1 The Role of Algorithms in Computing

Algorithms:

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 output. We can also view an algorithm as a tool for solving a well-speciﬁed computational problem. The 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.

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:

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.”

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:

We should consider algorithms, like computer hardware, as a technology. Total system performance depends on choosing efﬁcient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well. You might wonder whether algorithms are truly that 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

2.1 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, as illustrated in Figure 2.1. 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

1. for j = 2 to A.length

2. key = A[j]

3. i = j-1

4. while i > 0 and key < A[i] :

5. A[i+1] = A[i]

6. i =i- 1

7. A[i+1] = key

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. We need not make some educated guess about the running time and hope that it never gets much worse.

The “average case” is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[1…….j-1] to insert element A[ j ] ? On average, half the elements in A[1…..j-1] are less than A[ j ] and half the elements are greater. On average, therefore, we check half of the subarray A[1…..j-1] ,and so tj is about j/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Order of growth:

We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an2), since the lower-order terms are relatively insigniﬁcant for large values of n. We also ignore the leading term’s constant coefﬁcient, since constant factors are less signiﬁcant than the rate of growth in determining computational efﬁciency for large inputs.

2.3.1

The divide-and-conquer approach:

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves three steps at each level of the recursion:

**Divide** the problem into a number of sub problems that are smaller instances of the same problem.

**Conquer** the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straight forward manner.

**Combine** the solutions to the sub problems into the solution for the original problem.

2.3.2

Analyzing divide-and-conquer algorithms:

When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Chapter: 3 Growth of Functions

3.1

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. We sometimes ﬁnd it convenient, however, to abuse asymptotic notation in a variety of ways.

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.

3.2 Standard notations and common functions:

Monocity:

A  monotonic function (or monotone function) is 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 co efficients, 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.