Chapter 11

Searching, Sorting, and Complexity Analysis

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 11 describes searching, sorting, and complexity analysis. Students begin by learning the different ways in which one can measure the performance of an algorithm. Big-O notation is introduced next; emphasis is given to distinguishing the common orders of complexity and the algorithmic patterns that exhibit them. Finally, they study several search and sort algorithms and analyze their complexity.

Objectives

After completing this chapter, students will be able to:

- Measure the performance of an algorithm by obtaining running times and instruction counts with different data sets
- Analyze an algorithm's performance by determining its order of complexity, using big-O notation
- Distinguish the common orders of complexity and the algorithmic patterns that exhibit them
- Distinguish between the improvements obtained by tweaking an algorithm and reducing its order of complexity
- Write a simple linear search algorithm and a simple sort algorithm

Teaching Tips

Measuring the Efficiency of Algorithms

1. Explain that when choosing algorithms, we often have to settle for a space/time tradeoff.

Teaching Tip	For more information on the subject of Analysis of Algorithms, read: http://math.hws.edu/javanotes/c8/s6.html.

Measuring the Run Time of an Algorithm

1. Introduce the concept of *benchmark* and *profiling*.

2. Explain how to use `time.time()` to measure the run time of an algorithm.

3. Stress that this method exhibits two major problems: (1) running time varies with hardware, OS, and language, and (2), that it is impractical to determine the running time for some algorithms with very large data sets.

Counting Instructions

1. Stress that, when using this technique, we count the instructions in the high-level code in which the algorithm is written, not the instructions in the executable machine language program.

2. Explain why it is important to distinguish between instructions that execute the same number of times regardless of problem size and instructions whose execution count varies with problem size.

3. Use the `counting.py` and `countfib.py` scripts with Figures 11.3 and 11.4 to show how this technique works.

Measuring the Memory Used by an Algorithm

1. A complete analysis of the resources used by an algorithm includes the amount of memory required. Stress that, for this analysis, we focus on rates of potential growth.

Complexity Analysis

1. Explain that *complexity analysis* entails reading the algorithm and using pencil and paper to work out some simple algebra. Explain why this type of analysis is better than the ones introduced earlier.

Orders of Complexity

1. Use Figures 11.5 and 11.6 and Tables 11.1 and 11.2 to explain what *order of complexity* is and how to compare the efficiency between some common orders of complexity. Be sure to introduce the terms *linear*, *quadratic*, *constant*, *logarithmic*, *polynomial*, and *exponential* as they apply to complexity of algorithms.

Big-O Notation

1. Explain what *big-O notation* is. Introduce the terms *dominant* and *asymptotic analysis*.

Teaching Tip	For more information on big-O notation, visit: www.nist.gov/dads/HTML/bigOnotation.html.

The Role of the Constant of Proportionality

1. Explain that the *constant of proportionality* involves the terms and coefficients that are usually ignored during big-O analysis, and that, when these items are large, they may have an impact on the algorithm, particularly for small and medium-sized data sets.

Quick Quiz 1

1. What is benchmarking?
   Answer: One way to measure the time cost of an algorithm is to use the computer's clock to obtain an actual run time. This process, called benchmarking or profiling, starts by determining the time for several different data sets of the same size and then calculates the average time. Next, similar data are gathered for larger and larger data sets. After several such tests, enough data are available to predict how the algorithm will behave for a data set of any size.

2. True or False: Complexity analysis entails reading an algorithm and using pencil and paper to work out some simple algebra.
   Answer: True

3. True or False: An algorithm has linear performance if it requires the same number of operations for any problem size.
   Answer: False

4. The constant of _____ involves the terms and coefficients that are usually ignored during big-O analysis.
   Answer: proportionality

Search Algorithms

1. Briefly outline the different algorithms that can be used for searching and sorting lists.

Teaching Tip	For more information on search algorithms, read: http://www.softpanorama.org/Algorithms/searching.shtml.

