

1.1 Introduction

1.1.1.What is an Algorithm?

Algorithm: An algorithm is a finite sequence of unambiguous instructions to solve a particular problem.

- 1) **Input.** Zero or more quantities are externally supplied.
- 2) **Output.** At least one quantity is produced.
- 3) **Definiteness.** Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
- 4) **Finiteness.** If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- 5) **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterionc; it also must be feasible.

1.1.2. Algorithm Specification

An algorithm can be specified in

- 1) Simple English
- 2) Graphical representation like flowchart
- 3) Programming language like c++/java
- 4) Combination of above methods.

Example: Combination of simple English and C++, the algorithm for **selection sort** is specified as follows.

```
for (i=1; i<=n; i++) {  
    examine a[i] to a[n] and suppose  
    the smallest element is at a[j];  
    interchange a[i] and a[j];  
}
```

Example: In C++ the same algorithm can be specified as follows. Here *Type* is a basic or user defined data type.

```
void SelectionSort(Type a[], int n)  
// Sort the array a[1:n] into nondecreasing order.  
{  
    for (int i=1; i<=n; i++) {  
        int j = i;  
        for (int k=i+1; k<=n; k++)  
            if (a[k]<a[j]) j=k;  
        Type t = a[i]; a[i] = a[j]; a[j] = t;  
    }  
}
```

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

For example, $\gcd(60, 24)$ can be computed as follows: $\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12$.

Euclid's algorithm for computing $\gcd(m, n)$

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r .

Step 3 Assign the value of n to m and the value of r to n . Go to Step 1.

Alternatively, we can express the same algorithm in a pseudocode:

ALGORITHM *Euclid*(m, n)

```
//Computes gcd(m, n) by Euclid's algorithm  
//Input: Two nonnegative, not-both-zero integers m and n  
//Output: Greatest common divisor of m and n  
while n ≠ 0 do  
    r ← m mod n  
    m ← n  
    n ← r  
return m
```

Consecutive integer checking algorithm for computing $\gcd(m, n)$

Step 1 Assign the value of $\min\{m, n\}$ to t .

Step 2 Divide m by t . If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t . If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Middle-school procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m .

Step 2 Find the prime factors of n .

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n , respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

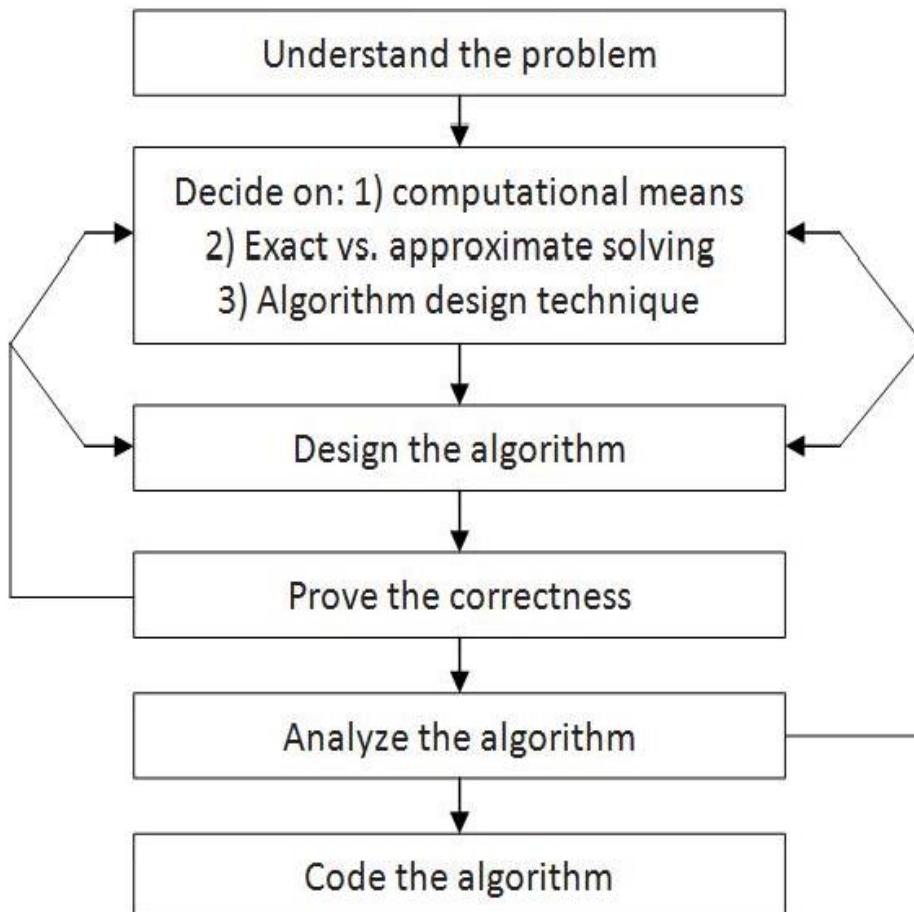
Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$\begin{aligned}60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\24 &= 2 \cdot 2 \cdot 2 \cdot 3 \\ \gcd(60, 24) &= 2 \cdot 2 \cdot 3 = 12.\end{aligned}$$

60
24
2 2 3

1.2 Algorithm design and analysis process



Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often.

Ascertaining the Capabilities of a Computational Device

Once you completely understand a problem. You need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine-a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903 1957), in collaboration with A. Burks and H. Goldstine, in 1946.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

The essence of this architecture is captured by the so-called random-access machine (RAM). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called sequential algorithms. The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmic for the foreseeable future.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly and solving it approximately. In the former case, an algorithm is called an exact algorithm; in the latter case, an algorithm is called an approximation algorithm. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures =Programs*.

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem. An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. We described Euclid's algorithm in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms. A pseudocode is a mixture of a natural language and programming language like constructs.

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its correctness. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, correctness of Euclid's algorithm for computing the greatest common divisor stems from correctness of the equality $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 6 in Exercises 1.1), the simple observation that the second number gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second number becomes 0.

