Chapter 1

Algorithms: Efficiency, Analysis, and Order

Preview

- Overview
- Lecture Notes
  - Algorithms
  - The Importance of Developing Efficient Algorithms
  - Analysis of Algorithms
  - Order
  - Outline of This Book
- Quick Check
  - Analysis of Algorithms
  - Complexity Analysis
  - Applying the Theory
  - Analysis of Correctness
- Quick Check
  - Order
  - An Intuitive Introduction to Order
  - A Rigorous Introduction to Order
  - Using a Limit to Determine Order
- Quick Check
- Classroom Discussion
- Homework

- Keywords
- Solutions to Few Exercises
- Important Links

Overview

This chapter introduces algorithms and discusses important characteristics such as efficiency, analysis and order. An algorithm is a step-by-step procedure used to solve a problem. But making sure the developer is using the most efficient algorithm is very crucial no matter how fast computers become or how cheap memory gets. In order to determine how efficiently an algorithm solves a problem, we need to analyze the algorithm. Order helps group algorithms according to their eventual behavior.

Lecture Notes

Algorithms

A computer program is composed of individual modules, understandable by a computer, that solve specific tasks (such as sorting). The concentration here is not on the design of entire programs, but rather on the design of the individual modules that accomplish the specific tasks. These specific tasks are called **problems**. A problem is a question to which we seek an answer.

A problem may contain **variables** that are not assigned specific values in the statement of the problem. These variables are called parameters to the problem.

Because a problem contains **parameters**, it represents a class of problems, one for each assignment of values to the parameters. Each specific assignment of values to the parameters is called an **instance** of the problem.

To produce a computer program that can solve all instances of a problem, we must specify a general step-by-step procedure for producing the solution to each instance. This step-by-step procedure is called an **algorithm**. We say that the algorithm solves the problem.

Because C++ is a language with which we are familiar, we can use C++ like **pseudocode** to write algorithms. Refer to problem 1.2 for an example of the pseudocode. Examples 1.1 and 1.2 were stated for numbers. We want to search and sort items that come with any ordered set. Often, each item identifies a record. We call that record a **key**. We write searching and sorting algorithms using the defined data type **keytype** for the items.

The pseudocode is similar, but not identical to C++. A notable exception is the use of arrays. C++ allows arrays to be indexed only by the integers starting at 0. There are two other significant deviations from C++ regarding arrays in pseudocode. First, we allow variable-length two-dimensional arrays as parameters to routines. Second, we declare variable-length arrays.

Whenever we can demonstrate steps more succinctly and clearly using actual mathematical expressions of actual C++ instructions, we do so. Refer to page 6 for some examples of data types. In general, we avoid features peculiar to C++ so that the pseudocode is accessible to someone who knows only another high-level language.

The Importance of Developing Efficient Algorithms

Efficiency is an important consideration when working with algorithms. A solution is said to be efficient if it solves the problem within the required resource constraints.

Sequential Search versus Binary Search

The sequential search algorithm begins at the first position in the array and looks at each value in turn until the item is found.

The binary search algorithm first compares x with the middle item of the array. If they are equal, the algorithm is done. If x is smaller than the middle item, then x must be in the first half of the array (if it is present at all),and the algorithm repeats the searching procedure on the first half of the array.(That is, x is compared with the middle item of the first half of the array. If they are equal, the algorithm is done, etc.) If x is larger than the middle item of the array, the search is repeated on the second half of the array. This procedure is repeated until x is found or it is determined that x is not in the array.

To compare Sequential Search to Binary Search, Sequential Search does *n* comparisons to determine that *x* is not in the array of size *n*. If *x* is in the array, the number of comparisons is no greater than *n*. A Binary Search is an algorithm for locating the position of an element in a sorted list. The method reduces the number of elements that need to be examined by two each time. Binary Search appears to be much more efficient than Sequential Search.

The Fibonacci Sequence

By definition, the first two Fibonacci numbers are 0 and 1. Each remaining number is the sum of the previous two. A recurrence relation defines a function by means of an expression that includes one or more (smaller) instances of itself. An example of a recurrence is the Fibonacci sequence:

```
Fib(n) = Fib(n – 1) + Fib(n – 2) for n > 2;  Fib(1) = Fib(2) = 1
```

By doing the examples in the book, we can see that this algorithm is extremely inefficient.

Quick Check

1. Variables are called _____ to the problem.
   Answer: parameters
2. Each specific assignment of values to the parameters is called an instance of the problem (True or False).
   Answer: True
3. The step-by-step procedure to solve a problem is called a(n) _____.
   Answer: algorithm
4. A problem can be solved by only one algorithm (True or False).
   Answer: False
5. Sequential search appears to be much more efficient than binary search (True or False)
   Answer: False

Analysis of Algorithms

Algorithm analysis measures the efficiency of an algorithm as the input size becomes large. The critical resource for a program is most often its running time. Other factors should be taken into account, like the space required to run the program.

