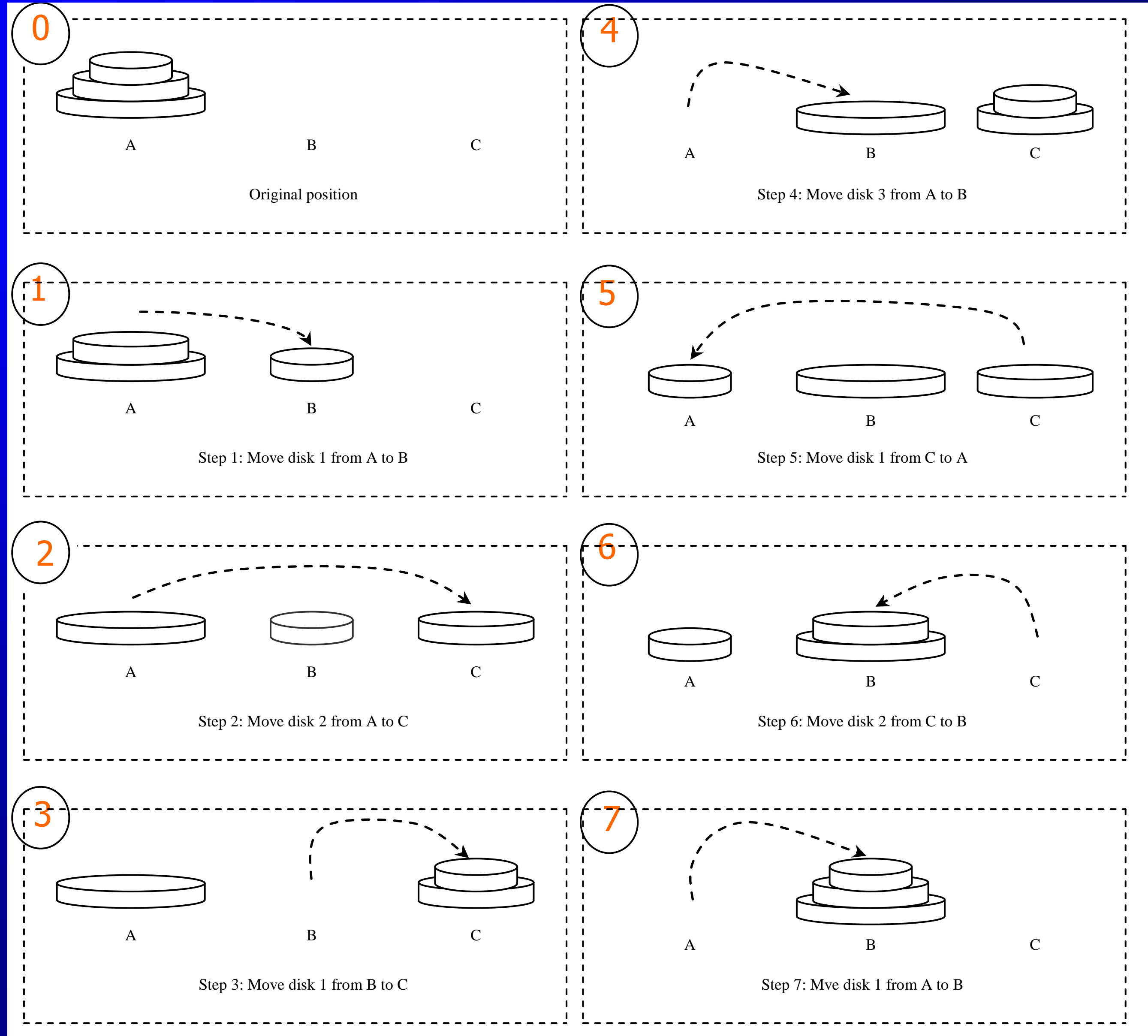
© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Towers of Hanoi

- There are n disks labeled 1, 2, 3, . . ., n, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

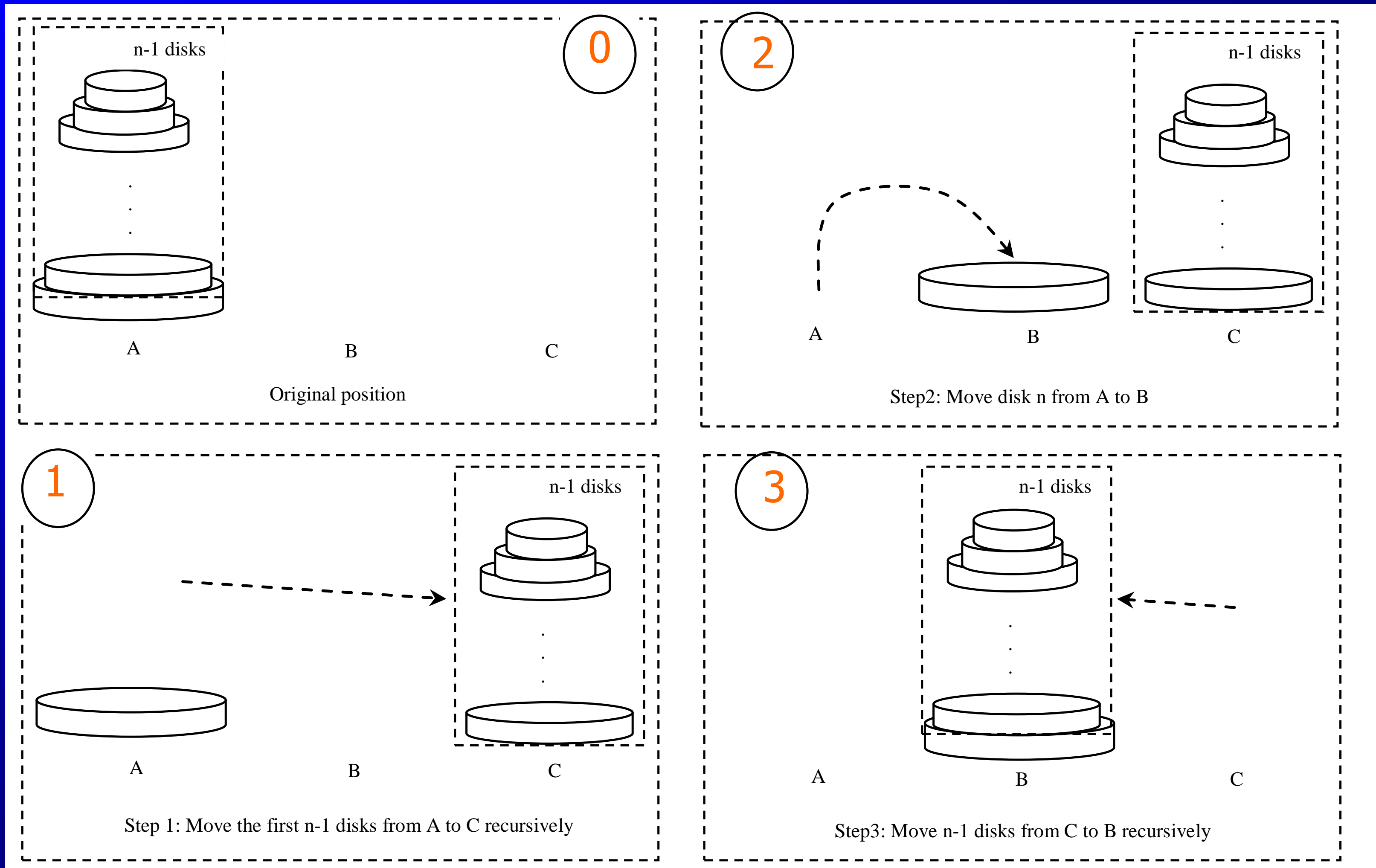
Towers of Hanoi, cont.



