Algorithm Efficiency and Sorting

Measuring the Efficiency of Algorithms

- Analysis of algorithms
  - Provides tools for contrasting the efficiency of different methods of solution
- A comparison of algorithms
  - Should focus of significant differences in efficiency
  - Should not consider reductions in computing costs due to clever coding tricks

Measuring the Efficiency of Algorithms

- Three difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - What computer should you use?
  - What data should the programs use?
- Algorithm analysis should be independent of
  - Specific implementations
  - Computers
  - Data

The Execution Time of Algorithms

- Counting an algorithm's operations is a way to access its efficiency
  - An algorithm’s execution time is related to the number of operations it requires
  - Examples
    - Traversal of a linked list
    - The Towers of Hanoi
    - Nested Loops

Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the problem size
- An algorithm's growth rate
  - Enables the comparison of one algorithm with another
  - Examples

    Algorithm A requires time proportional to n^2

    Algorithm B requires time proportional to n
- Algorithm efficiency is typically a concern for large problems only

Algorithm Growth Rates

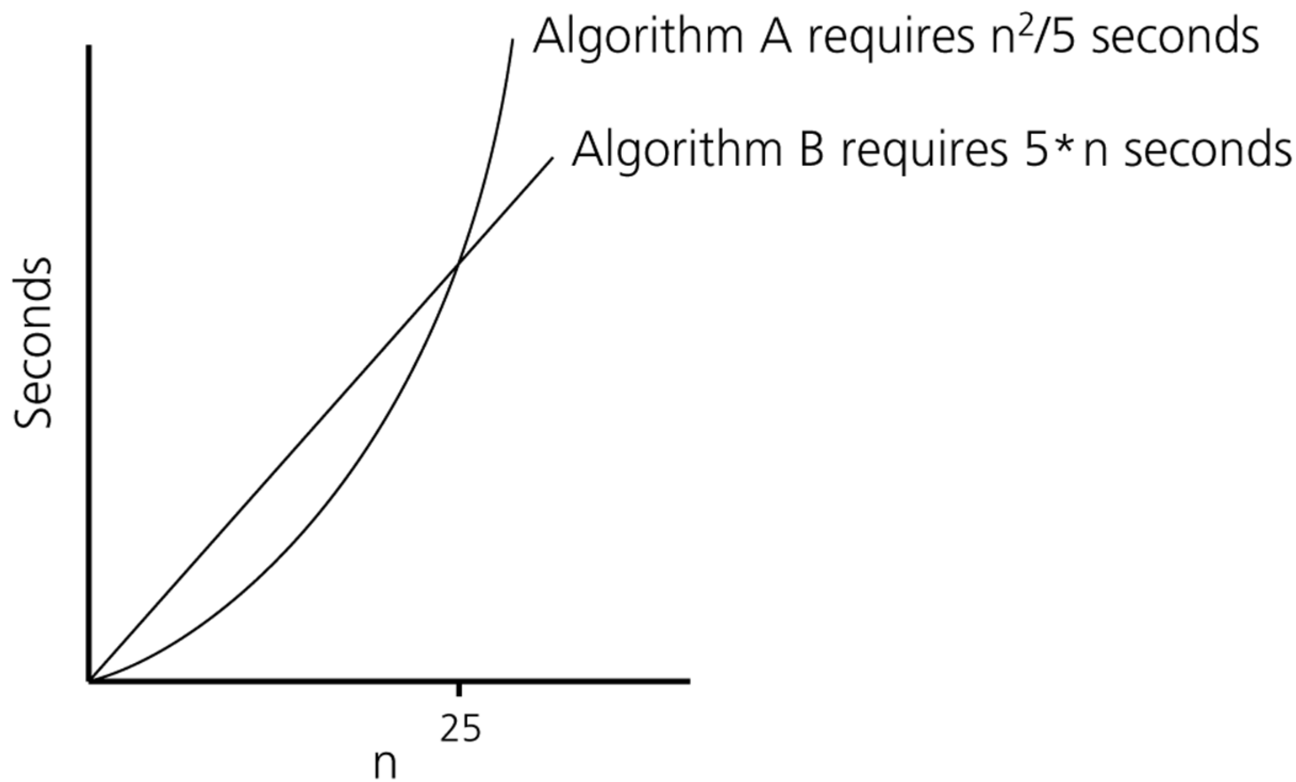

Figure 10-1

Time requirements as a function of the problem size *n*

Order-of-Magnitude Analysis and Big O Notation

- Definition of the order of an algorithm

  Algorithm A is order f(n) – denoted O(f(n)) – if constants k and n_0 exist such that A requires no more than k * f(n) time units to solve a problem of size $n \geq n_0$
- Growth-rate function
  - A mathematical function used to specify an algorithm's order in terms of the size of the problem
- Big O notation
  - A notation that uses the capital letter O to specify an algorithm's order
  - Example: O(f(n))