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency time efficiency and space efficiency. Time efficiency indicates how fast the algorithm runs; space efficiency indicates how much extra memory the algorithm needs

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

1.3. Important Problem Types

In the limitless sea of problems one encounters in computing, there are a few areas that have attracted particular attention from researchers. By and large, interest has been driven either by the problem's practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases. In this section, we take up the most important problem types:

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

- 2 Searching
- 3 String processing
- 4 Graph problems
- 5 Combinatorial problems
- 6 Geometric problems
- 7 Numerical problems.

1.3.1. Sorting

The **sorting problem** asks us to rearrange the items of a given list in ascending order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.)

Two properties of sorting algorithms deserve special mention. A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. The second notable feature of a sorting algorithm is the amount of **extra memory** the algorithm requires. An algorithm is said to be in-place if it does not require extra memory, except, possibly, for a few memory units.

1.3.2. Searching

The **searching problem** deals with finding a given value, called a **search key**, in a given set. There are plenty of searching algorithms to choose from. They range from the straight forward **sequential search** to a spectacularly efficient but limited **binary search** and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

1.3.3. String Processing

A **string** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeroes and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}. It should be pointed out, however, that string-processing algorithms have been important for computer science.

1.3.4. Graph Problems

One of the oldest and most interesting areas in algorithmic is graph algorithms. Informally, a **graph** can be thought of as a collection of points called vertices, some of

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

which are connected by line segments called edges. (A more formal definition is given in the next section.) Graphs are an interesting subject to study for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of real-life applications, including transportation and communication networks, project scheduling, and games.

1.3.5. Combinatorial Problems

From a more abstract perspective, the traveling salesman problem and the graph coloring problem are examples of combinatorial problems. These are problems that ask (explicitly or implicitly) to find a combinatorial object-such as a permutation, a combination, or a subset-that satisfies certain constraints and has some desired property (e.g., maximizes a value or minimizes a cost).

1.3.6. Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems, including problems of constructing simple geometric shapes-triangles, circles, and so on-with an unmarked ruler and a compass. Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers-no more rulers and compasses, just bits, bytes, and good old human ingenuity. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.

1.3.7. Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

1.4. Fundamental Data Structures

Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms. **A data structure** is a way of organizing data in memory so that it can be used efficiently.

structure can be defined as a particular scheme of organizing related data items.

1.4.1. Linear Data Structures

The two most important elementary data structures are the array and the linked list. **A (one-dimensional) array is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.**



Array of n elements.

A **linked list** is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list. In a **singly linked list**, each node except the last one contains a single pointer to the next element. Another extension of the structure called the **doubly linked list**, in which every node, except the first and the last, contains pointers to both its successor and its predecessor.

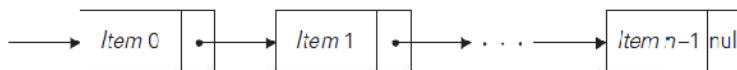


FIGURE 1.4 Singly linked list of n elements.

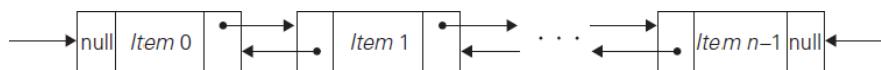


FIGURE 1.5 Doubly linked list of n elements.

A **list** is a finite sequence of data items, i.e. a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element. Two special types of lists, stacks and queues, are particularly important.

A **stack** is a list in which insertions and deletions can be done only at the end. This end is called the top because a stack is usually visualized horizontally but vertically. As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in the "last-in-first-out" (LIFO) fashion, exactly as the stack of plates does if we can remove only the top plate or add another plate to top of the stack.

A **queue**, on the other hand, is a list from which elements are deleted from one end of the structure, called the front (this operation is called **dequeue**), and new elements are added to the other end, called the rear (this operation is called **enqueue**). Consequently, a queue operates in the "first-in-first-out" (FIFO) fashion (akin, say, to a queue of customers served by a single teller in a bank). Queues also have many

important applications, including several algorithms for graph problems.

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

1.4.2. Graphs

A graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs." Formally, a **graph** $G = \{V, E\}$ is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called **edges**.

If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u) , we say that the vertices u and v are **adjacent** to each other and that they are connected by the **undirected edge** (u, v) . We call the vertices u and v **endpoints** of the edge (u, v) and say that u and v are incident to this edge; we also say that the edge (u, v) is incident to its endpoints u and v . A graph G is called **undirected** if every edge in it is undirected.

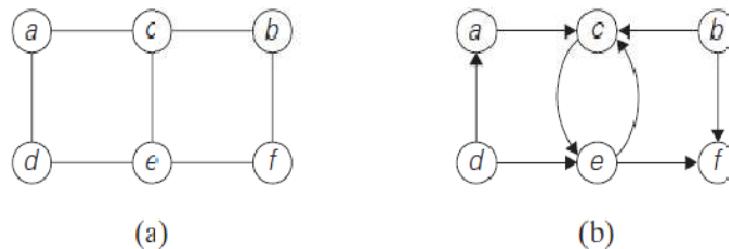
If a pair of vertices (u, v) is not the same as the pair (v, u) , we say that the edge (u, v) is **directed** from the vertex u , called the edge's tail, to the vertex v , called the edge's **head**. We also say that the edge (u, v) leaves u and enters v . A graph whose every edge is directed is called **directed**. Directed graphs are also called **digraphs**.

The graph depicted in Figure (a) has six vertices and seven undirected edges:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

$$V = \{a, b, c, d, e, f\}, E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$



(a) Undirected graph. (b) Digraph.

Graph Representations- Graphs for computer algorithms are usually represented in one of two ways: the **adjacency matrix** and **adjacency lists**.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

The **adjacency matrix** of a graph with n vertices is an $n \times n$ Boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i^{th} row and the j^{th} column is equal to 1 if there is an edge from the i^{th} vertex to the j^{th} vertex, and equal to 0 if there is no such edge.

The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a	→ c	→ d
b	→ c	→ f
c	→ a	→ b → e
d	→ a	→ e
e	→ c	→ d → f
f	→ b	→ e

(b)

FIGURE ... (a) Adjacency matrix and (b) adjacency lists of the graph in Figure

Weighted Graphs: Weighted graphs A weighted graph (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called **weights** or **costs**. An interest in such graphs is motivated by numerous real-life applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

Both principal representations of a graph can be easily adopted to accommodate weighted graphs. If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the i^{th} to the j^{th} vertex if there is such an edge and a special symbol, e.g., ∞ , if there is no such edge. Such a matrix is called the **weight matrix** or **cost matrix**. This approach is illustrated in Figure 1.Sb. (For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.) Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge

Paths and cycles among many interesting properties of graphs, two are important for a great number of applications: **connectivity** and **acyclicity**. Both are based on the notion of a path.