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solution to Towers of Hanoi

- Move the first n - 1 disks from A to C with the assistance of tower B.
- Move disk n from A to B.
- Move n - 1 disks from C to B with the assistance of tower A.

TowersOfHanoi

Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Towers of Hanoi

```
def main():
    n = eval(input("Enter number of disks: "))

    # Find the solution recursively
    print("The moves are:")
    moveDisks(n, 'A', 'B', 'C')

# The function for finding the solution to move n disks
#   from fromTower to toTower with auxTower
def moveDisks(n, fromTower, toTower, auxTower):
    if n == 1: # Stopping condition
        print("Move disk", n, "from", fromTower, "to", toTower)
    else:
        moveDisks(n - 1, fromTower, auxTower, toTower)
        print("Move disk", n, "from", fromTower, "to", toTower)
        moveDisks(n - 1, auxTower, toTower, fromTower)

main() # Call the main function
```

- Move the first n - 1 disks from A to C with the assistance of tower B.
- Move disk n from A to B.
- Move n - 1 disks from C to B with the assistance of tower A.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Analyzing Towers of Hanoi

The Towers of Hanoi problem presented, moves *n* disks from tower A to tower B with the assistance of tower C recursively as follows:

- Move the first $n - 1$ disks from A to C with the assistance of tower B.
- Move disk *n* from A to B.
- Move $n - 1$ disks from C to B with the assistance of tower A.

Let $T(n)$ denote the complexity for the algorithm that moves disks and c denote the constant time to move one disk, i.e., T(1) is c. So,

$$T(n) = T(n-1) + c + T(n-1) = 2T(n-1) + c$$
$$= 2(2(T(n-2) + c) + c) = 2^{n-1}T(1) + c2^{n-2} + ... + c2 + c =$$
$$= c2^{n-1} + c2^{n-2} + ... + c2 + c = c(2^n - 1) = O(2^n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Common Recurrence Relations

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 17)
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Comparing Common Growth Functions

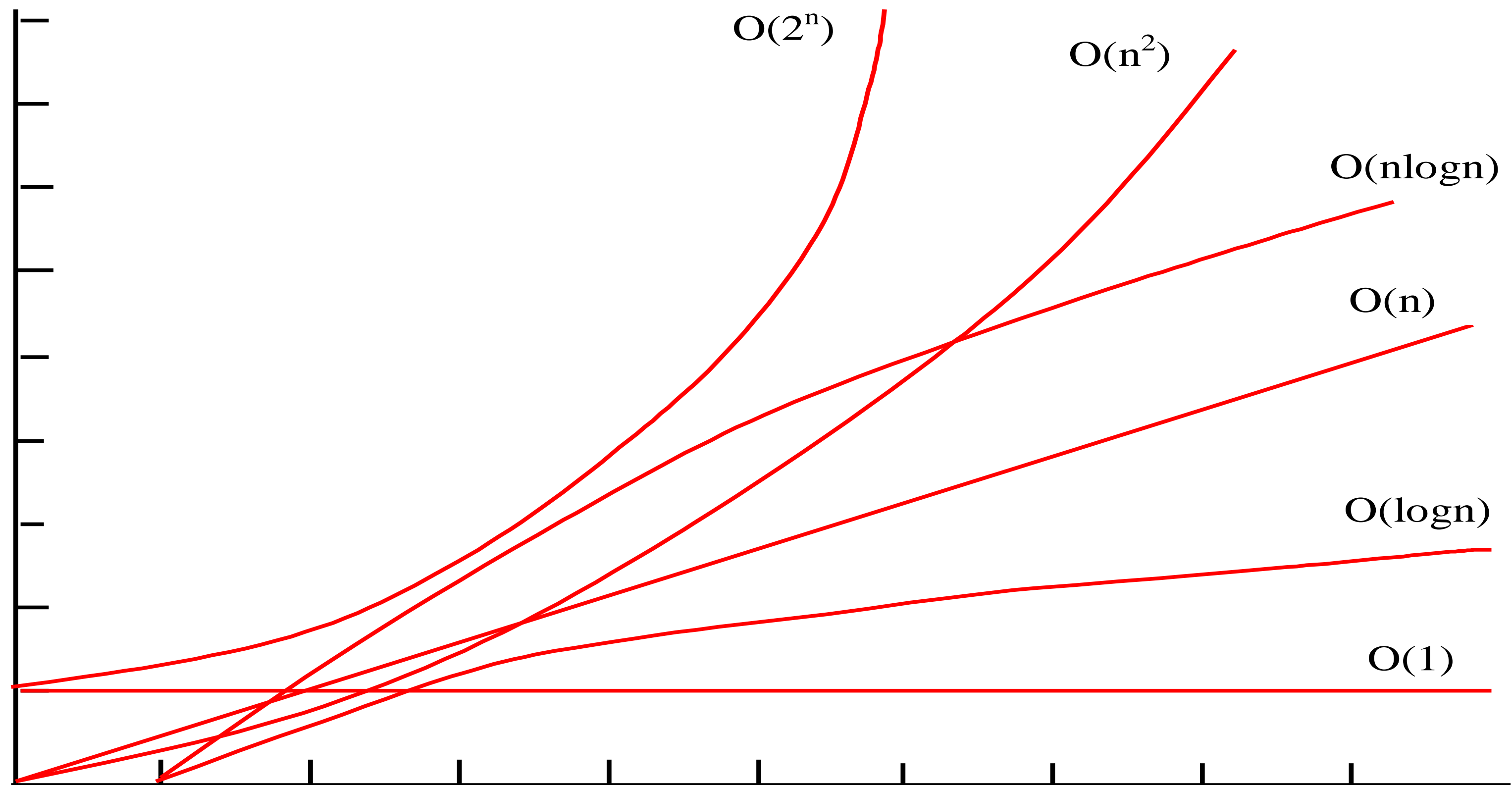
$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: Fibonacci Numbers

```
/** The method for finding the Fibonacci number */
public static long fib(long index) {
  if (index == 0) // Base case
    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
}
```

```
Finonacci series: 0 1 1 2 3 5 8 13 21  34  55 89…
         indices: 0 1 2 3 4 5 6 7   8   9   10 11
```