Order-of-Magnitude Analysis and Big O Notation

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Figure 10-3a

A comparison of growth-rate functions: a) in tabular form

Order-of-Magnitude Analysis and Big O Notation

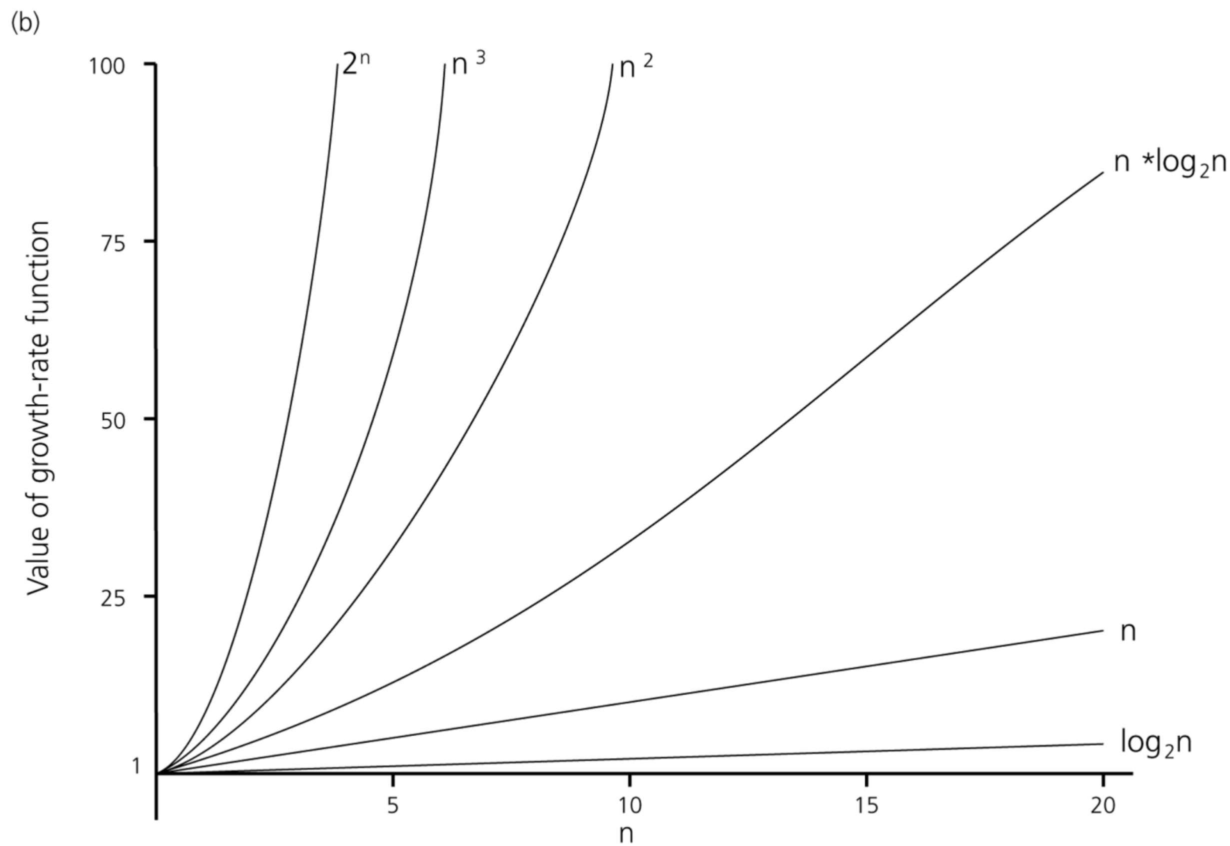

Figure 10-3b

A comparison of growth-rate functions: b) in graphical form

Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions

  $O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

- Properties of growth-rate functions
  - You can ignore low-order terms
  - You can ignore a multiplicative constant in the high-order term
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Order-of-Magnitude Analysis and Big O Notation

- Worst-case and average-case analyses
  - An algorithm can require different times to solve different problems of the same size
    - Worst-case analysis
      - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
    - Average-case analysis
      - A determination of the average amount of time that an algorithm requires to solve problems of size n

Keeping Your Perspective

- Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency
- When choosing an ADT's implementation, consider how frequently particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient

Keeping Your Perspective

- If the problem size is always small, you can probably ignore an algorithm's efficiency
- Weigh the trade-offs between an algorithm's time requirements and its memory requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on large problems

The Efficiency of Searching Algorithms

- Sequential search
  - Strategy
    - Look at each item in the data collection in turn, beginning with the first one
    - Stop when
      - You find the desired item
      - You reach the end of the data collection

The Efficiency of Searching Algorithms

- Sequential search
  - Efficiency
    - Worst case: O(n)
    - Average case: O(n)
    - Best case: O(1)

The Efficiency of Searching Algorithms

- Binary search
  - Strategy
    - To search a sorted array for a particular item
      - Repeatedly divide the array in half
      - Determine which half the item must be in, if it is indeed present, and discard the other half
  - Efficiency
    - Worst case: $O(\log_2 n)$
- For large arrays, the binary search has an enormous advantage over a sequential search

Sorting Algorithms and Their Efficiency

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer’s main memory
  - An external sort
    - The collection of data will not fit in the computer’s main memory all at once but must reside in secondary storage

Sorting Algorithms and Their Efficiency

- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects
- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records

Selection Sort

- Selection sort
  - Strategy
    - Select the largest item and put it in its correct place
    - Select the next largest item and put it in its correct place, etc.

