Chapter 1: The Role of Algorithms in Computing

# Algorithms:

Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the outputThe statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship. For example, we might need to sort a sequence of numbers into nondecreasing order.

Sorting is by no means the only computational problem for which algorithms have been developed. Practical applications of algorithms are ubiquitous.

# Data Structures:

A data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

Parallelism:

For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, however: because power density increases super linearly with clock speed, chips run the risk of melting once their clock speeds become high enough. In order to perform more computations per second, therefore, chips are being designed to contain not just one but several processing “cores.” We can liken these multicore computers to several sequential computers on a single chip; in other words, they are a type of “parallel computer.” In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind.

# Algorithms as a technology:

If computers were infinitely fast, any correct method for solving a problem would do. Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory.

# Efficiency:

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

# Algorithms and other technologies:

Algorithms are truly important on contemporary computers in light of other advanced technologies, such as advanced computer architectures and fabrication technologies, easy-to-use, intuitive, graphical user interfaces (GUIs), object-oriented systems, integrated Web technologies, and fast networking, both wired and wireless.

Chapter 2: Getting Started

# Insertion sort:

Insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

# Pseudocode:

1. for j = 2 to A.length

2 key = A[j]

3 // Insert A[j] into the sorted sequence A[1 .. j - 1]

4 i = j - 1

5 while i > 0 and A[i] > key

6 A[i + 1] = A[i]

7 i = i - 1

8 A[i + 1] = key

# Analyzing algorithms:

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

# Worst-case and average-case analysis:

The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. The “average case” is often roughly as bad as the worst case.

# Order of growth:

One algorithm becomes more efficient than another if its worst case running time has a lower order of growth. Due to constant factors and lower order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth.

# Designing algorithms:

Designing the right algorithm for a given application is a difficult job. It requires a major creative act, taking a problem and pulling a solution out of the ether. This is much more difficult than taking someone else's idea and modifying it or tweaking it to make it a little better. The space of choices a person can make in algorithm design is enormous, enough to leave him plenty of freedom to hang himself.

Chapter 3: Growth of Functions

# Asymptotic notation:

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N = {0, 1, 2, …}. Such notations are convenient for describing the worst-case running-time function T(n), which usually is defined only on integer input sizes.

Asymptotic notation, functions, and running times:

Asymptotic notation can be primarily used to describe the running times of algorithms. It actually applies to functions, however.

Θ-notation:

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic tight bound, we use Θ-notation.

O-notation:

The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation. We use O-notation to give an upper bound on a function, to within a constant factor.

Ω-notation:

Just as O-notation provides an asymptotic upper bound on a function, Ω-notation provides an asymptotic lower bound.

o-notation:

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. We use o-notation to denote an upper bound that is not asymptotically tight.

ω-notation:

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight.

# Standard notations and common functions:

Monocity:

A monotonic function (or monotone function) is a function  between ordered sets that preserves or reverses the given order. This concept first arose in calculus, and was later generalized to the more abstract setting of order theory.

Floors and ceilings:

The floor function is the function that takes as input a real number{\displaystyle x} x and gives as output the greatest integer less than or equal to x{\displaystyle x}, denoted floor(x)=└x┘{\displaystyle \operatorname {floor} (x)=\lfloor x\rfloor }. Similarly, the ceiling function maps x{\displaystyle x} to the least integer greater than or equal to x{\displaystyle x}, denoted ceil(x)=┌x┐{\displaystyle \operatorname {ceil} (x)=\lceil x\rceil }.

Modular arithmetic:

Modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" when reaching a certain value, called the modulus.

Polynomials:

A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables.

Exponentials:

An exponential function is a function of the form where b is a positive real number, and in which the argument x occurs as an exponent.

Logarithms:

The logarithm is the inverse function to exponentiation. That means the logarithm of a given number x is the exponent to which another fixed number, the base b, must be raised, to produce that number x.

Factorials:

The factorial of a positive integer n, denoted by n!, is the product of all positive integers less than or equal to n.

Functional iteration:

An iterated function is a function X → X which is obtained by composing another function f: X → X with itself a certain number of times. The process of repeatedly applying the same function is called iteration.

The iterated logarithm function:

The iterated logarithm of n{\displaystyle n}, written log\*n{\displaystyle n}, is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1{\displaystyle 1}.

Fibonacci numbers:

The Fibonacci numbers, commonly denoted *Fn*, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.