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Complexity for Recursive Fibonacci Numbers

Since

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\le 2T(n-1) + c \\
&\le 2(2T(n-2) + c) + c \\
&= 2^2 T(n-2) + 2c + c \\
&\ldots \\
&\le 2^{n-1}T(1) + 2^{n-2}c + \ldots + 2c + c \\
&= 2^{n-1}T(1) + (2^{n-2} + \ldots + 2 + 1)c \\
&= 2^{n-1}T(1) + (2^{n-1} - 1)c \\
&= 2^{n-1}c + (2^{n-2} + \ldots + 2 + 1)c \\
&= O(2^n)
\end{aligned}
$$

and

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&= T(n-2) + T(n-3) + c + T(n-2) + c \\
&\ge 2T(n-2) + 2c \\
&\ge 2(2T(n-4) + 2c) + 2c \\
&\ge 2^2 T(n-2-2) + 2^2 c + 2c \\
&\ge 2^3 T(n-2-2-2) + 2^3 c + 2^2 c + 2c \\
&\ge 2^{n/2} T(1) + 2^{n/2} c + \ldots + 2^3 c + 2^2 c + 2c \\
&= 2^{n/2} c + 2^{n/2} c + \ldots + 2^3 c + 2^2 c + 2c \\
&= O(2^n)
\end{aligned}
$$

Therefore, the recursive Fibonacci method takes

$$O(2^n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Recursive Version is inefficient

- Recursive algorism for Fibonacci is not efficient.
- The trouble with the recursive fib method is that the method is invoked redundantly with the same arguments.
- For example, to compute **fib(4)**, **fib(3)** and **fib(2)** are invoked. To compute **fib(3)**, **fib(2)** and **fib(1)** are invoked. Note that **fib(2)** is redundantly invoked.
- We can improve it by avoiding repeatedly calling of the **fib** method with the same argument.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Finding Fibonacci numbers using dynamic programming

- Note that a new Fibonacci number is obtained by adding the preceding two numbers in the sequence.
- If you use the two variables **f0** and **f1** to store the two preceding numbers, the new number, **f2**, can be immediately obtained by adding **f0** with **f1**.
- Now you should update **f0** and **f1** by assigning **f1** to **f0** and assigning **f2** to **f1**, as shown below.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

```
                 f0  f1  f2
Fibonacci series: 0   1   1   2  3  5  8  13  21  34  55  89...

         indices: 0   1   2   3  4  5  6  7   8   9   10  11


                   f0  f1  f2
Fibonacci series: 0   1   1   2   3   5   8   13   21  34  55  89...

         indices: 0   1   2   3   4   5   6   7    8   9   10  11


                      f0  f1  f2
Fibonacci series: 0   1   1   2   3   5   8   13   21   34   55  89...

         indices: 0   1   2   3   4   5   6   7    8    9    10  11


                                            f0  f1  f2
Fibonacci series: 0   1   1   2   3   5   8   13  21  34  55  89...

         indices: 0   1   2   3   4   5   6   7   8   9   10  11
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: Non-recursive version of Fibonacci Numbers

```
def fib(n):
    f0 = 0  # For fib(0)
    f1 = 1  # For fib(1)
    f2 = 1  # For fib(2)

    if n == 0:
        return f0
    elif n == 1:
        return f1
    elif n == 2:
        return f2
    for i in range(3, n + 1):
        f0 = f1
        f1 = f2
        f2 = f0 + f1

    return f2
```

Obviously, the complexity of this new algorithm is $O(n)$. This is a tremendous improvement over the recursive algorithm.

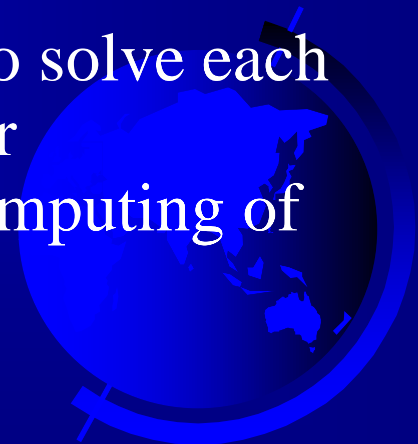
Improved Fibonacci

Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Dynamic Programming

- The algorithm for computing Fibonacci numbers presented here uses an approach known as *dynamic programming*.
- Dynamic programming is to solve subproblems, then combine the solutions of subproblems to obtain an overall solution.
- This naturally leads to a recursive solution. However, it would be inefficient to use recursion, because the subproblems overlap.
- The key idea behind dynamic programming is to solve each subprogram only once and storing the results for subproblems for later use to avoid redundant computing of the subproblems.



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

The search for an efficient algorithm for finding the greatest common divisor of two integers.

- The greatest common divisor (GCD) of two integers is the largest number that can evenly divide both integers.
- The method on the next slide presents a brute-force algorithm for finding the greatest common divisor of two integers m and n.
- Brute force refers to an algorithmic approach that solves a problem in the simplest or most direct or obvious way.
- As a result, such an algorithm can end up doing far more work to solve a given problem than a cleverer or more sophisticated algorithm might do.
- On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: GCD Algorithms Version 1

```
def gcd(m, n):
    gcd = 1
    k = 2
    while k <= m and k <= n:
        if m % k == 0 and n % k == 0:
            gcd = k
        k += 1
    return gcd
```

Obviously, the complexity of this algorithm is $O(n)$.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Improved Solutions

- Is there a better algorithm for finding the GCD?
- Rather than searching a possible divisor from **1** up, it is more efficient to search from **n** down.
- Once a divisor is found, the divisor is the GCD.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: GCD Algorithms Version 2

```
for k in range(n, 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break
```

The worst-case time complexity of this algorithm is still $O(n)$.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Improved Solutions

- Is there a better algorithm for finding the GCD?
- A divisor for a number n cannot be greater than n / 2, so you can further improve the algorithm changing the initial value of the loop

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: GCD Algorithms Version 3

```
for k in range(int(m / 2), 0, -1):
    if m % k == 0 and n % k == 0:
        gcd = k
        break
```

The worst-case time complexity of this algorithm is still $O(n)$.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Improved Solutions

- Is there a better algorithm for finding the GCD?
- A more efficient algorithm for finding the GCD was discovered by Euclid around 300 b.c. This is one of the oldest known algorithms.
- It can be defined recursively as follows:

  Let gcd(m, n) denote the gcd for integers m and n:
  - If m % n is 0, gcd (m, n) is n.
  - Otherwise, gcd(m, n) is gcd(n, m % n).

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Euclid's Algorithm Implementation