A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.

If all vertices of a path are distinct, the path is said to be **simple**. The **length** of a path is the total number of vertices in a vertex sequence defining the path minus one, which is the same as the number of edges in the path.

For example, a, c, b, f is a simple path of length 3 from a to f in the graph of Figure 1.6a, whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f.

In the case of a directed graph, we are usually interested in directed paths. A **directed path** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For example, a, c, e, f is a directed path from a to f in the graph of Figure 1.6b.

A graph is said to be **connected** if for every pair of its vertices u and v there is a path from u to v. informally, this property means that if we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece. If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph. Formally, a **connected component** is a maximal (not expandable via an inclusion of an extra vertex) connected sub graph3 of a given graph. For example, the graphs of Figures 1.6a and 1.6 b are connected, while the graph in Figure 1.9 is not because there is no path, for example, from a to f. The graph in Figure 1.9 has two connected components with vertices (a, b, c, d, e) and (f, g, h, i], respectively.

It is important to know for many applications whether or not a graph under consideration has cycles. A **cycle** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, j, h, i, g, .f is a cycle in the graph of Figure 1.9. A graph with no cycles is said to be acyclic. We discuss acyclic graphs in the next subsection.

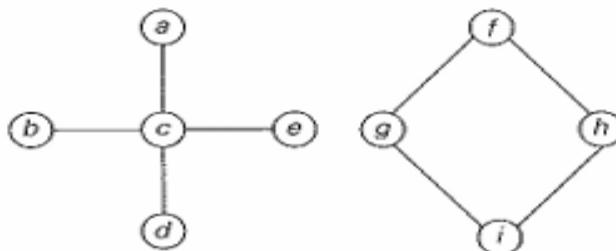


FIGURE 1.9 Graph that is not connected

1.4.3 Trees

A **tree** (more accurately, **free tree**) is a connected acyclic graph (Figure 1.10a). A graph that has no cycles but is not necessarily connected is called **a forest**: each of its connected components is a tree (Figure 1.10b).

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices:

$$|E| = |V| - 1.$$

As the graph of Figure 1.9 demonstrates, this property is necessary but not sufficient for a graph to be a tree. However, for connected graphs it is sufficient and hence provides a convenient way of checking whether a connected graph has a cycle.

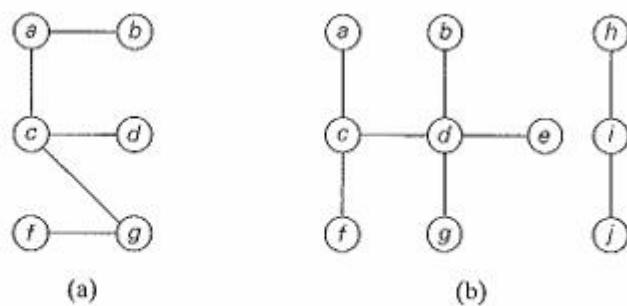


FIGURE 1.10 (a) Tree. (b) Forest.

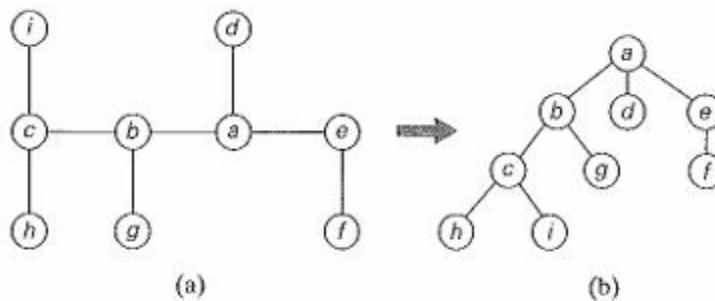


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Rooted trees another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the **root** of the so-called **rooted tree**. A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (Level 1), the vertices two edges apart from the root below that (level2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called **ancestors** of v . The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as **proper ancestors**. If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq P v$), u is said to be the **parent** of v and v is called a **child** of u ; vertices that have the same parent are said to be **siblings**. A vertex with no children is called a **leaf**; a vertex with at least one child is called **parental**.

All the vertices for which a vertex v is an ancestor are said to be **descendants** of v ; the **proper descendants** exclude the vertex v itself. All the descendants of a vertex v with all the edges connecting them form the **subtree** of T rooted at that vertex. Thus, for the tree of Figure 1.11b, the root of the tree is a ; vertices d, g, f, h , and i are leaves, while vertices a, b, e , and c are parental; the parent of a ; the children of b are c and g ; the siblings of b and e ; the vertices of the subtree rooted at b are $\{b, c, g, h, i\}$.

The depth of a vertex v is the length of the simple path from the root to v . The height of a tree is the length of the longest simple path from the root to a leaf. For example,

the depth of vertex c in the tree in Figure 1.11b is 2, and the height of the tree is 3. Thus, if we count tree levels top down starting with 0 for the root's level, the depth of a vertex is simply its level in the tree, and the tree's height is the maximum level of its vertices. (You should be alert to the fact that some authors define the height of a tree as the number of levels in it; this makes the height of a tree larger by 1 than the height defined as the length of the longest simple path from the root to a leaf.)

Ordered trees An ordered tree is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right. A **binary tree** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a **left child** or a **right child** of its parent. The subtree with its root at the left (right) child of a vertex is called the **left (right) subtree** of that vertex. An example of a binary tree is given in Figure 1.12a.