Shaded elements are selected;
boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

Figure 10-4

A selection sort of an array of five integers

Selection Sort

- Analysis
  - Selection sort is $O(n^2)$
- Advantage of selection sort
  - It does not depend on the initial arrangement of the data
- Disadvantage of selection sort
  - It is only appropriate for small n

Selection Code

```
// This code will compile with warnings about unchecked exceptions

    public class SortsClass {

    public static void selectionSort(Comparable[] theArray,
    int n) {
   // ---------------------------------------------------
   // Sorts the items in an array into ascending order.
   // Precondition: theArray is an array of n items.
   // Postcondition: theArray is sorted into
   // ascending order.
   // Calls: indexOfLargest.
   // ---------------------------------------------------
   // last = index of the last item in the subarray of
   // items yet to be sorted
   // largest = index of the largest item found
```

Selection Code

```
for (int last = n-1; last >= 1; last--) {
    // Invariant: theArray[last+1..n-1] is sorted
    // and > theArray[0..last]
    // select largest item in theArray[0..last]
      int largest = indexOfLargest(theArray, last+1);
    // swap largest item theArray[largest] with
    // theArray[last]
      Comparable temp = theArray[largest];
      theArray[largest] = theArray[last];
      theArray[last] = temp;
    } // end for
  } // end selectionSort
```

Selection Code

```
private static int indexOfLargest(Comparable[] theArray,
     int size) {
    // ---------------------------------------------------
    // Finds the largest item in an array.
    // Precondition: theArray is an array of size items;
    // size >= 1.
    // Postcondition: Returns the index of the largest
    // item in the array.
    // ---------------------------------------------------
      int indexSoFar = 0; // index of largest item found so far
    // Invariant: theArray[indexSoFar]>=theArray[0..currIndex-1]
      for (int currIndex = 1; currIndex < size; ++currIndex) {
        if (theArray[currIndex].compareTo(theArray[indexSoFar])>0) {
          indexSoFar = currIndex;
        } // end if
      } // end for
      return indexSoFar; // index of largest item
    } // end indexOfLargest
```

Performance Analysis

Bubble Sort

- Bubble sort
  - Strategy
    - Compare adjacent elements and exchange them if they are out of order
      - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
      - Repeating this process will eventually sort the array into ascending (or descending) order

Bubble Sort

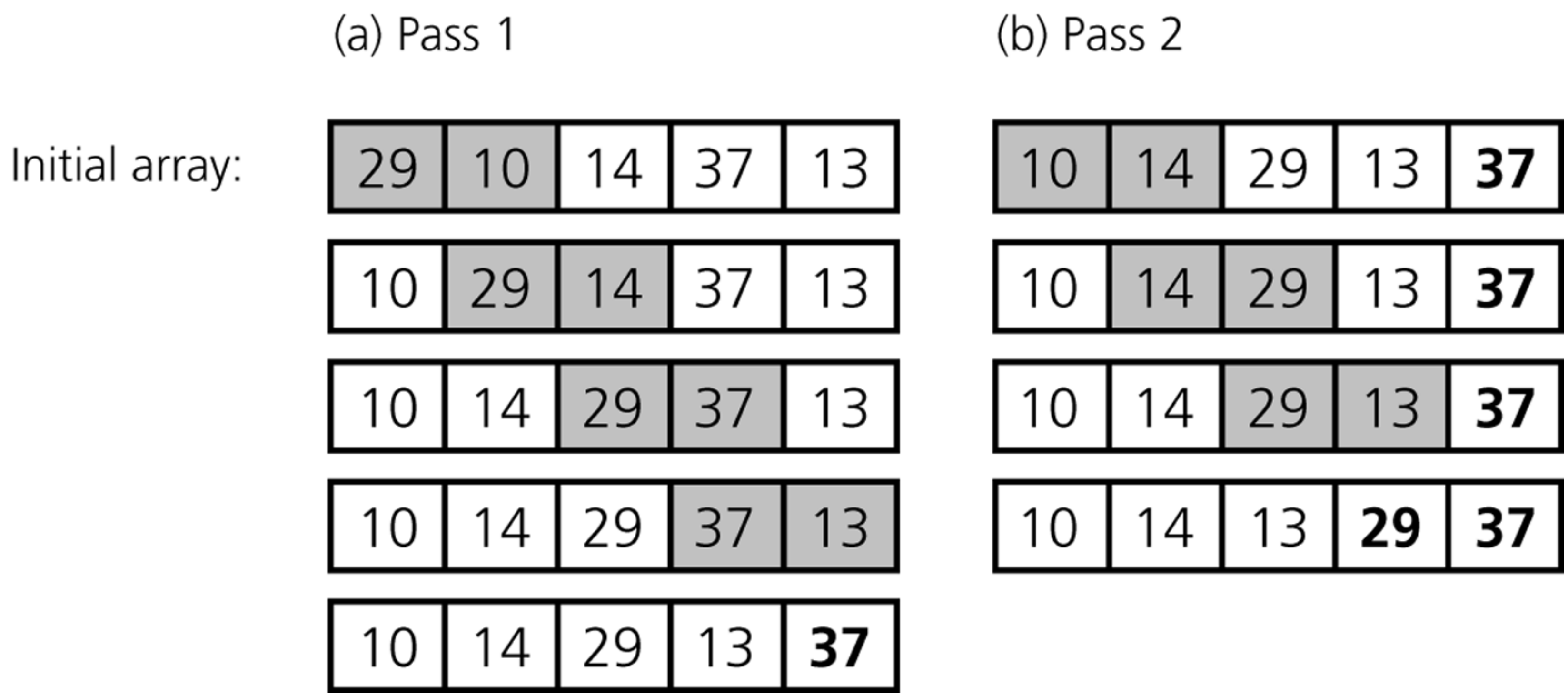

Figure 10-5

The first two passes of a bubble sort of an array of five integers: a) pass 1; b) pass 2