```
def gcd(m, n):
    if m % n == 0:
        return n
    else:
        return gcd(n, m % n)
```

The time complexity of this algorithm is O(logn).

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

The time complexity

- In the best case when **m % n** is **0**, the algorithm takes just one step to find the GCD.
- It is difficult to analyze the average case.
- However, we can prove that the worst-case time complexity is O(log n).

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

The worst case time complexity

Assuming $m \geq n$, we can show that `m % n < m / 2`, as follows:

- If `n <= m / 2`, `m % n < m / 2`, since the remainder of m divided by n is always less than n.
- If `n > m / 2`, `m % n = m - n < m / 2`. Therefore, `m % n < m / 2`.

Euclid's algorithm recursively invokes the `gcd` method. It first calls `gcd(m, n)`, then calls `gcd(n, m % n)`, and `gcd(m % n, n % (m % n))`, and so on, as follows:

```
  gcd(m, n)
= gcd(n, m % n)
= gcd(m % n, n % (m % n))
= ...
```

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

The worst case time complexity

Since `m % n < m / 2` and `n % (m % n) < n / 2`, the argument passed to the `gcd` method is reduced by half after every two iterations. After invoking `gcd` two times, the second parameter is less than $n/2$. After invoking `gcd` four times, the second parameter is less than $n/4$. After invoking `gcd` six times, the second parameter is less than $\frac{n}{2^3}$. Let k be the number of times the `gcd` method is invoked. After invoking `gcd` k times, the second parameter is less than $\frac{n}{2^{(k/2)}}$, which is greater than or equal to 1. That is,

$$\frac{n}{2^{(k/2)}} \geq 1 \quad => \quad n \geq 2^{(k/2)} \quad => \quad \log n \geq k/2 \quad => \quad k \leq 2 \log n$$

Therefore, $k \leq 2 \log n$. So the time complexity of the `gcd` method is $O(\log n)$.

The worst case occurs when the two numbers result in the most divisions. It turns out that two successive Fibonacci numbers will result in the most divisions. Recall that the Fibonacci series begins with 0 and 1, and each subsequent number is the sum of the preceding two numbers in the series, such as:

0 1 1 2 3 5 8 13 21 34 55 89 . . .

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

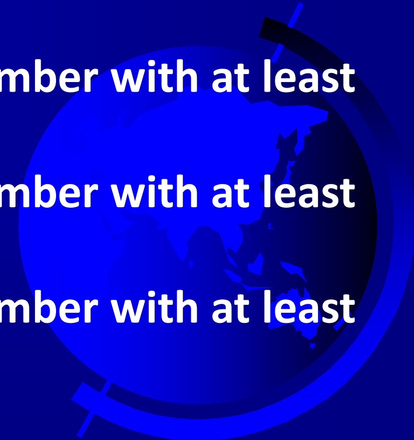
Efficient Algorithms for Finding Prime Numbers

- https://www.eff.org/awards/coop

The Electronic Frontier Foundation (EFF), the first civil liberties group dedicated to protecting the health and growth of the Internet, is sponsoring cooperative computing awards, with over half a million dollars in prize money, to encourage ordinary Internet users to contribute to solving huge scientific problems.

Through the EFF Cooperative Computing Awards, EFF will confer prizes of:

- **$50,000** to the first individual or group who discovers **a prime number with at least 1,000,000 decimal digits** (awarded Apr. 6, 2000)
- **$100,000** to the first individual or group who discovers **a prime number with at least 10,000,000 decimal digits** (awarded Oct. 22, 2009)
- **$150,000** to the first individual or group who discovers **a prime number with at least 100,000,000 decimal digits**
