Chapter 10

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 on 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 $n$

Algorithm A requires $n^2/5$ seconds
Algorithm B requires $5 \times n$ seconds
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 $k \times f(n)$ 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)

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\log_2 n$</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>n $\times \log_2 n$</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td>$10^7$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$10^2$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^8$</td>
<td>$10^{10}$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>$10^9$</td>
<td>$10^{12}$</td>
<td>$10^{15}$</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^3$</td>
<td>$10^{30}$</td>
<td>$10^{301}$</td>
<td>$10^{3,010}$</td>
<td>$10^{30,103}$</td>
<td>$10^{301,030}$</td>
</tr>
</tbody>
</table>

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.

Figure 10-4
A selection sort of an array of five integers

<table>
<thead>
<tr>
<th>Initial array:</th>
<th>29 10 14 37 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 1st swap:</td>
<td>29 10 14 13 37</td>
</tr>
<tr>
<td>After 2nd swap:</td>
<td>13 10 14 29 37</td>
</tr>
<tr>
<td>After 3rd swap:</td>
<td>13 10 14 29 37</td>
</tr>
<tr>
<td>After 4th swap:</td>
<td>10 13 14 29 37</td>
</tr>
</tbody>
</table>
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 $n$
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

• 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

Initial array: 29 10 14 37 13

- Copy 10
- Shift 29
- Insert 10; copy 14
- Shift 29
- Insert 14; copy 37, insert 37 on top of itself
- Copy 13
- Shift 37, 29, 14

Sorted array: 10 13 14 29 37

Insert 13

Figure 10-7
An insertion sort of an array of five integers.
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: $O(n \times \log_2 n)$
  – Average case: $O(n \times \log_2 n)$
  – Advantage
    • It is an extremely efficient algorithm with respect to time
  – Drawback
    • It requires a second array as large as the original array
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](image)

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
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

A radix sort of eight integers

<table>
<thead>
<tr>
<th>Original integers</th>
<th>Grouped by fourth digit</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150</td>
<td>(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)</td>
<td></td>
</tr>
<tr>
<td>1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004</td>
<td>(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)</td>
<td>Combined</td>
</tr>
<tr>
<td>0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283</td>
<td>(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)</td>
<td>Grouped by third digit</td>
</tr>
<tr>
<td>0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560</td>
<td>(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)</td>
<td>Combined</td>
</tr>
<tr>
<td>0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154</td>
<td>(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)</td>
<td>Grouped by first digit</td>
</tr>
<tr>
<td>0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154</td>
<td>(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)</td>
<td>Combined (sorted)</td>
</tr>
</tbody>
</table>
A Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$n \times \log n$</td>
<td>$n \times \log n$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$n^2$</td>
<td>$n \times \log n$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$n^2$</td>
<td>$n \times \log n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \times \log n$</td>
<td>$n \times \log n$</td>
</tr>
</tbody>
</table>

**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
Assignment

- Read through Chapter 10s