Bubble Sort Code

```
public static void bubbleSort(Comparable[] theArray, int n) {

  // ---------------------------------------------------
  // Sorts the items in an array into ascending order.
  // Precondition: theArray is an array of n items.
  // Postcondition: theArray is sorted into ascending
  // order.
  // ---------------------------------------------------
    boolean sorted = false; // false when swaps occur
    for (int pass = 1; (pass < n) && !sorted; ++pass) {
    // Invariant: theArray[n+1-pass..n-1] is sorted
    // and > theArray[0..n-pass]
      sorted = true; // assume sorted
      for (int index = 0; index < n-pass; ++index) {
      // Invariant: theArray[0..index-1] <= theArray[index]
        int nextIndex = index + 1;
        if (theArray[index].compareTo(theArray[nextIndex]) > 0) {
        // exchange items
          Comparable temp = theArray[index];
          theArray[index] = theArray[nextIndex];
          theArray[nextIndex] = temp;
          sorted = false; // signal exchange
        } // end if
      } // end for
    // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
    } // end for
  } // end bubbleSort
```

Bubble Sort Analysis

- Analysis
  - Worst case: $O(n^2)$
  - Best case: $O(n)$

Insertion Sort

- Insertion sort
  - Strategy
    - Partition the array into two regions: sorted and unsorted
    - Take each item from the unsorted region and insert it into its correct order in the sorted region

