Chapter 1: The Role of Algorithms in Computing

1.Algorithms:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

* **Search** − Algorithm to search an item in a data structure.
* **Sort** − Algorithm to sort items in a certain order.
* **Insert** − Algorithm to insert item in a data structure.
* **Update** − Algorithm to update an existing item in a data structure.
* **Delete** − Algorithm to delete an existing item from a data structure.

Algorithm is a systematic way of doing things, and a problem is anything that need physical or mental effort to solve.

If you are taking about computer problems and computer algorithms specifically, then problems can be like:

\*Find the shortest path between two nodes → can be solves with Dijkstra algorithm.

\*Find the first N prime numbers → Can be solved by Sieve of Eratosthenes.

\*Sorting N numbers in specific way → can be solved using merge sort.

\*Find a good block fit for a task, among many free blocks in memory management for OS part → first/best fit algorithm.

\*convert X amount of money to the minimum number of coins/papers → coin change dp algorithm.

**Data Structure**:

A data structure is a particular way of organizing data in a computer so that it can be used effectively. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

Hard problems:

Our usual measure of efﬁciency is speed, i.e., how long an algorithm takes to produce its result. There are some problems, however, for which no efﬁcient solution is known. Chapter 34 studies an interesting subset of these problems, which are known as NP-complete.

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.

2. Algorithms as a technology:

If computers were inﬁnitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice, but you would most often use whichever method was the easiest to implement.

Efficiency:

Algorithm efficiency A measure of the average execution time necessary for an algorithm to complete work on a set of data. Algorithm efficiency is characterized by its order. Typically a bubble sort algorithm will have efficiency in sorting *N* items proportional to and of the order of *N*2, usually written O(*N*2). This is because an average of *N*/2 comparisons are required *N*/2 times, giving *N*2/4 total comparisons, hence of the order of *N*2. In contrast, quicksort has an efficiency O(*N* log2*N*).  
  
If two algorithms for the same problem are of the same order then they are approximately as efficient in terms of computation. Algorithm efficiency is useful for quantifying the implementation difficulties of certain problems.

Algorithms and other technologies:

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

1.Insertion Sort:

Insertion sort, which is an efﬁcient algorithm for sorting a small number of elements, which takes as a parameter an array A[1....n]containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by A:length.) The algorithm sorts the input numbers in place: it rearranges the numbers within the array A,with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the INSERTION-SORT procedure is ﬁnished.

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

2. Analyzing algorithms:

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one processor, random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

Worst-case and average-case analysis:

we shall usually concentrate on ﬁnding only the worst-case running time, that is, the longest running time for any input of size n. 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. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[1..j]to insert element A. On average, half the elements in A [1...j] are less than j, and half the elements are greater. On average, therefore, we check half of the subarray A[1..j], and so it is about j=2.

Order of growth:

we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as an 2C for some constants a, b, and c that depend on the statement costs ci. 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 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.

The divide-and-conquer approach:

Recursive algorithms typically follow a divide-and-conquer approach. they break the problem into several subproblems 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. 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 sub problems in a straight forward manner. Combine the solutions to the subproblems into the solution for the original problem.

3.Merge Sort:

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure MERGE.A(p,q,r), where A is an array and p, q, andr are indices into the array such that p q<r. The procedure assumes that the subarrays A and Care in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p...r].

Pseudocode:

MERGE Sort(A,p,r)

if p<r

q= └(p+r)/2┘

{\displaystyle \operatorname {floor} (x)=\lfloor x\rfloor } MERGE Sort(A,p,q)

MERGE Sort(A,q+1,r)

MERGE (A,p,q,r)

##

MERGE (A,p,q,r)

n1=q-p+1;

n2=r-q;

let L[1...n1+1] and R[1...n2+1] be new arrays

for i=1 to n1

L[i]=A[p+i-1]

for j=1 to n2

R[j]=A[q+j]

L[n1+1]=

R[n2+1]=

i=1

j=1

for k=p to r

if L[i]<=R[j]

A[k]=L[i]

i=i+1

else A[k]=R[j]

j=j+1

Chapter 3: Growth of Functions

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 . Such notations are convenient for describing the worst-case running-time function T(n), which usually is deﬁned only on integer input sizes. we might extend the notation to the domain of real numbers or, alternatively, restrict it to a subset of the natural numbers.