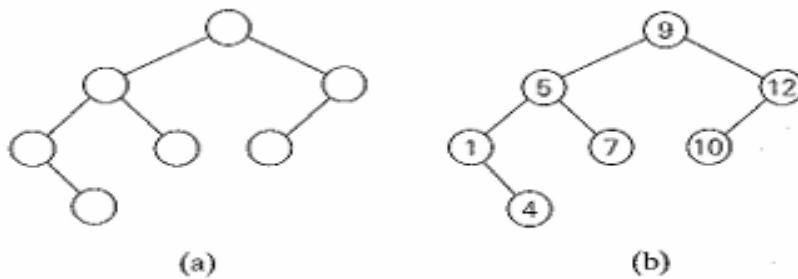


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a. Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called **binary search trees**. Binary trees and binary search trees have a wide variety of applications in computer science; you will encounter some of them throughout the book. In particular, binary search trees can be generalized to more general kinds of search trees called **multiway search trees**, which are indispensable for efficient storage of very large files on disks.

1.4.4 Sets and Dictionaries

A **set** can be described as an unordered collection (possibly empty) of distinct items called **elements** of the set. A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2, 3, 5, 7\}$) or by specifying a property that all the set's elements and only they must satisfy (e.g., $S = \{n : n \text{ is a prime number smaller than } 10\}$).

The most important **set operations** are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

Sets can be **implemented** in computer applications in two ways. The first considers only sets that are subsets of some large set U , called the universal set. If set U has n elements, then any subset S of U can be represented by a bit string of size n , called a **bit vector**, in which the i^{th} element is 1 if and only if the i^{th} element of U is included in set S .

The second and more common way to represent a set for computing purposes is to use the **list** structure to indicate the set's elements. This is feasible only for finite sets. The requirement for uniqueness is sometimes circumvented by the introduction of a **multiset**, or a **bag**, an unordered collection of items that are not necessarily distinct. Note that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

Dictionary: In computing, the operations we need to perform for a sector a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the **dictionary**.

A number of applications in computing require a dynamic partition of some n -elements set into a collection of disjoint subsets. After being initialized as a collection of n one-elementsubsets, the collection is subjected to a sequence of intermixed union and search operations. This problem is called the **set union** problem.

1.5 Analysis Framework

1.5.1 Measuring an Input's Size

It is observed that almost all algorithms **run longeron larger inputs**. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the **algorithm's input size**. There are situations, where the choice of a **parameter indicating an input size does matter**.

We should make a special note about measuring the size of inputs for algorithms involving **properties of numbers** (e.g., checking whether a given integer is prime). For such algorithms, computer scientists prefer measuring size by the number b of bits in the n 's binary representation = $\lceil \log_2 n \rceil + 1$. This metric usually gives a better idea about the efficiencyof algorithms in question.

1.5.2 Units for Measuring Running Time

To measure an algorithm's efficiency, we would like to have a **metric that does**

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

not depend on these extraneous factors. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

For example, most **sorting** algorithms work by **comparing elements** (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.

As another example, algorithms for **matrix multiplication** and **polynomial evaluation** require two arithmetic operations: **multiplication and addition**.

Let cop be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula:

$$T(n)=copC(n)$$

Unless n is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time. It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the counts **order of growth** to within a constant multiple for large-size inputs.

1.5.3 Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? Because for large values of n , it is the function's order of growth that counts: just look at table which contains values of a few functions particularly important for analysis of algorithms.

Table: Values of several functions important for analysis of algorithms

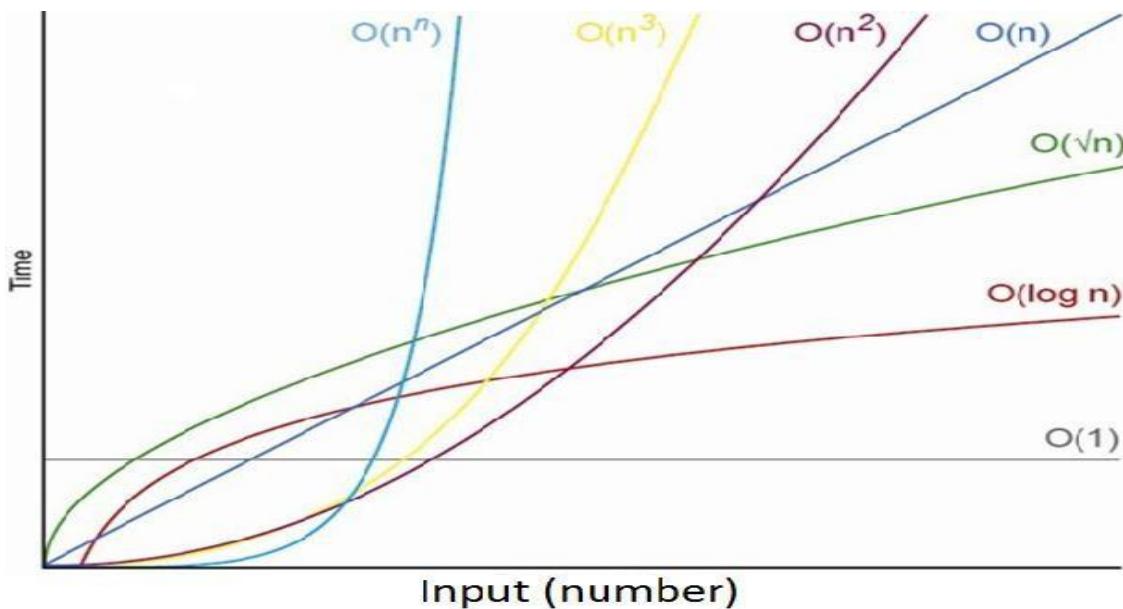
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

N	1	$\log N$	N	$N \log N$	N^2	N^3	2^N	$N!$
1	1	0	1	0	1	1	2	1
2	1	1	2	2	4	8	4	2
3	1	2	3	5	9	27	8	6
4	1	2	4	8	16	64	16	24
5	1	2	5	12	25	125	32	120
6	1	3	6	16	36	216	64	720
7	1	3	7	20	49	343	128	5040
8	1	3	8	24	64	512	256	40320
9	1	3	9	29	81	729	512	362880
10	1	3	10	33	100	1000	1024	3628800
11	1	3	11	38	121	1331	2048	39916800

fast ↓	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">constant</td></tr> <tr><td style="padding: 2px;">$\log n$</td><td style="padding: 2px;">logarithmic</td></tr> <tr><td style="padding: 2px;">n</td><td style="padding: 2px;">linear</td></tr> <tr><td style="padding: 2px;">$n \log n$</td><td style="padding: 2px;">$n \log n$</td></tr> <tr><td style="padding: 2px;">n^2</td><td style="padding: 2px;">quadratic</td></tr> <tr><td style="padding: 2px;">n^3</td><td style="padding: 2px;">cubic</td></tr> <tr><td style="padding: 2px;">2^n</td><td style="padding: 2px;">exponential</td></tr> <tr><td style="padding: 2px;">$n!$</td><td style="padding: 2px;">factorial</td></tr> </table>	1	constant	$\log n$	logarithmic	n	linear	$n \log n$	$n \log n$	n^2	quadratic	n^3	cubic	2^n	exponential	$n!$	factorial	High time efficiency
1	constant																	
$\log n$	logarithmic																	
n	linear																	
$n \log n$	$n \log n$																	
n^2	quadratic																	
n^3	cubic																	
2^n	exponential																	
$n!$	factorial																	
slow ↓		↓ low time efficiency																



