Chapter 10
Algorithm Efficiency and Sorting
CS102 Sections 51 and 52
Marc Smith and Jim Ten Eyck
Spring 2008

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

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

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 \( k \times f(n) \) 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 growth of some common functions**

\[
\begin{align*}
O(1) & < O(\log_2 n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)
\end{align*}
\]

**Properties of growth-rate functions**

- You can ignore low-order terms
- You can ignore a multiplicative constant in the high-order term
- \( O(f(n)) + O(g(n)) = O(f(n) + g(n)) \)

---

**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

• 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

The Efficiency of Searching Algorithms

• Sequential search
  – Efficiency
    • Worst case: $O(n)$
    • Average case: $O(n)$
    • Best case: $O(1)$

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: | 13 10 14 29 37 |
| After 2nd swap: | 13 10 14 29 37 |
| After 3rd swap: | 10 13 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 $n$

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

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

- 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

- 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
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](image1)
A partition about a pivot

- 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](image2)
A worst-case partitioning with quicksort

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

![Figure 10-20](image3)
A average-case partitioning with quicksort

Radix Sort

- Analysis
  - 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

<table>
<thead>
<tr>
<th>Original integers</th>
<th>Grouped by fourth digit</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0123, 2154, 0022, 0004, 0283, 1560, 1061, 2150)</td>
<td>(1560, 2150, 1061, 0283, 0123, 2032, 0004)</td>
<td>(1560, 0234, 1234, 0004, 2034)</td>
</tr>
<tr>
<td>(0004, 0232, 0123, 2034, 2150, 2154)</td>
<td>(1560, 1061, 0283)</td>
<td>(0004, 0123, 0234, 2032)</td>
</tr>
<tr>
<td>(0004, 0123, 0234, 2032, 2150, 2154)</td>
<td>(1560, 0234, 1234, 0004, 2034)</td>
<td>(0004, 0123, 0234, 2032, 2150, 2154)</td>
</tr>
</tbody>
</table>

Figure 10-21
A radix sort of eight integers

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>$n \cdot \log n$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \cdot \log n$</td>
<td>$n \cdot \log n$</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
- 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