Asymptotic notation, functions, and running times:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

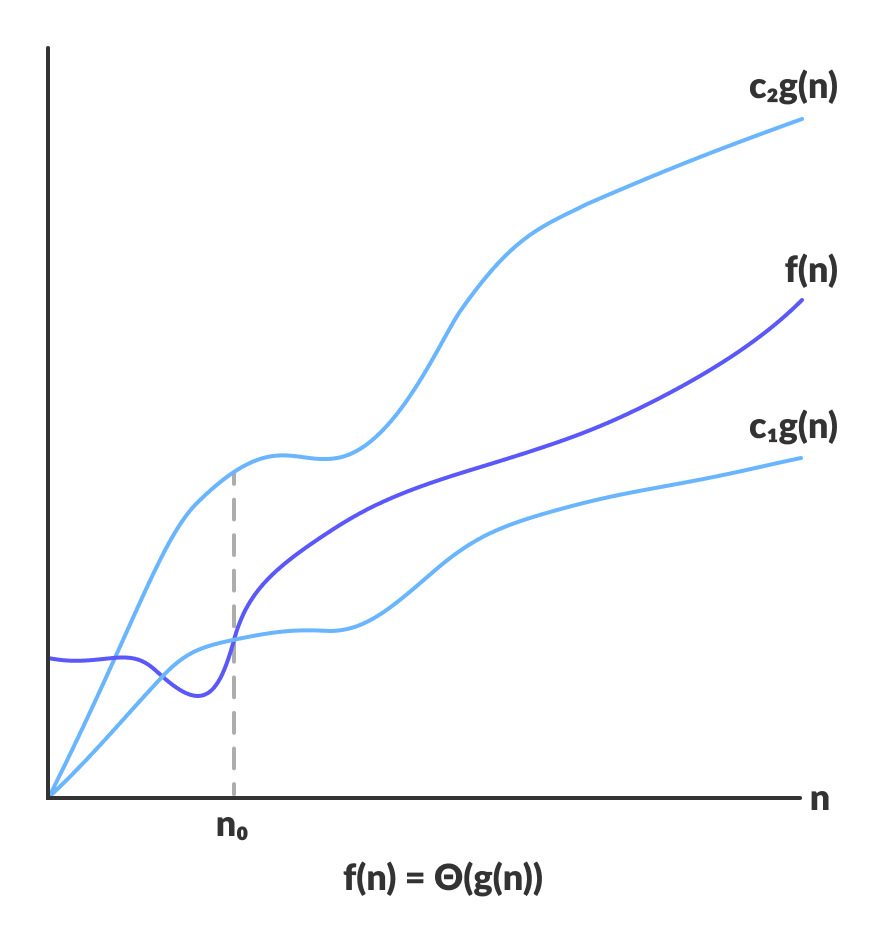
Theta Notation (Θ-notation):

Theta notation encloses the function from above and below. Since, it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }



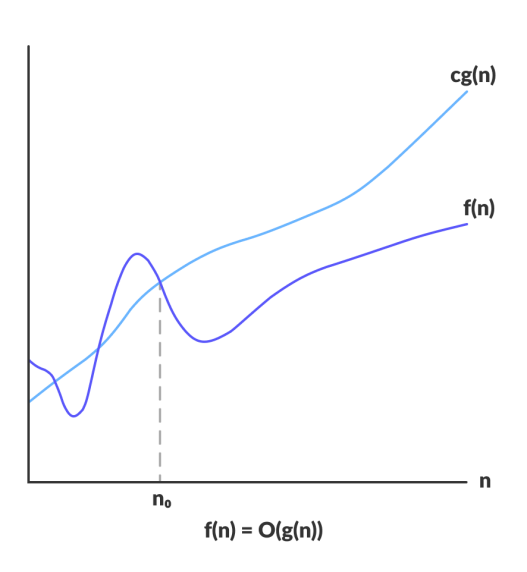
Big-O Notation (O-notation);

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.