1.6 Performance Analysis

There are two kinds of efficiency: **time efficiency** and **space efficiency**.

- Time efficiency indicate how fast an algorithm in question runs;
- Space efficiency deals with the extra space the algorithm requires.

In the early days of electronic computing, both resources **time** and **space** were at a premium. The research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, we primarily concentrate on time efficiency.

1.6.1 Space complexity

Total amount of computer memory required by an algorithm to complete its execution is called as **space complexity** of that algorithm. The Space required by an algorithm is the sum of following components

- A **fixed** part that is independent of the input and output. This includes memory space for codes, variables, constants and so on.
- A **variable** part that depends on the input, output and recursion stack. (We call these parameters as instance characteristics)

Space requirement $S(P)$ of an algorithm P , $S(P) = c + Sp$ where c is a constant depends on the fixed part, Sp is the instance characteristics

Example-1: Consider following algorithm **abc()**

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

```
float abc(float a, float b, float c)
{   return (a + b + b*c + (a+b-c)/(a+b) + 4.0);
}
```

Here fixed component depends on the size of a,b and c. Also instance characteristics $Sp=0$

Example-2: Let us consider the algorithm to find sum of array. For the algorithm given here the problem instances are characterized by n , the number of elements to be summed. The space needed by $a[]$ depends on n . So the space complexity can be written as; $Ssum(n) \geq (n+3); n$ for a [], One each for n, i and s.

```
float Sum(float a[], int n)
{   float s = 0.0;
    for (int i=1; i<=n; i++)
        s += a[i];
    return s;
}
```

1.6.2 Time complexity

Usually, the execution time or run-time of the program is referred as its time complexity denoted by tp (instance characteristics). This is the sum of the time taken to execute all instructions in the program. Exact estimation runtime is a complex task, as the number of instructions executed is dependent on the input data. Also different instructions will take different time to execute. So for the estimation of the time complexity **we count only the number of program steps**. We can determine the **steps needed by a program** to solve a particular problem instance in two ways.

Method-1: We introduce a new variable ***count*** to the program which is initialized to zero. We also introduce statements to increment ***count*** by an appropriate amount into the programs when each time original program executes, the ***count*** also incremented by the step count.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

Example: Consider the algorithm **sum()**. After the introduction of the count the program will be as follows. We can estimate that invocation of **sum()**executes

```
float Sum(float a[], int n)
{
    float s = 0.0;
    count++; // count is global
    for (int i=1; i<=n; i++) {
        count++; // For 'for'
        s += a[i]; count++; // For assignment
    }
    count++; // For last time of 'for'
    count++; // For the return
    return s;
}
```

total number of **2n+3** steps.

Method-2: Determine the step count of an algorithm by building a table in which we list the total number of steps contributed by each statement. An example is shown below. The code will find the sum of n numbers

Statement	s/e	frequency	total steps
float Sum(float a[], int n)	0	–	0
{ float s = 0.0;	1	1	1
for (int i=1; i<=n; i++)	1	n + 1	n + 1
s += a[i];	1	n	n
return s;	1	1	1
}	0	–	0
Total			2n + 3

Example: Matrix addition

Statement	s/e	freq	total
void Add(Type a[][][SIZE], ...)	0	-	0
{ for (int i=1; i<=m; i++)	1	$m + 1$	$m + 1$
for (int j=1; j<=n; j++)	1	$m(n + 1)$	$mn + m$
c[i][j] = a[i][j]			
+ b[i][j];	1	mn	mn
}	0	-	0
Total			$2mn + 2m + 1$

3. Fibonacci series

```

algorithm Fibonacci(n)                                Step Count
{
    if n <= 1 then                                     ---- 1
        output 'n'
    else
        f2 = 0;                                         ---- 1
        f1 = 1;                                         ---- 1

        for i = 2 to n do                            ---- n
        {
            f = f1 + f2;                           ---- n - 1
            f2 = f1;                             ---- n - 1
            f1 = f;                               ---- n - 1
        }
        output 'f'                                    ---- 1
}
Total no of steps= 4n + 1

```

1.6.3 Trade-off

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

1.6.4 Worst-Case, Best-Case, and Average-Case Efficiencies

```
ALGORITHM SequentialSearch( $A[0..n - 1]$ ,  $K$ )
  //Searches for a given value in a given array by sequential search
  //Input: An array  $A[0..n - 1]$  and a search key  $K$ 
  //Output: The index of the first element of  $A$  that matches  $K$ 
  //          or  $-1$  if there are no matching elements
   $i \leftarrow 0$ 
  while  $i < n$  and  $A[i] \neq K$  do
     $i \leftarrow i + 1$ 
  if  $i < n$  return  $i$ 
  else return  $-1$ 
```

Clearly, the running time of this algorithm can be quite different for the same list size n . In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n : $C_{\text{worst}}(n) = n$.

The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size n , which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size n , which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. For example, for sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly, $C_{\text{best}}(n) = 1$.

Let us consider again sequential search. The standard assumptions are that **(a) the probability of a successful search is equal to p (0 <= p <= 1)** and **(b) the probability of the first match occurring in the i th position of the list is the same for every i** .