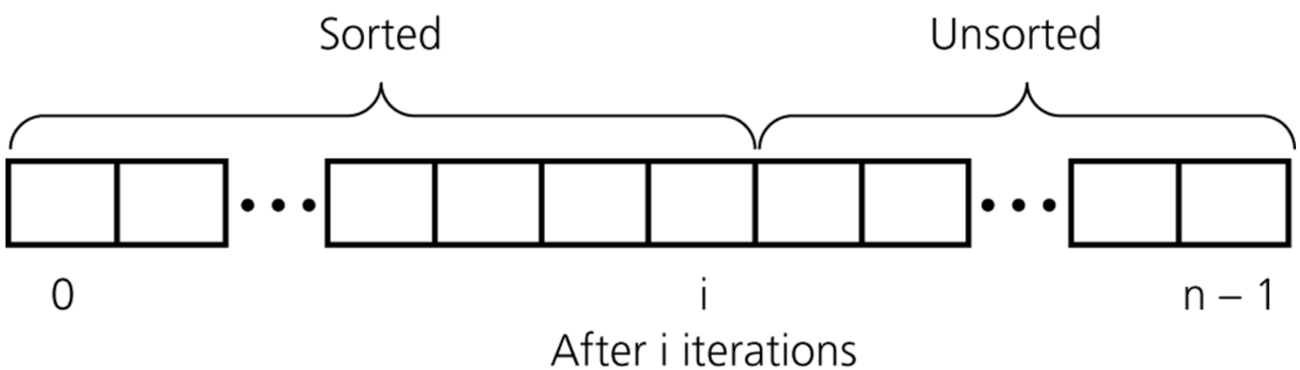

Figure 10-6

An insertion sort partitions the array into two regions

Insertion Sort

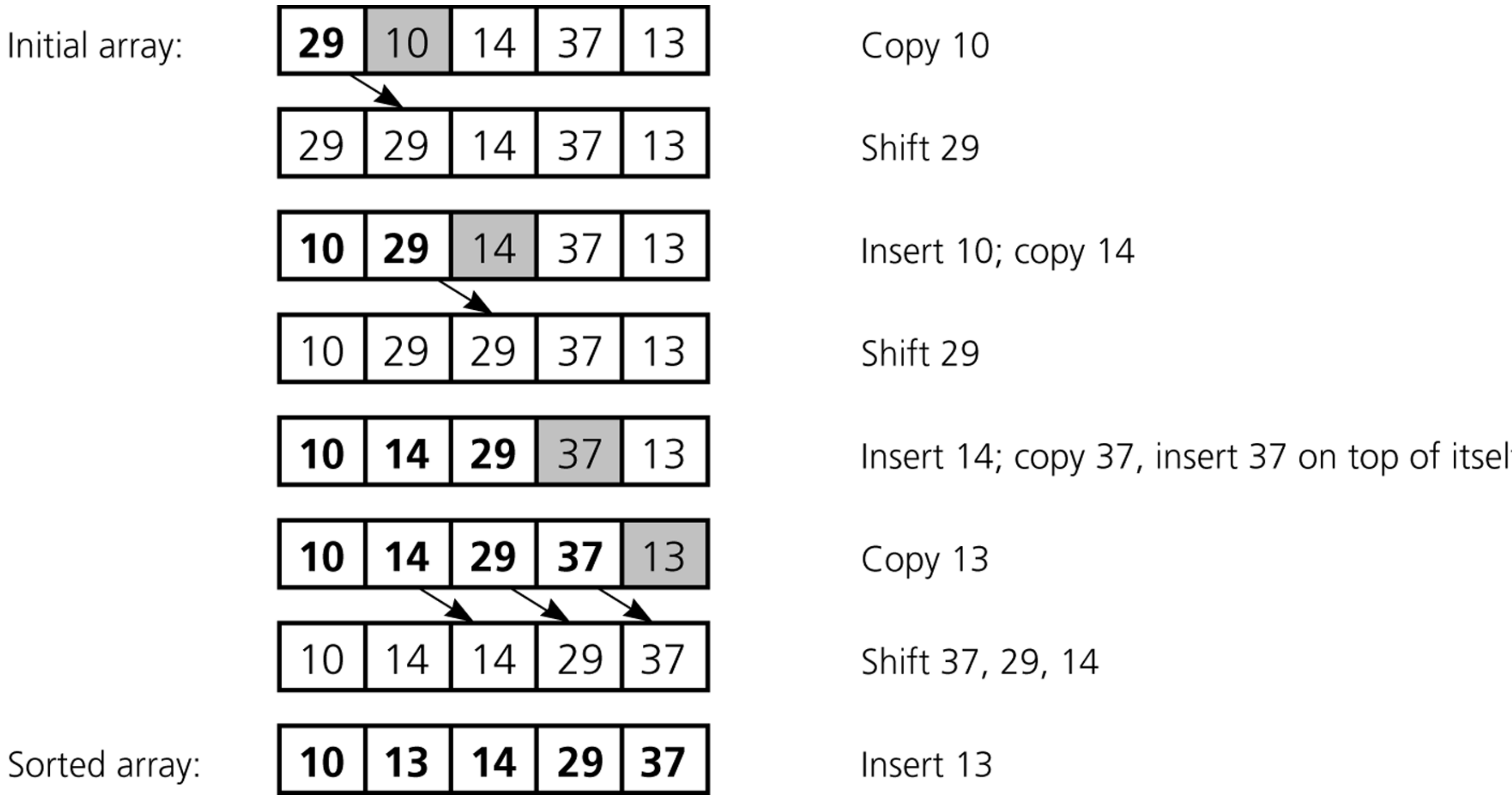

Figure 10-7

An insertion sort of an array of five integers.

Insertion Sort Code

```
public static void insertionSort(Comparable[] theArray,
      int n) {
    // -------------------------------------------------
    // Sorts the items in an array into ascending order.
    // Precondition: theArray is an array of n items.
    // Postcondition: theArray is sorted into ascending
    // order.
    // -------------------------------------------------
    // unsorted = first index of the unsorted region,
    // loc = index of insertion in the sorted region,
    // nextItem = next item in the unsorted region
    // initially, sorted region is theArray[0],
    // unsorted region is theArray[1..n-1];
      for (int unsorted = 1; unsorted < n; ++unsorted) {
      // Invariant: theArray[0..unsorted-1] is sorted
      // find the right position (loc) in
      // theArray[0..unsorted] for theArray[unsorted],
      // which is the first item in the unsorted
      // region; shift, if necessary, to make room
```

Insertion Sort Code

```
Comparable nextItem = theArray[unsorted];
        int loc = unsorted;
        while ((loc > 0) &&
        (theArray[loc-1].compareTo(nextItem) > 0)) {
        // shift theArray[loc-1] to the right
          theArray[loc] = theArray[loc-1];
          loc--;
        } // end while
      // insert nextItem into sorted region
        theArray[loc] = nextItem;
      } // end for
    } // end insertionSort
```

Insertion Sort

- Analysis
  - Worst case: $O(n^2)$
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is prohibitively inefficient

Mergesort

- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
- Mergesort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array items
  - Strategy
    - Divide an array into halves
    - Sort each half
    - Merge the sorted halves into one sorted array