O(g(n)) = { f(n): there exist positiv constants c and n0

such that 0 ≤ f(n) ) ≤ cg(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).



Omega Notation (Ω-notation):

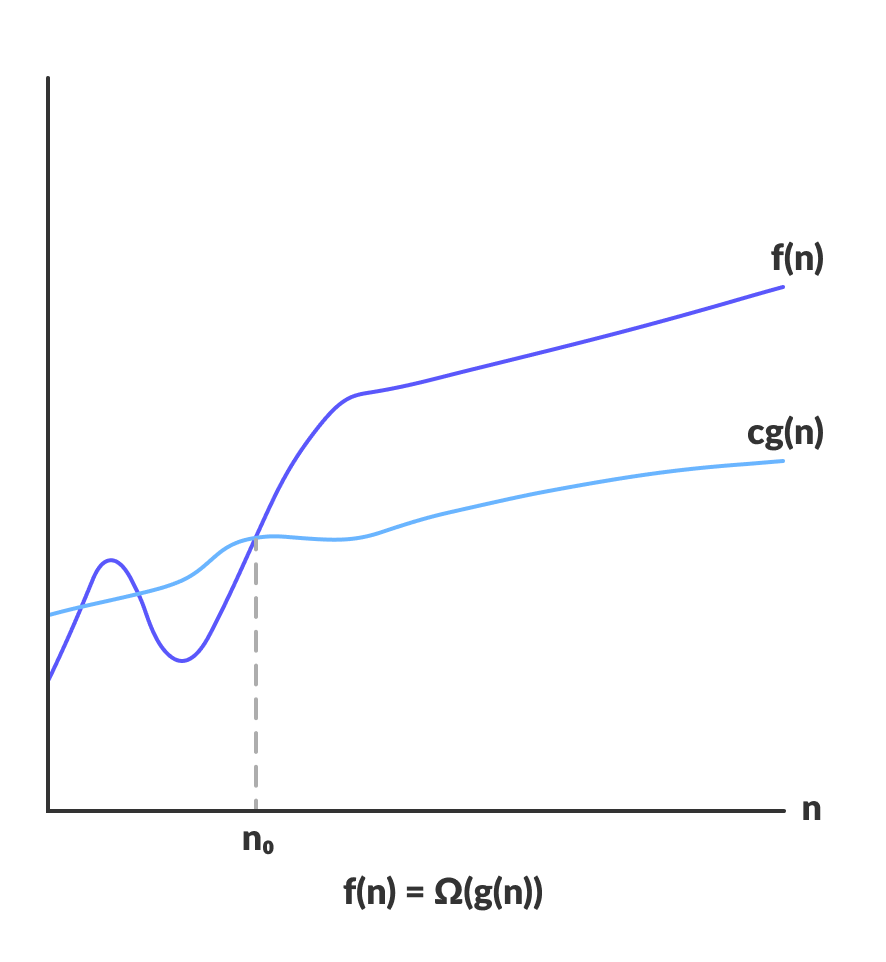
Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.

Ω(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).



2. Standard notations and common functions:

Monotonicity:

A function f(n) is monotonically increasing if m <= n implies f(m)<= f(n). Similarly, it is monotonically decreasing if m <=n implies f(m)>= f(n).A function f(n) is strictly increasing if m<n implies f(m) < f(n) and strictly decreasing if m<n implies f(m)> f(n).

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 }.

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.

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.

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.