Chapter 1: The Role of Algorithms in Computing

**Algorithms** : An algorithm is any well-defined computational procedure thet takes some value, or set of values, as input and produce some value, or set of values, as output.The algorithm describes a specific computational procedure for achieving that input/output relationship.Sorting is by no means the only computational problem for whice algorithms have beer developed. Not every problem solved by algorithms has an easily identiﬁed set of candidate solutions.

**Data structures** : A data structure is a way to store and organize data in order to facilitate access and modiﬁcations.

**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 superlinearly 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 toseveralsequential computers onasingle chip; inotherwords, theyare atypeof“parallel computer.” Inordertoelicitthebestperformance frommulticore computers, we need to design algorithms with parallelism in mind

Chapter 2: Getting Started

**Insertion sort** : A good algorithm for sorting a small number of elements. It works the way you might sort a hand of playing cards. Start with an empty left hand and the cards face down on the table. 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, 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.

INSERTION-SORT(A)

for j ← 2 to n

do key ← A[j]

Insert A[j] into the sorted sequence A[1 . . j − 1]

i ← j − 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i – 1

A[i + 1] ← key

Initialization: It is true prior to the ﬁrst iteration of the loop.

Maintenance: If it is true before an iteration of the loop, I tremains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

We start by presenting the INSERTION-SORT procedure with the time cost of each statement and the number of times each statement is executed. For each j = 2,3…n, where n=A.length, we let tj denote the number of times the while loop test in line 5 is executed for that value of j. When a for or while loop exits in the usual way, the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

The running time of the algorithm is the sum of running times for each statement executed.

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. The worst-case running time of an algorithm gives us an upper bound on the running time for any input.

Divide the problem into a number of subproblems that are smaller instances of the same problem. Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner. Combine the solutions to the subproblems into the solution for the original problem. The merge sort algorithm closely follows the divide-and-conquer paradigm.

Merge sort algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.

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 oforder 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.