Complexity Analysis

When analyzing the efficiency of an algorithm in terms of time, we do not determine the actual number of CPU cycles because this depends on the computer on which the algorithm is run. We don't count every instruction executed because the number of instructions depends on the programming language used. We want a measure that is independent of the computer, the programming language, the programmer,

and all the complex details of the algorithm. In general, the running time of the algorithm increases with the size of the input. The total running time is proportional to how many times some basic operation is done. Therefore, we analyze an algorithm's efficiency be determining the number of times some basic operation is done as a function of the size of the input. The size of the input is called the **input size**.
In general, a time complexity analysis of an algorithm is the determination of how many times the basic operation is done for each value of the input size.

A **time complexity analysis** of an algorithm is the determination of how many times the basic operation is done for each value of the input size. A time complexity analysis determines how many times the basic operation is done for each value of the input size.

There is no hard and fast rule for choosing the basic operation. It is largely a matter of judgment and experience.

T(n) is defined as the number of times the algorithm does the basic operation for an instance of size n. T(n) is called the every-case time complexity of the algorithm, and the determination of T(n) is called an every-case time complexity analysis .
W(n) is defined as the maximum number of times the algorithm will ever do its basic operation for an input size of n. So W(n) is called the worst case time complexity of the algorithm, and the determination of W(n) is called a worst-case time complexity analysis.
A(n) is called the average-case time complexity of the algorithm, and the determination of A(n) is called an average-case time complexity analysis.
B(n) is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n .So B(n) is called the best-case time complexity of the algorithm, and the determination of B(n) is called a best-case time complexity analysis.

Memory complexity is an analysis of how efficient the algorithm is in terms of memory. A **complexity function** can be any function that maps the positive integers to the nonnegative reals.

Applying the Theory

When applying the theory of algorithm analysis, three important factors should be taken into account: the time it takes to execute the basic operation, the overhead instructions, and the control instructions on the actual computer on which the algorithm is implemented.

Overhead instructions mean instructions such as initialization instructions before a loop. The number of times these instructions execute does not increase with input size.

Control instructions mean instructions such as incrementing an index to control a loop. The number of times these instructions execute increases with input size.

The basic operation, overhead instructions, and control instructions are all properties of an algorithm and the implementation of the algorithm.

Analysis of Correctness

Analysis of an algorithm means an efficiency analysis in terms of either time or memory.
Other types of analysis include analyzing the correctness of an algorithm by developing a proof that the algorithm actually does what it is supposed to do.

Quick Check

1. Algorithm analysis measures the efficiency of an algorithm as the _____ becomes large.
   Answer: input size
2. _____ is defined as the number of times the algorithm does the basic operation for an instance of size n.

Answer: T(n)

3. W(n) is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n (True or False).
   Answer: False
4. A(n) is called the average-case time complexity of the algorithm (True or False).
   Answer: True
5. Analyzing the _____ of an algorithm is developing a proof that the algorithm actually does what it is supposed to do.
   Answer: correctness

Order

Algorithms with time complexities such as n and 100n are called linear-time algorithms because their time complexities are linear in the input size n, whereas algorithms with time complexities such as n^2 and 0.01n^2 are called quadratic-time algorithms because their time complexities are quadratic in the input size n. Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.

An Intuitive Introduction to Order

Functions such as 5n^2 and 5n^2 +100 are called **pure quadratic functions** because they contain no linear term, whereas a function such as 0.1n^2 + n +100 is called a **complete quadratic function** because it contains a linear term.

We should always be able to throw away low-order terms when classifying complexity functions. For example, it seems that we should be able to classify 0.1n^3 +10n^2 + 5n + 25 with pure cubic functions. When an algorithm's time complexity is in θ (n^2), the algorithm is called a **quadratic-time algorithm** or a θ(n^2) algorithm.

Examples of complexity categories:

Θ(lg(n)), θ(n), θ(n lg(n)), θ(n^2), θ(n^3), θ(2^n)

There is more information in knowing a time complexity than in simply knowing its order. If it takes the same amount of time to process basic operations and execute the overhead instructions in both algorithms, then the quadratic-time algorithm is more efficient in instances smaller than 10,000. If the application never requires instances larger than this, the quadratic-time algorithm should be implemented. When it is difficult to determine time complexities exactly, we have to be content with determining only the order.

A Rigorous Introduction to Order

For a given complexity function f (n), O(f(n)) is the set of complexity functions g(n) for which there exists some positive real constant c and some nonnegative integer N such that for all n ≥ N,

$g(n) \leq c \times f(n)$

If g(n) ϵ O(f(n)), we say that g(n) is big O of f(n).