Search for a Minimum

1. Using the `ourMin` function code provided in the book, explain why the search for a minimum in an unsorted list is O(*n*).

Linear Search of a List

1. Use the `linearSearch` code provided in the book to explain how a *sequential* or *linear search* works.

Best-Case, Worst-Case, and Average-Case Performance

1. Explain that the performance a linear search is O(*n*) in the worst and average case and O(1) in the best case. Demonstrate using an example.

Binary Search of a List

1. Explain why it is better to use a binary search when searching sorted data. Use the code provided in the book and Figure 11.7 to show how to implement a binary search function.

2. Stress that even though a binary search is more efficient than linear search, there is the additional cost of keeping the list in sorted order.

Comparing Data Items

1. Remind students that to allow algorithms to use comparison operators with a new class of objects, we need to overload the ==, >, and < operator.

2. Use the `SavingsAccount` example provided in the book to show students how to implement the `__eq__`, `__lt__`, and `__gt__` methods.

Quick Quiz 2

1. Python's _____ function returns the minimum or smallest item in a list.
   Answer: `min`

2. True or False: Python's `is` operator is implemented as a method named `__contains__` in the `list` class.
   Answer: False

3. True or False: The worst-case complexity of a linear search is O(*n*).
   Answer: True

4. When searching sorted data, it is better to use a(n) _____ search than a sequential search.
   Answer: binary

Sort Algorithms

1. Introduce the `swap` function that will be used in the algorithms presented in this section.

Teaching Tip	For more information on sorting algorithms, visit: http://www.softpanorama.org/Algorithms/sorting.shtml.

Selection Sort

1. Use Table 11.3 and the code provided in the book to explain how *selection sort* works.

Teaching Tip	You can find an online selection sort demo at: http://apcsteacher.com/reference/sortdemo/sort_demo_select.htm.

2. Explain that this algorithm is $O(n^2)$ in all cases.

3. Stress that for large data sets, the cost of swapping items might also be significant. This additional cost is linear in the worst and average cases.

Bubble Sort

1. Use Table 11.4 and the code provided in the book to explain how *bubble sort* works.

Teaching Tip	You can find online bubble sort demos at: http://web.engr.oregonstate.edu/~minoura/cs261/javaProgs/sort/BubbleSort.html http://coderaptors.com/?BubbleSort

2. Explain that this algorithm is $O(n^2)$. Note that the worst-case behavior for exchanges is greater than linear.

3. Briefly explain how the algorithm can be improved, but note that even with this improvement, the average case is still $O(n^2)$.

Insertion Sort

1. Use the bullet points included in the text, Table 11.5, and the code provided in the book to explain how *insertion sort* works.

Teaching Tip	You can find an online insertion sort demo at: http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html.

2. Explain that the worst- and average-case behavior of this algorithm is $O(n^2)$. Explain why it is linear in its best case.

Best-Case, Worst-Case, and Average-Case Performance Revisited

1. Use the examples provided in the text to demonstrate how the best-case, average-case, and worst-case performances for an algorithm may vary.

2. Note that there are algorithms whose best-case and average-case performances are similar but whose performance can degrade to a worst case. When choosing or developing an algorithm, it is important to be aware of these distinctions.

Teaching Tip	For more information on useful sorting demonstrations, visit: www.bluffton.edu/~nesterd/java/SortingDemo.html www.cs.rit.edu/~atk/Java/Sorting/sorting.html www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html

An Exponential Algorithm: Recursive Fibonacci

1. Use Figure 11.8 to explain why the recursive Fibonacci algorithm is inefficient. Stress that exponential algorithms are generally impractical to run with any but very small problem sizes.

2. Briefly explain how *memoization* can be used to improve the efficiency of recursive functions that are called repeatedly with the same arguments.