Under these assumptions-the validity of which is usually difficult to verify, their reasonableness notwithstanding-we can find the average number of key comparisons $C_{\text{avg}}(n)$ as follows. In the case of a successful search, the probability of the first match occurring in the i th position of the list is p / n for every i , and the number of comparisons made by the algorithm in such a situation is obviously i . In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1-p)$.

Therefore,

$$\begin{aligned}
 C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + i \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\
 &= \frac{p}{n} [1 + 2 + \dots + i + \dots + n] + n(1-p) \\
 &= \frac{p}{n} \frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).
 \end{aligned}$$

This general formula yields some quite reasonable answers. For example, if $p = 1$ (i.e., the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; i.e., the algorithm will inspect, on average, about half of the list's elements. If $p = 0$ (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

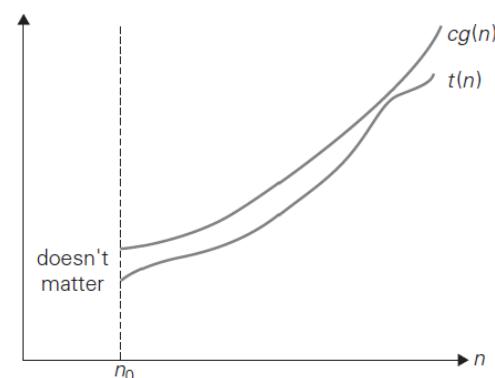
1.7 Asymptotic Notations

The efficiency analysis frame work concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: O (bigoh), Ω (bigomega), Θ (bigtheta)and o (little oh)

1.7.1 Big-Oh notation

Definition: A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0.$$



Big-oh notation: $t(n) \in O(g(n))$.

Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$. Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0 .

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5\text{)} = 101n \leq 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0 . For example, we could also reason that

$$100n + 5 \leq 100n + 5n \text{ (for all } n \geq 1\text{)} = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

Example: To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

$$5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2,$$

thus $5n^2 + 3n + 20 = O(n^2)$.

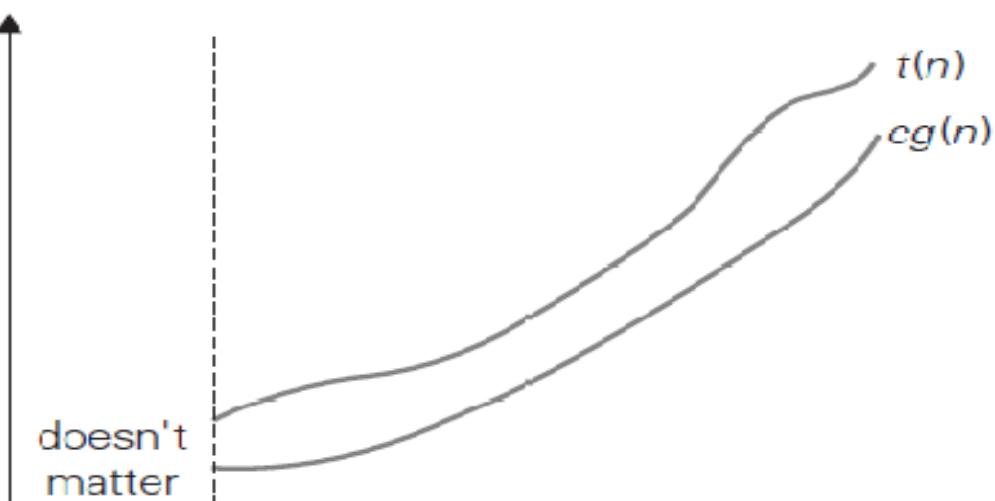
Example: To prove $100n + 5 \in O(n^2)$

$$100n + 5 \leq 105n^2. (c=105, n_0=1)$$

Example: To prove $n^2 + n = O(n^3)$

Take $c = 1+1=2$, if $n \geq n_0=1$, then $n^2 + n = O(n^3)$

1.7.2 Omega notation



A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that
 $t(n) \geq c g(n)$ for all $n \geq n_0$.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \quad \text{for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

Example: $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, but $100n + 5 \notin \Omega(n^2)$.

Example: $n^3 + 4n^2 = \Omega(n^2)$

- Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$
- It is not too hard to see that if $n \geq 0$,

$$n^3 \leq n^3 + 4n^2$$

- We have already seen that if $n \geq 1$,

$$n^2 \leq n^3$$

- Thus when $n \geq 1$,

$$n^2 \leq n^3 \leq n^3 + 4n^2$$

- Therefore,

$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$

- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$
 (by definition of Big- Ω , with $n_0 = 1$, and $c = 1$.)

1.7.3 Theta notation

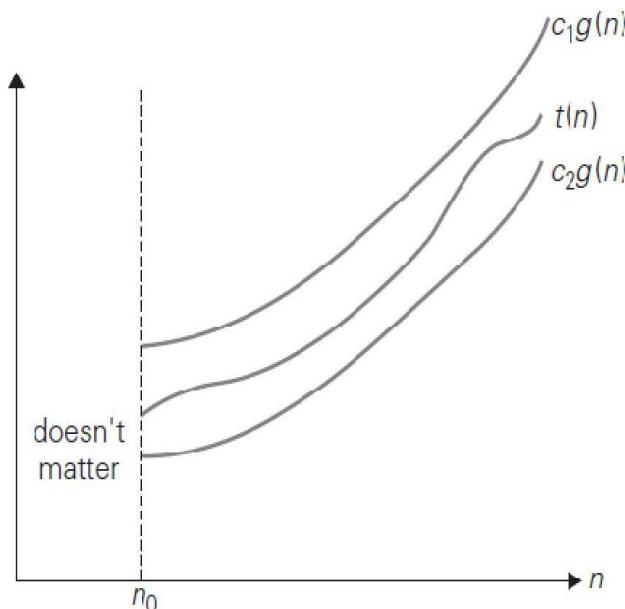
A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$,

if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n ,

i.e., if there exist Theta Notation some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