"Big O" describes the **asymptotic behavior** of a function because it is concerned only with eventual behavior. We say that "big O" puts an asymptotic upper bound on a function.

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like "big O", this is a measure of the algorithm's growth rate, and it works for any resource, but we most often measure the least amount of time required. The lower bound of an algorithm is denoted by the symbol Ω. The following is the definition for Ω:

For a given complexity function f(n), Ω(f (n)) is the set of complexity functions g (n) for which there exists some positive real constant c and some nonnegative integer N such that, for all $n \geq N$,

$g(n) \geq c \times f(n)$.

If g(n) ϵ Ω(f(n)), we say that g(n) is omega of f(n).

For a given complexity function f(n),

$\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

This means that θ(f(n)) is the set of complexity functions g (n) for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,

$c \times f(n) \leq g(n) \leq d \times f(n)$.

If g(n) ϵ θ(f(n)), we say that g (n) is order of f(n).

Proof by contradiction: assume something is true, and then do manipulations that lead to a result that is not true.

For a given complexity function f(n), o(f(n)) is the set of all complexity functions g (n) satisfying the following: For every positive real constant c there exists a nonnegative integer N such that, for all $n \geq N$, $g(n) \leq c \times f(n)$.

If g(n) ϵ o(f(n)), we say that g (n) is small o of f(n).

Using a Limit to Determine Order

Knowledge of this material is not required elsewhere in the text.

Order can sometimes be determined using a limit.

Lim (g(n) / f(n)) as n -> ∞ =	c	implies g(n) ϵ θ(f(n)) if c > 0
	0	implies g(n) ϵ o(f(n))
	∞	implies f(n) ϵ o(g(n))

<u>Quick Check</u>

1. Algorithms with time complexities such as n and 100n are called _____ algorithms.
   Answer: linear-time
2. Algorithms with time complexities such as n^2 are called quadratic-time algorithms (True or False).
   Answer: True
3. Any quadratic-time algorithm is eventually more efficient than any linear-time algorithm (True or False).
   Answer: False
4. Functions such as 5n^2 and 5n^2 +100 are called _____ functions.
   Answer: pure quadratic
5. _____assumes something is true, and then does manipulations that lead to a result that is not true.
   Answer: Proof by contradiction

Classroom Discussion

- Does every problem have an algorithm? Explain
- Give some examples of recurrence relations (other than Fibonacci).
- Using the definitions of “big O” and Ω, find the upper and lower bounds for the following expressions: c*n^3 + d and c*n*log(n) + d*n.

Homework

Assign Exercises 14, 19, 26, and 33.

Keywords

- **Algorithm –** applying a technique to a problem that results in a step-by-step procedure for solving the problem.
- **Asymptotic behavior** – limiting behavior
- **Asymptotic upper bound** – the upper limit of the eventual behavior
- **Average-case time complexity (A(n))** – average of the number of times the algorithm does the basic operation for an input size of n.
- **Average-case time complexity analysis –** determination of A(n).
- **Best-case time complexity (B(n))** – minimum number of times the algorithm will ever do its basic operation for an input size of n.
- **Best-case time complexity analysis** – determination of B(n).
- **Complexity categories** – such as θ(lg(n)), θ(n), θ(n lg(n))
- **Complexity function –** a function that maps the positive integers to the nonnegative reals.
- **Every-case time complexity (T(n))** – number of times the algorithm does the basic operation for an instance of size n.
- **Every-case time complexity analysis** – determination of T(n)
- **Instance–** (of a problem) – specific assignment of values to the parameters of the problem.
- **Input size –** the size of the input.
- **Key –** something that uniquely identifies a record.
- **List** – collection of items arranged in a particular sequence.
- **Memory complexity –** analysis of how efficient the algorithm is in terms of memory.
- **Overhead instructions–** instructions such as initialization instructions before a loop.
- **Parameters** – variables in a problem that are not assigned specific values in the statement of the problem.
- **Problem** – a question to which we seek an answer.
- **Solution** (to an instance of a problem) – answer to the question asked by the problem in that instance.
- **Time complexity analysis –** determination of how many times the basic operation is done for each value of the input size.
- **Variable** – something that is not assigned a specific value.
- **Worst-case time complexity(W(n))** – maximum number of times the algorithm will ever do its basic operation for an input size of n.
- **Worst-case time complexity analysis –** determination of W(n).

Important Links

- http://www.nist.gov/dads/ (Dictionary of Algorithms and Data Structures)
- http://www.sciencedirect.com/science/journal/01966774/ (Journal of Algorithms)