- **$250,000** to the first individual or group who discovers **a prime number with at least 1,000,000,000 decimal digits**



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solutions

- A brute-force algorithm for finding prime numbers:
  - The algorithm checks whether **2**, **3**, **4**, **5**, . . . , or **n – 1** is divisible by **n**. If not, **n** is prime. This algorithm takes $O(n)$ time to check whether **n** is prime.
- Note that you need to check only whether **2**, **3**, **4**, **5**, . . . , and **n/2** is divisible by **n**. If not, **n** is prime. This algorithm is slightly improved, but it is still of $O(n)$.
- Soluton 1: Brute-force Algorithm: PrimeNumber

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solutions

- In fact, we can prove that if **n** is not a prime, **n** must have a factor that is greater than **1** and less than or equal to $\sqrt{n}$.
- Solution2 : Algorithm that checks possible divisors up to Math.sqrt(n)

PrimeNumbers

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solutions

- In fact, there is no need to actually compute Math.sqrt(number) for every number. A good compiler should evaluate Math.sqrt(number) only once for the entire for loop. But, you can not be sure. Change the algoritm so that Math.sqrt(number) will be evaluated only once for the entire for loop.
- You need look only for the perfect squares such as 4, 9, 16, 25, 36, 49, and so on.
- Note that for all the numbers between 36 and 48, inclusively, their (int)(Math.sqrt(number)) is 6.
- To determine whether i is prime, the algorithm checks whether 2, 3, 4, 5, . . . , and $\sqrt{i}$ are divisible by i.
- This algorithm can be further improved. In fact, you need to check only whether the prime numbers from 2 to $\sqrt{i}$ are possible divisors for i.
- With these insights, you can develop a more efficient solution.
- Solution 3 : A more efficient algorithm

EfficientPrimeNumbers

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solutions

- Let us examine the well-known Eratosthenes algorithm for finding prime numbers.
- Eratosthenes (276–194 b.c.) was a Greek mathematician who devised a clever algorithm, known as the *Sieve of Eratosthenes*, for finding all prime numbers <= *n*.
- His algorithm is to use an array named primes of n Boolean values. Initially, all elements in primes are set true.
- Since the multiples of 2 are not prime, set primes[2 * i] to false for all 2 <= i <= n/2, as shown in the Figure. Since we don’t care about primes[0] and primes[1], these values are marked x in the figure.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Sieve of Eratosthenes

- Since the multiples of **3** are not prime, set **primes[3 * i]** to **false** for all 3 <= *i* <= *n*/3. Because the multiples of **5** are not prime, set **primes[5 * i]** to **false** for all 5 <= *i* >= *n*/5.
- Note that you don't need to consider the multiples of **4**, because the multiples of **4** are also the multiples of **2**, which have already been considered. Similarly, multiples of **6**, **8**, and **9** need not be considered.
- You only need to consider the multiples of a prime number **k = 2**, **3**, **5**, **7**,**11**, . . . , and set the corresponding element in **primes** to **false**.