Converting Fibonacci to a Linear Algorithm

1. Use pseudocode and/or Python code to explain how to convert Fibonacci to a linear algorithm.

Quick Quiz 3

1. How does selection sort work?
   Answer: Perhaps the simplest strategy is to search the entire list for the position of the smallest item. If that position does not equal the first position, the algorithm swaps the items at those positions. It then returns to the second position and repeats this process, swapping the smallest item with the item at the second position, if necessary. When the algorithm reaches the last position in this overall process, the list is sorted. The algorithm is called selection sort because each pass through the main loop selects a single item to be moved.

2. True or False: The performance of bubble sort is O(*n*).
   Answer: False

3. The worst-case behavior of insertion sort is O(_____).
   Answer: n^2

4. True or False: Exponential algorithms are generally impractical to run with any but very small problem sizes.
   Answer: True

Case Study: An Algorithm Profiler

1. Guide students as they step through this section.

Teaching Tip	For more information on the use of profilers, visit: http://en.wikipedia.org/wiki/Performance_analysis#Use_of_profilers.

Request

1. Ask students to write a program to allow profiling of sort algorithms.

Analysis

1. Make sure students understand the role of the `Profiler` class (see Table 11.6) and the specific uses of the various methods included in that class.

Design

1. Describe the role of the two modules used in this case study: `profiler` and `algorithms`.

Implementation (Coding)

1. Guide students as they go through the code provided in this section.

Class Discussion Topics

1. Ask your students to talk about any other sort algorithms that they may be aware of. What is the complexity of these algorithms?

2. After finishing this chapter, ask your students if they feel comfortable doing a complexity analysis of an algorithm.

Additional Projects

1. Ask students to do some research on the quicksort algorithm. They should create a Python function that implements this algorithm and a script to test it.

2. Ask students to do some research on the heap sort algorithm. They should create a Python function that implements this algorithm and a script to test it.

Additional Resources

1. Big-O notation:
   www.nist.gov/dads/HTML/bigOnotation.html

2. Sorting algorithm:
   http://en.wikipedia.org/wiki/Sorting_algorithm

3. Sorting demonstrations:
   www.bluffton.edu/~nesterd/java/SortingDemo.html

4. Sorting algorithms:
   www.cs.rit.edu/~atk/Java/Sorting/sorting.html

5. Dictionary of algorithms and data structures:
   www.nist.gov/dads/

Key Terms

- **algorithm(s):** A finite sequence of instructions that, when applied to a problem, will solve it.
- **asymptotic analysis:** The process of approximating the performance of an algorithm by focusing on the largest degree term of a polynomial.
- **benchmarking:** The process of measuring the performance of an algorithm by timing it on a real computer. Also known as profiling
- **big-O notation:** A formal notation used to express the amount of work done by an algorithm or the amount of memory used by an algorithm.
- **binary search:** The process of examining a middle value of a sorted collection to see which half contains the value in question and halving until the value is located.
- **bubble sort:** A sorting algorithm that swaps consecutive elements that are out of order to bubble the elements to the top or bottom on each pass.
- **complexity analysis:** The process of deriving a formula that expresses the rate of growth of work or memory as a function of the size of the data or problem that it solves. See also big-O notation.
- **constant of proportionality:** The measure of the amount of work of an algorithm that never varies with the size of its data set.
- **counter:** A variable used to count the number of times some process is completed.
- **Fibonacci numbers:** A series of numbers generated by taking the sum of the previous two numbers in the series. The series begins with the numbers 0 and 1.
- **insertion sort:** A sorting algorithm that locates an insertion point and takes advantage of partial orderings in an array.
- **linear:** An increase of work or memory in direct proportion to the size of a problem.

- **logarithmic:** An increase of work in proportion to the number of times that the problem size can be divided by 2.
- **profiling:** See benchmarking.
- **quadratic:** An increase of work or memory in proportion to the square of the size of the problem.
- **recursive function:** A function that calls itself.
- **selection sort:** A sorting algorithm that sorts the components of a list in either ascending or descending order. This process puts the smallest or largest element in the top position and repeats the process on the remaining list components.
- **sequential search:** The process of searching a list by examining the first component and then examining successive components in the order in which they occur. Also referred to as a linear search.