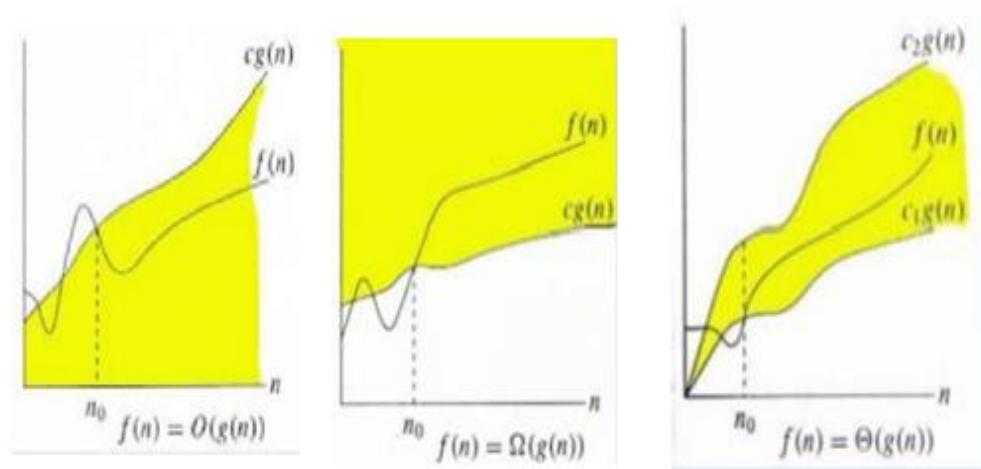
$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0.$$

Big-theta notation: $t(n) \in \Theta(g(n))$.



Graphical representation

- Which one best to represent order of growth



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof:

- When $n \geq 1$,

$$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$$

- When $n \geq 0$,

$$n^2 \leq n^2 + 5n + 7$$

- Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big- Θ , with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Proof:

- Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$$

- Thus,

$$\frac{1}{2}n^2 + 3n = O(n^2)$$

- Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$$

- So

$$\frac{1}{2}n^2 + 3n = \Omega(n^2)$$

- Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

$$\frac{1}{2}n^2 + 3n = \Theta(n^2)$$

For example, let us prove that $\frac{1}{2}n(n - 1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n - 1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n \cdot \frac{1}{2}n \quad (\text{for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Little Oh

- The function $f(n) = o(g(n))$ [i.e f of n is a little oh of g of n] if and only if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

The function $3n + 2 = o(n^2)$ since $\lim_{n \rightarrow \infty} \frac{3n+2}{n^2} = 0$.

- For comparing the order of growth **limit** is used

Little Oh

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

1.7.4 Basic asymptotic efficiency Classes

<i>Class</i>	<i>Name</i>	<i>Comments</i>
1	<i>constant</i>	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	<i>logarithmic</i>	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	<i>linear</i>	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	<i>linearithmic</i>	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.

n^2	<i>quadratic</i>	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	<i>cubic</i>	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	<i>exponential</i>	Typical for algorithms that generate all subsets of an n -element set. Often, the term “exponential” is used in a broader sense to include this and larger orders of growth as well.
$n!$	<i>factorial</i>	Typical for algorithms that generate all permutations of an n -element set.

1.8.1 Mathematical Analysis of Non-recursive & Recursive Algorithms

Mathematical Analysis of Non-recursive Algorithms

1. Decide on a **parameters** indicating an input's size.
2. Identify the algorithm's **basic operation**.
3. Check whether the **number of times the basic operation is executed depends only on the size of an input**.
If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the **number of times the algorithm's basic operation is executed**.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, **establish its order of growth**.

Example-1: To find maximum element in the given array

ALGORITHM *MaxElement(A[0..n – 1])*

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n – 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n – 1 do
    if A[i] > maxval           Best, Worst, Average case exist?
        maxval ← A[i]
return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Let us denote $C(n)$ the number of times this comparison is executed and try to find a formula expressing it as a function of size n . The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n - 1$ (inclusively). Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing else but 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n). \quad \blacksquare$$

Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i \quad (\text{R1})$$

$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i \quad (\text{R2})$$

and two summation formulas

$$\sum_{i=l}^u 1 = u - l + 1 \text{ where } l \leq u \text{ are some lower and upper integer limits} \quad (\text{S1})$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad (\text{S2})$$

(Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.)

Example-2: To check whether all the elements in the given array are distinct

ALGORITHM *UniqueElements(A[0..n - 1])*

```

//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n - 1]
//Output: Returns “true” if all the elements in A are distinct
//          and “false” otherwise
for i ← 0 to n - 2 do
    for j ← i + 1 to n - 1 do           Best, Worst, Average case exist?
        if A[i] = A[j] return false
return true

```

$$\begin{aligned}
 C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
 &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).
 \end{aligned}$$

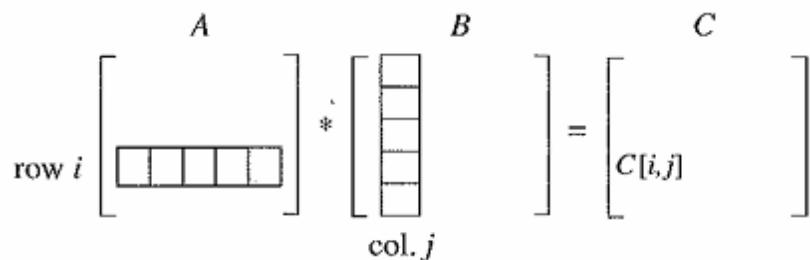
We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \dots + 1 \stackrel{\text{(§2)}}{=} \frac{(n-1)n}{2}.$$

Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements. ■

Example-3: To perform matrix multiplication

EXAMPLE 3 Given two n -by- n matrices A and B , find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an n -by- n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B :



where $C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \leq i, j \leq n-1$.