- Afterward, if **primes[i]** is still true, then **i** is a prime number. As shown in Figure 22.3, **2**, **3**, **5**, **7**, **11**, **13**, **17**, **19**, and **23** are prime numbers.

primes array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	×	×	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k = 2$	×	×	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k = 3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k = 5$	×	×	(T)	(T)	F	(T)	F	(T)	F	F	F	(T)	F	(T)	F	F	F	(T)	F	(T)	F	F	F	(T)	F	F	F	F

Solution 4 : SieveOfEratosthenes Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Finding Prime Numbers

Compare three versions:

- Brute-force PrimeNumber PrimeNumbers
- Check possible divisors up to Math.sqrt(n)
- Check possible prime divisors up to Math.sqrt(n) EfficientPrimeNumbers

SieveOfEratosthenes

Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Time Complexities

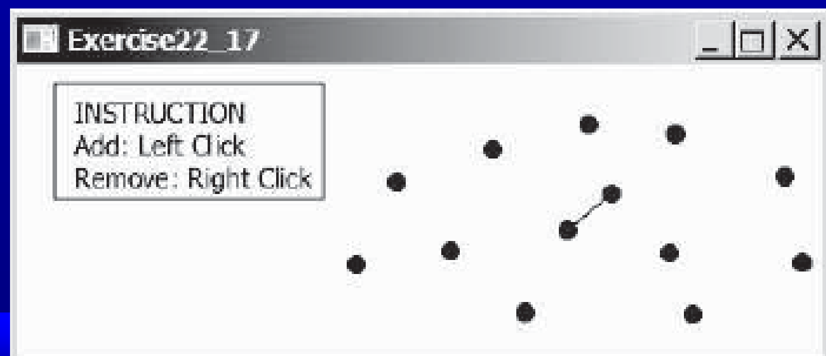
- Comparisons of Prime-Number Algorithms

Complexity	*Description*
$O(n^2)$	Brute-force, checking all possible divisors
$O(n\sqrt{n})$	Checking divisors up to $\sqrt{n}$
$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Checking prime divisors up to $\sqrt{n}$
$O\left(\frac{n\sqrt{n}}{\log n}\right)$	Sieve of Eratosthenes

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Finding the Closest Pair of Points Using Divide-and-Conquer

- A brute-force algorithm for finding the closest pair of points computes the distances between all pairs of points and finds the one with the minimum distance.
- Clearly, the algorithm takes $O(n^2)$ time.
- Can we design a more efficient algorithm?



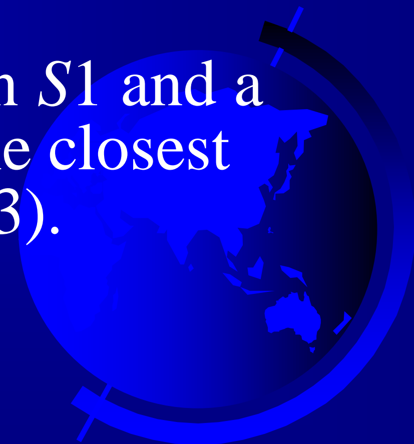
Divide-and-Conquer

- The *divide-and-conquer* approach divides the problem into subproblems, solves the subproblems, then combines the solutions of subproblems to obtain the solution for the entire problem.
- Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don’t overlap.
- A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem. In fact, all the recursive problems follow the divide-and-conquer approach.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

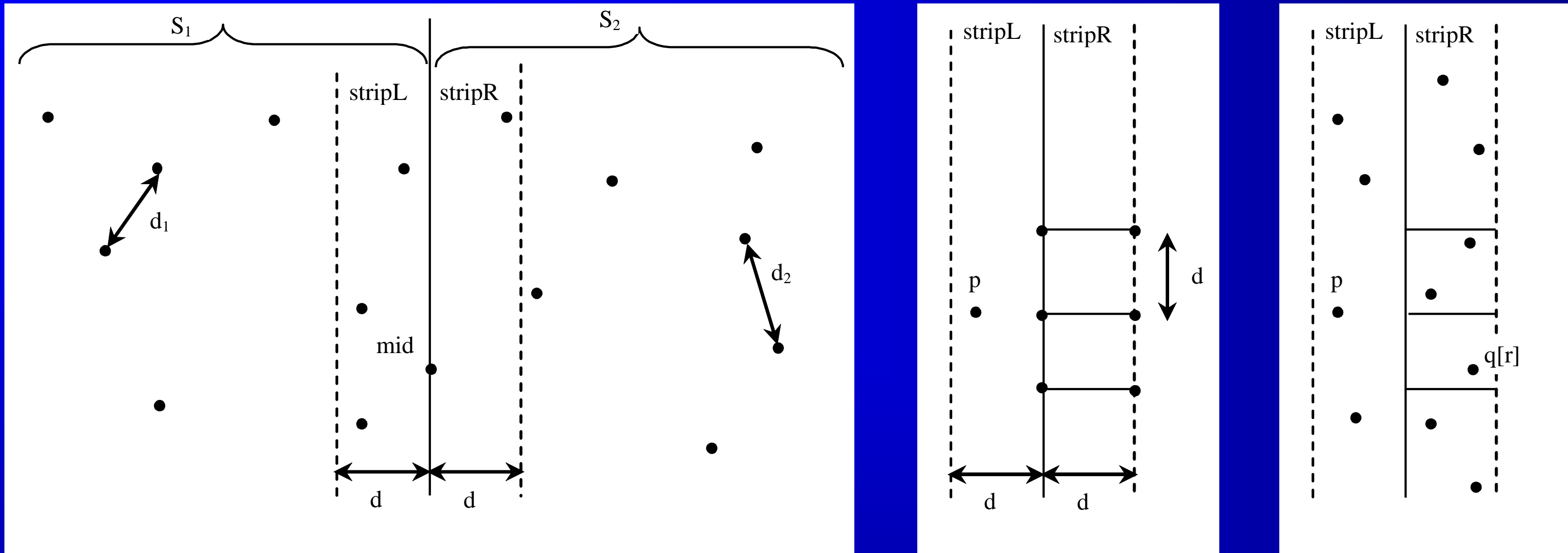
Divide-and-conquer algorithm for Finding the Closest Pair of Points

- Step 1: Sort the points in increasing order of *x*-coordinates. For the points with the same *x*-coordinates, sort on *y*-coordinates. This results in a sorted list *S* of points.
- Step 2: Divide *S* into two subsets, *S*1 and *S*2, of equal size using the midpoint in the sorted list. Let the midpoint be in *S*1. Recursively find the closest pair in *S*1 and *S*2. Let *d*1 and *d*2 denote the distance of the closest pairs in the two subsets, respectively.
- Step 3: Find the closest pair between a point in *S*1 and a point in *S*2 and denote their distance as *d*3. The closest pair is the one with the distance min(*d*1, *d*2, *d*3).



© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: Closest Pair of Points

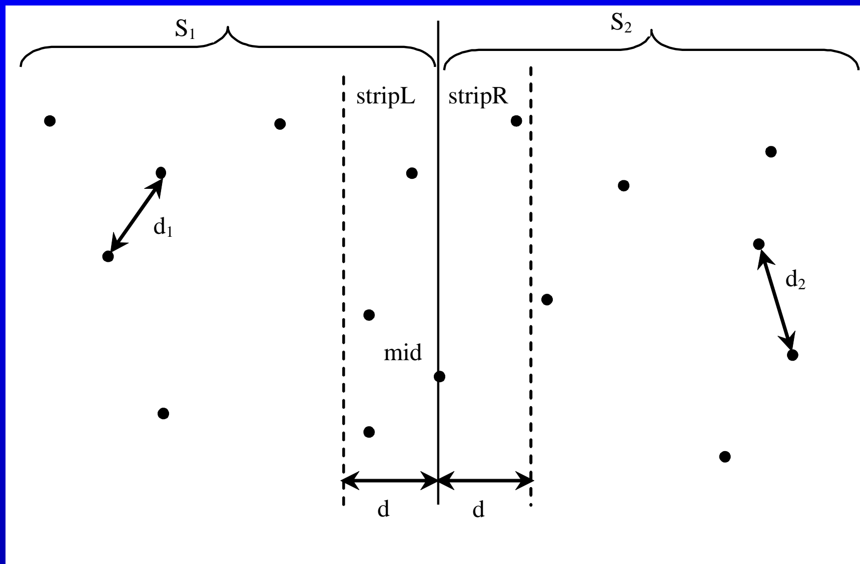


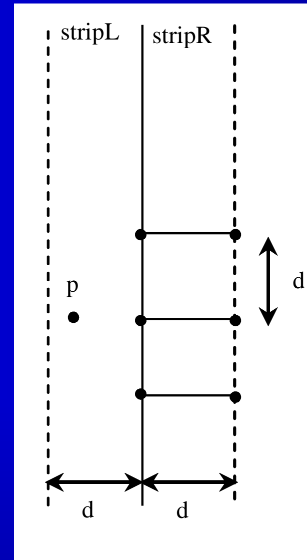