Mergesort

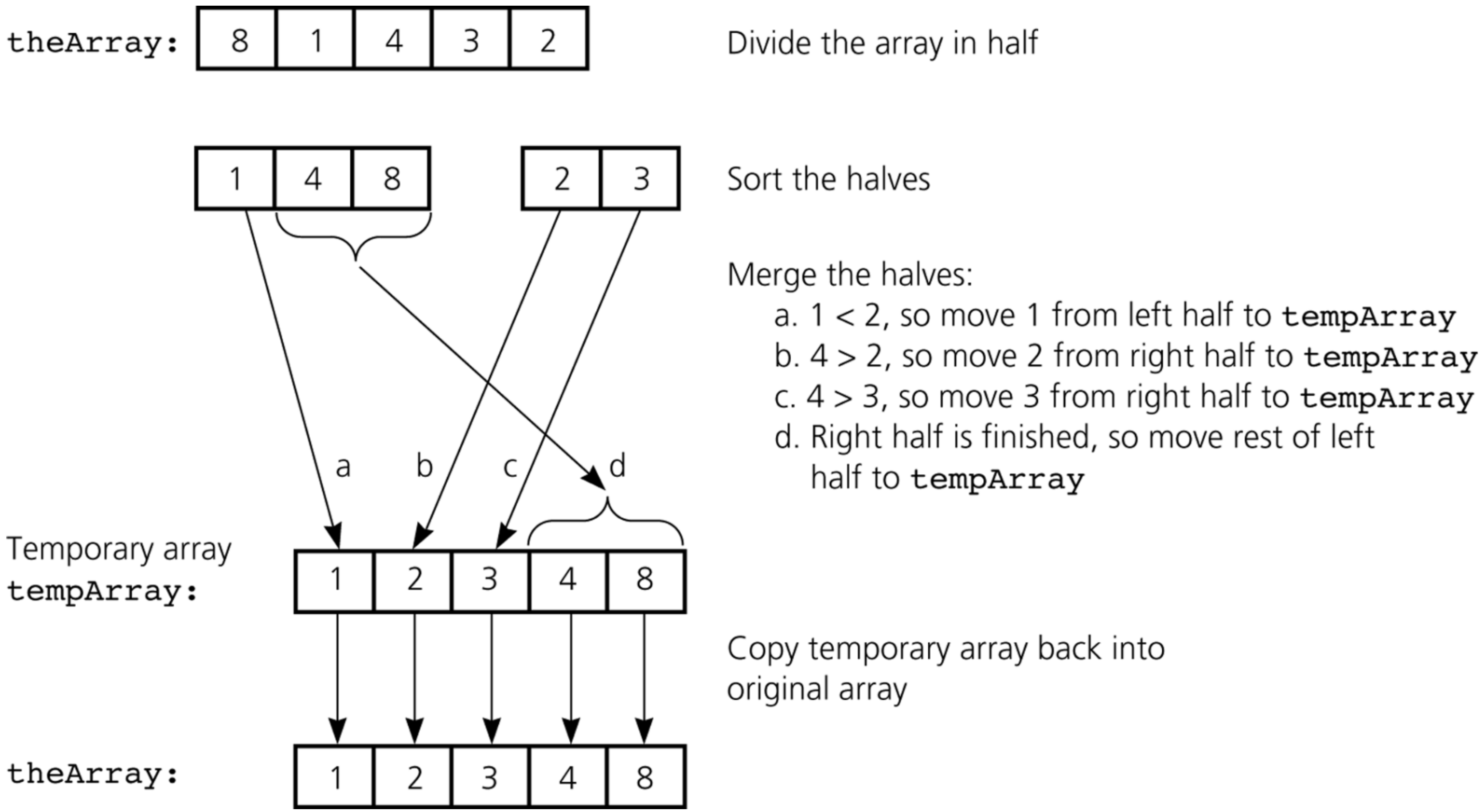

Figure 10-8

A mergesort with an auxiliary temporary array

Mergesort

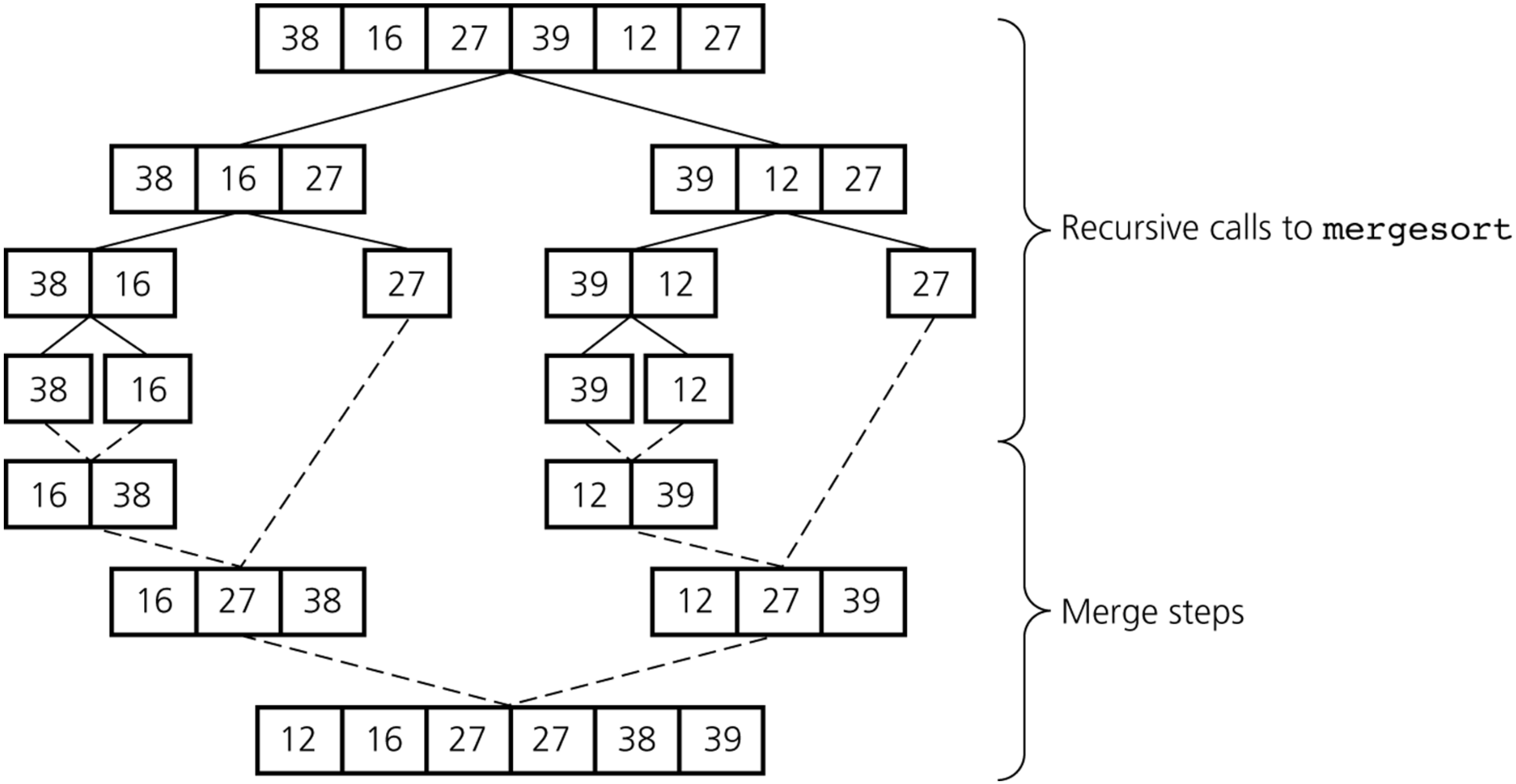

Figure 10-9

A mergesort of an array of six integers

Mergesort

- Analysis
  - Worst case: $O(n * \log_2 n)$
  - Average case: $O(n * \log_2 n)$
  - Advantage
    - It is an extremely efficient algorithm with respect to time
  - Drawback
    - It requires a second array as large as the original array

Mergesort

Click here to open the mergesort program

Quicksort

- Quicksort
  - A divide-and-conquer algorithm