Step 3 can be done in $O(n)$ time. Let $d = \min(d1, d2)$. We already know that the closestpair distance cannot be larger than d. For a point in $S1$ and a point in $S2$ to form the closest pair in S, the left point must be in **stripL** and the right point in **stripR**, as illustrated in Figure

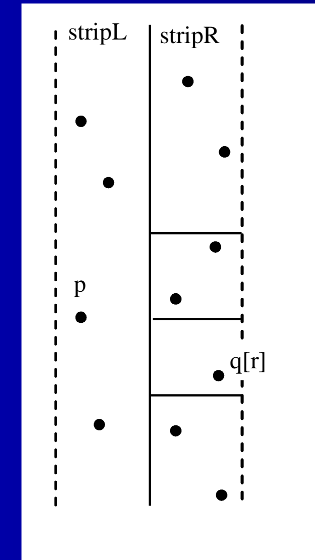
$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Case Study: Closest Pair of Points







$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Solving the Eight Queens Problem Using Backtracking

- The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. The problem can be solved using recursion.
- In this section, we will introduce a common algorithm design technique called backtracking for solving this problem.
- The backtracking approach searches for a candidate solution incrementally, abandoning that option as soon as it determines that the candidate cannot possibly be a valid solution, and then looks for a new candidate.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Eight Queens Problem Using Backtracking

The search starts from the first row with $k = 0$, where k is the index of the current row being considered. The algorithm checks whether a queen can be possibly placed in the jth column in the row for $j = 0, 1, \ldots, 7$, in this order. The search is implemented as follows:

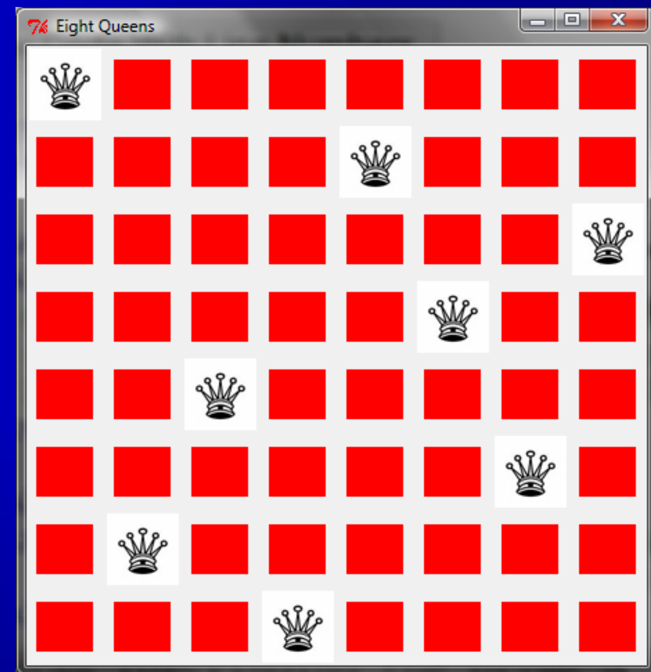
- If successful, it continues to search for a placement for a queen in the next row. If the current row is the last row, a solution is found.
- If not successful, it backtracks to the previous row and continues to search for a new placement in the next column in the previous row.
- If the algorithm backtracks to the first row and cannot find a new placement for a queen in this row, no solution can be found.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Eight Queens

You can use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row.

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3



EightQueens

Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

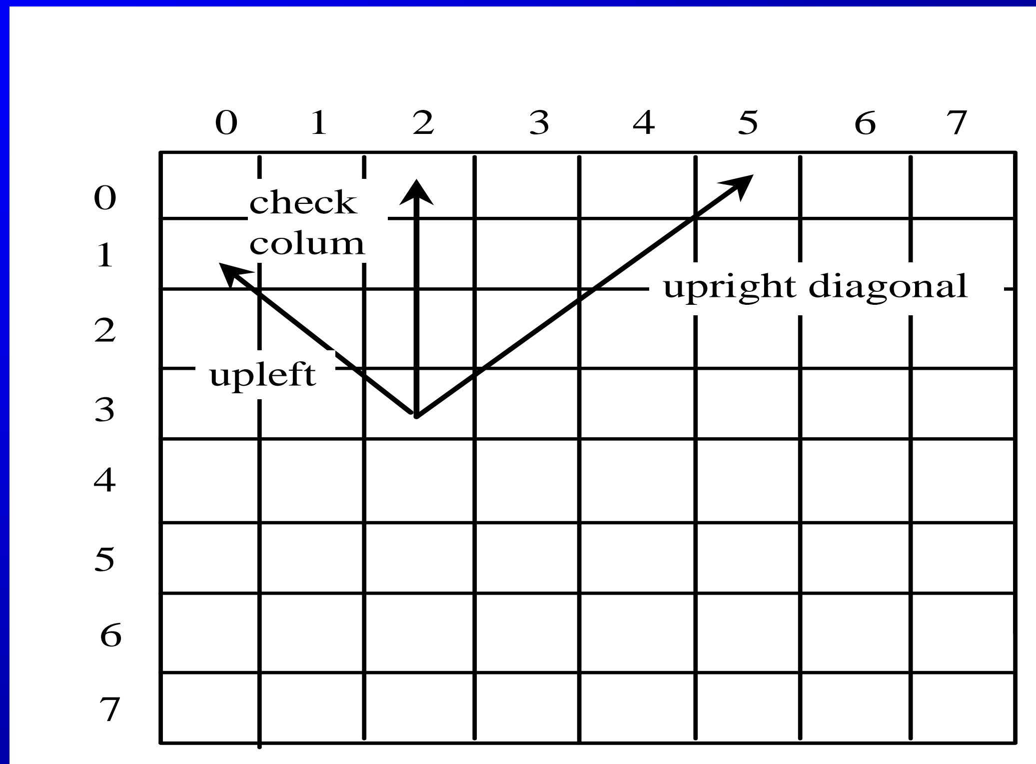
Backtracking

There are many possible candidates? How do you find a solution? The backtracking approach is to search for a candidate incrementally and abandons it as soon as it determines that the candidate cannot possibly be a valid solution, and explores a new candidate.

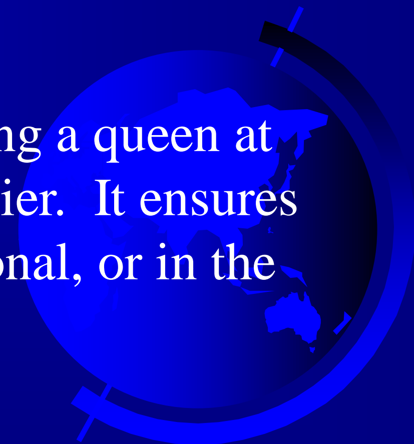
http://www.cs.armstrong.edu/liang/animation/EightQueensAnimation.html

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Eight Queens



The isValid(row, column) method is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier. It ensures that no queen is placed in the same column, in the upper-left diagonal, or in the upper-right diagonal , as shown in Figure.

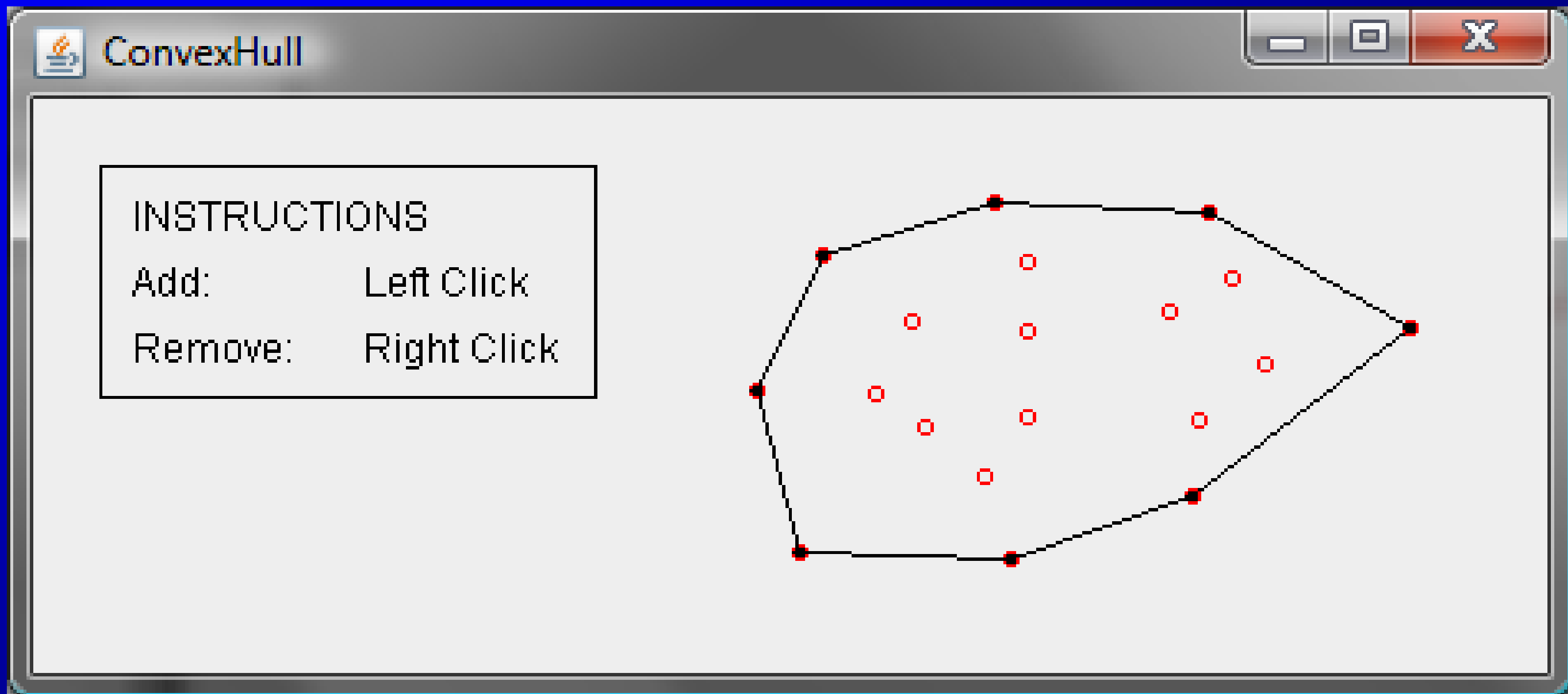


© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

animation

Convex Hull Animation

http://www.cs.armstrong.edu/liang/animation/ConvexHull.html



Run

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Convex Hull

Given a set of points, a convex hull is a smallest convex polygon that encloses all these points, as shown in Figure a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices v0, v1, v2, v3, v4, and v5 in Figure a form a convex polygon, but not in Figure b, because the line that connects v3 and v1 is not inside the polygon.

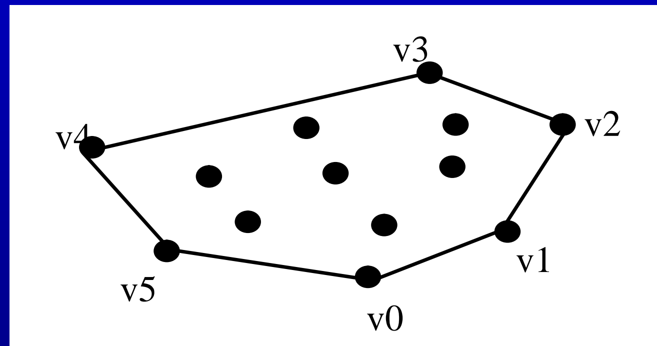


Figure a

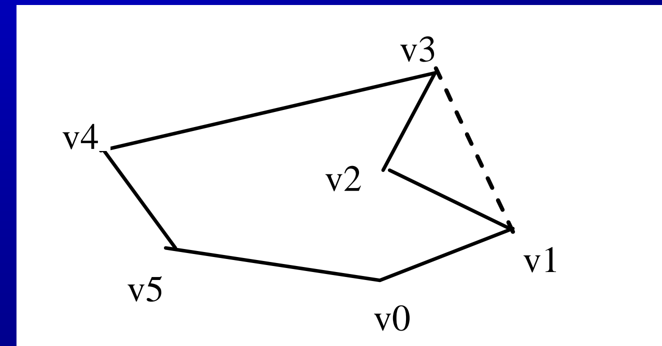
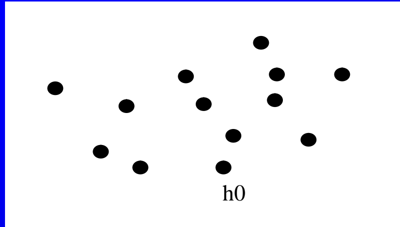


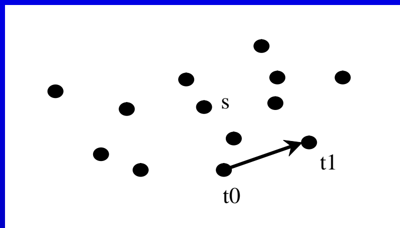
Figure b

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

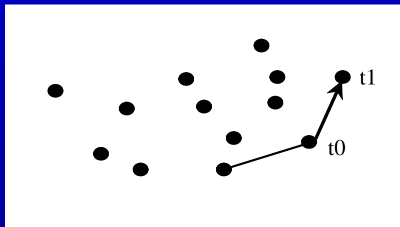
Gift-Wrapping

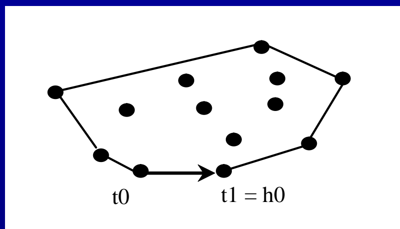


Step 1: Given a set of points S, let the points in S be labeled s0, s1, ..., sk. Select the rightmost lowest point h0 in the set S. Add h0 to list H. (H is initially empty. H will hold all points in the convex hull after the algorithm is finished.) Let t0 be h0.



(Step 2: Find the rightmost point t1): Let t1 be s0. For every point p in S. If p is on the right side of the direct line from t0 to t1 then let t1 be p. (After Step 2, no points lie on the right side of the direct line from t0 to t1, as shown in Figure)





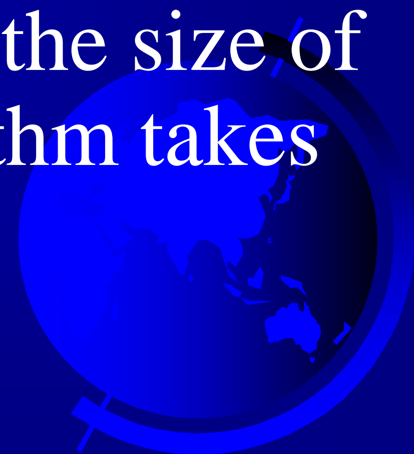
Step 3: If t1 is h0, done. the points in H form a convex hull for S.

Otherwise: add t1 to H. Let t0 be t1, go to Step 2.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

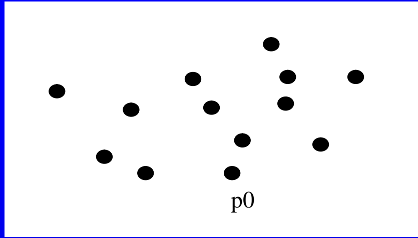
Gift-Wrapping Algorithm Time

- Finding the rightmost lowest point in Step 1 can be done in O(n) time. Whether a point is on the left side of a line, right side, or on the line can decided in O(1) time (see Exercise 3.32).
- Thus, it takes O(n) time to find a new point t1 in Step 2.
- Step 2 is repeated h times, where h is the size of the convex hull. Therefore, the algorithm takes O(hn) time. In the worst case, h is n.

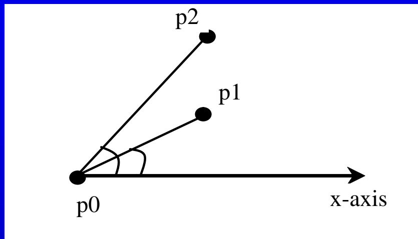


© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

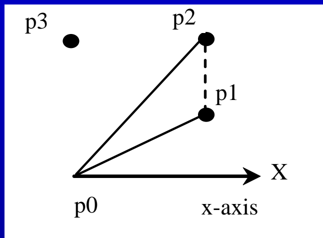
Graham's Algorithm

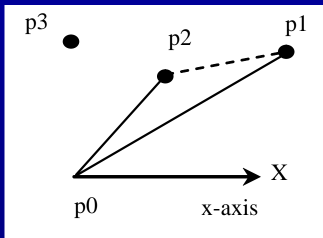


Given a set of points S, select the rightmost lowest point and name it p0 in the set S. As shown in Figure 16.7a, p0 is such a point.



Sort the points in S angularly along the x-axis with p0 as the center. If there is a tie and two points have the same angle, discard the one that is closest to p0. The points in S are now sorted as p0, p1, p2, ..., p_{n-1}.





The convex hull is discovered incrementally. Initially, p0, p1, and p2 form a convex hull. Consider p3. p3 is outside of the current convex hull since points are sorted in increasing order of their angles. If p3 is strictly on the left side of the line from p1 to p2, push p3 into H. Now p0, p1, p2, and p3 form a convex hull. If p3 is on the right side of the line from p1 to p2 (see Figure 9.7d), pop p2 out of H and push p3 into H. Now p0, p1, and p3 form a convex hull and p2 is inside of this convex hull.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Graham's Algorithm Time

O(nlogn)

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.

Practical Considerations

The big O notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient. As shown in the preceding example, both algorithms in Listings 5.6 and 16.2 have the same complexity, but the one in Listing 16.2 is obviously better practically.

© Copyright 2012 by Pearson Education, Inc. All Rights Reserved.