  - Strategy
    - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

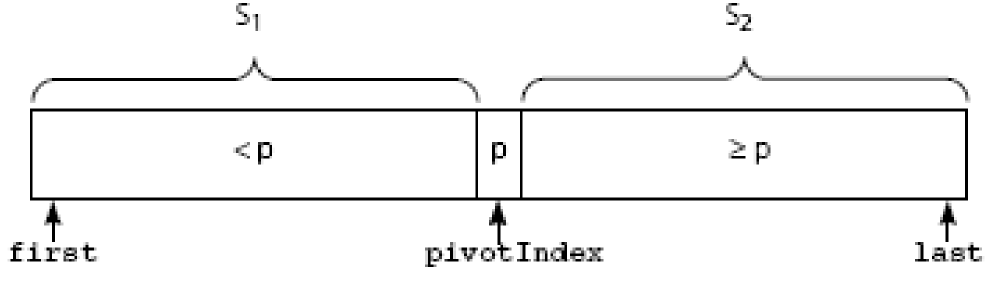

Figure 10-12

A partition about a pivot

Quicksort

- Using an invariant to develop a partition algorithm
  - Invariant for the partition algorithm

    The items in region S_1 are all less than the pivot, and those in S_2 are all greater than or equal to the pivot

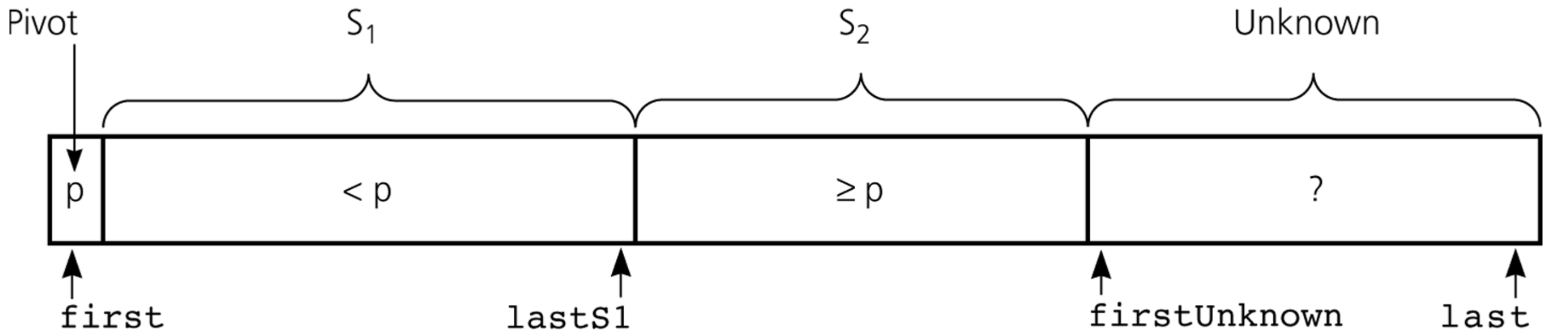

Figure 10-14

Invariant for the partition algorithm

Quicksort

- Analysis
  - Worst case
    - `quicksort` is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot

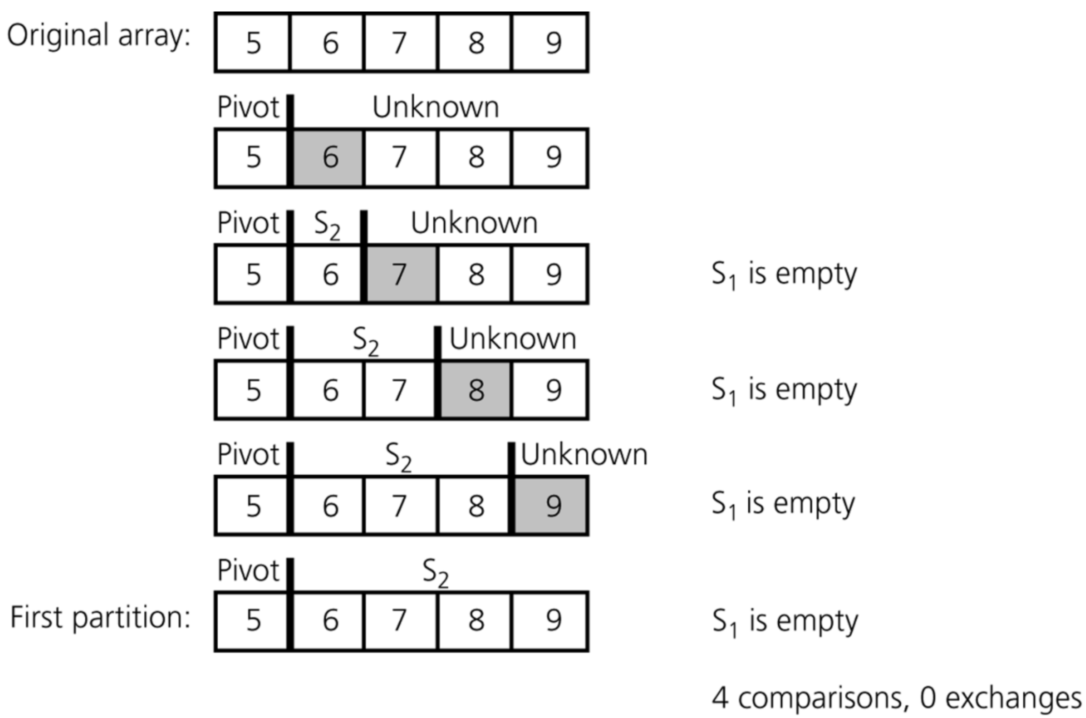

Figure 10-19

A worst-case partitioning with ***quicksort***

Quicksort

- Analysis
  - Average case
    - `quicksort` is O(n * $\log_2 n$) when S_1 and S_2 contain the same – or nearly the same – number of items arranged at random

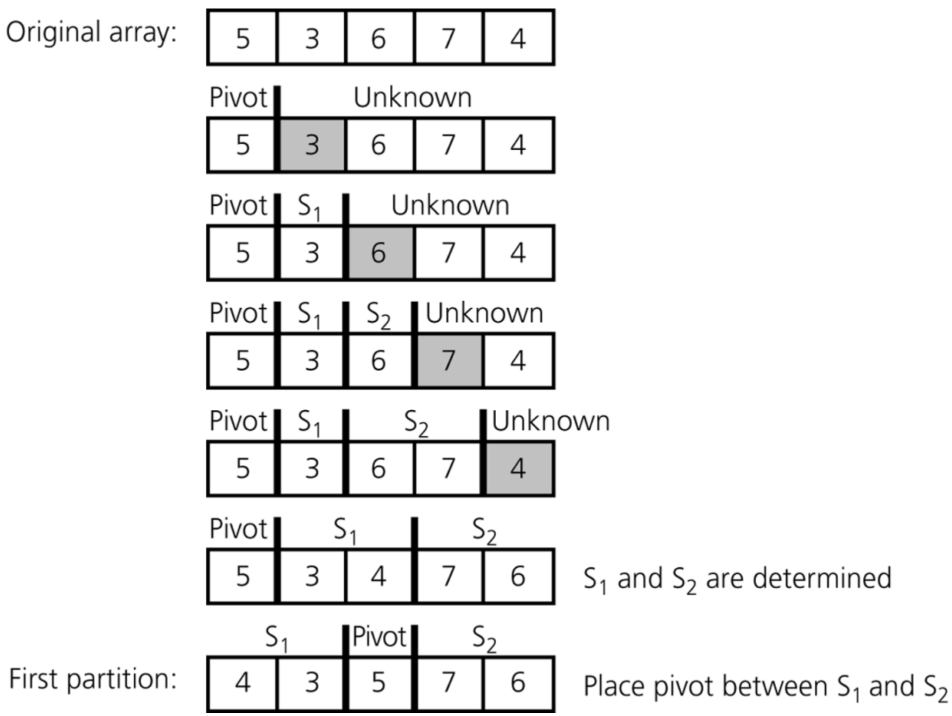

Figure 10-20

A average-case partitioning with ***quicksort***

Quicksort

- Analysis
  - `quicksort` is usually extremely fast in practice
  - Even if the worst case occurs, `quicksort`'s performance is acceptable for moderately large arrays
  - Click here to open the quicksort program

Radix Sort

- Radix sort
  - Treats each data element as a character string
  - Strategy
    - Repeatedly organize the data into groups according to the i^{th} character in each element
- Analysis
  - Radix sort is O(n)

Radix Sort

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

Figure 10-21

A radix sort of eight integers

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Figure 10-22

Approximate growth rates of time required for eight sorting algorithms

Summary

- Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function
- To compare the inherit efficiency of algorithms
  - Examine their growth-rate functions when the problems are large
  - Consider only significant differences in growth-rate functions

Summary

- Worst-case and average-case analyses
  - Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size
  - Average-case analysis considers the expected amount of work an algorithm requires on a problem of a given size
- Order-of-magnitude analysis can be used to choose an implementation for an abstract data type
- Selection sort, bubble sort, and insertion sort are all $O(n^2)$ algorithms
- Quicksort and mergesort are two very efficient sorting algorithms

Acknowledgement

All of the material for the slides were adapted from

Data Abstraction and Problem Solving with Java, 2/E

Frank Carrano, *University of Rhode Island*

Janet Prichard, *Bryant College*