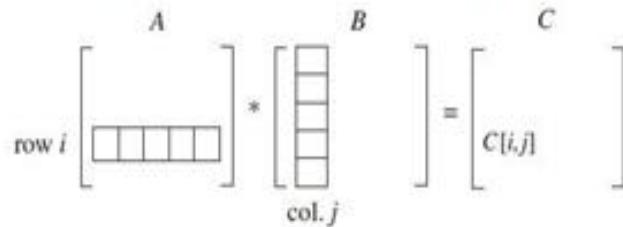
ALGORITHM *MatrixMultiplication(A[0..n - 1, 0..n - 1], B[0..n - 1, 0..n - 1])*

```

//Multiplies two square matrices of order  $n$  by the definition-based algorithm
//Input: Two  $n \times n$  matrices  $A$  and  $B$ 
//Output: Matrix  $C = AB$ 
for  $i \leftarrow 0$  to  $n - 1$  do
    for  $j \leftarrow 0$  to  $n - 1$  do Best, Worst, Average case exist?
         $C[i, j] \leftarrow 0.0$ 
        for  $k \leftarrow 0$  to  $n - 1$  do
             $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$ 
return  $C$ 

```

$$C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n-1]B[n-1, j]$$



Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n - 1$. Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now we can compute this sum by using formula (S1) and rule (R1) (see above). Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Example-4: To count the bits in the binary representation

ALGORITHM *Binary(n)*

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← [n/2]
return count
```

Best, Worst, Average case exist?

The basic operation is $count=count + 1$ repeats $\lfloor \log_2 n \rfloor + 1$ number of times

First, notice that the most frequently executed operation here is not inside the

while loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop's variable takes on only a few values between its lower and upper limits; therefore we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison $n > 1$ will be executed is actually $[\log_2 n] + 1$ -the number of bits in the binary representation of n according to formula (2.1)

1.9.1 Analysis of Recursive Algorithms.

Analysis of Recursive Algorithms

1. Decide on a parameter indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n . Since

$$n! = 1 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM $F(n)$

```
//Computes  $n!$  recursively
//Input: A nonnegative integer  $n$ 
//Output: The value of  $n!$ 
if  $n = 0$  return 1
else return  $F(n - 1) * n$ 
```

$$M(n) = M(n-1) + \frac{1}{F(n-1)} \quad \begin{matrix} \text{to compute} \\ F(n-1) \end{matrix} \quad \begin{matrix} \text{to multiply} \\ F(n-1) \text{ by } n \end{matrix} \quad \text{for } n > 0.$$

$$\boxed{M(n) = M(n-1) + 1 \quad \text{for } n > 0, \\
M(0) = 0.}$$

- We can use backward substitutions method to solve this

$$\begin{aligned}
 M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\
 &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\
 &= [M(n - 3) + 1] + 2 = M(n - 3) + 3. \\
 &\quad \vdots \\
 &= M(n - i) + i = \dots = M(n - n) + n = n.
 \end{aligned}$$

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,⁵ whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

$$F(n) = F(n - 1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = M(n - 1) + \frac{1}{\substack{\text{to compute} \\ F(n-1)}} \frac{1}{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

$M(0) = 0.$

the calls stop when $n = 0$ no multiplications when $n = 0$

Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n - 1) + 1 \quad \text{for } n > 0, \quad (2.2)$$

$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$F(n) = F(n - 1) \cdot n \quad \text{for every } n > 0,$$

$$F(0) = 1.$$

$$\begin{aligned} M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\ &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\ &= [M(n - 3) + 1] + 2 = M(n - 3) + 3. \end{aligned}$$

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n - 1) + 1 = \dots = M(n - i) + i = \dots = M(n - n) + n = n.$$

Example-2: Tower of Hanoi puzzle.

- In this puzzle, There are n disks of different sizes that can slide onto any of three pegs.
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary.
- We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

- The problem has an elegant recursive solution
- To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary),
 - we first move recursively $n-1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary),
 - then move the largest disk directly from peg 1 to peg 3, and,
 - finally, move recursively $n-1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- If $n = 1$, we move the single disk directly from the source peg to the destination peg.

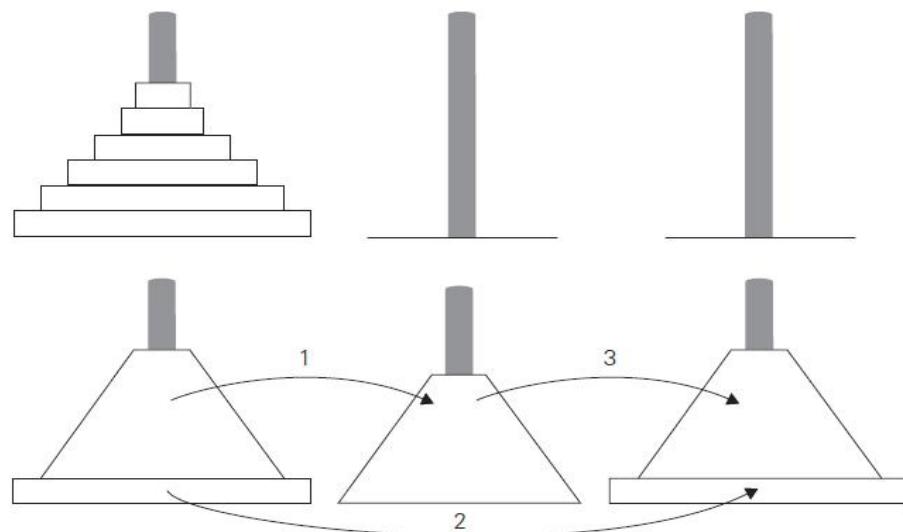


Figure:Recursive solution to the Tower of Hanoi puzzle

Algorithm:

TowerOfHanoi(n, source, dest, aux)

If n == 1, then

move disk from source to dest else

TowerOfHanoi (n - 1, source, aux, dest)

TowerOfHanoi (n - 1, aux, dest, source)

End if

Recurrence relation for total number of moves

The number of moves **M(n)** depends only on n. The recurrence equation is

$$M(n) = M(n - 1) + 1 + M(n - 1) \quad \text{for } n > 1.$$

We have the following recurrence relation for the number of moves M(n):

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 && \text{for } n > 1 \\ M(1) &= 1. \end{aligned}$$

- We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 && \text{sub. } M(n - 1) = 2M(n - 2) + 1 \\ &= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 && \text{sub. } M(n - 2) = 2M(n - 3) + 1 \\ &= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1. \end{aligned}$$

- The pattern of the first three sums on the left suggests that the next one will be $2^4M(n - 4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^iM(n - i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 = 2^iM(n - i) + 2^i - 1$$

- Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

$$\begin{aligned} M(n) &= 2^{n-1}M(n - (n - 1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \end{aligned}$$

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 \quad \text{for } n > 1, \\ M(1) &= 1. \end{aligned} \tag{2.3}$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 && \text{sub. } M(n - 1) = 2M(n - 2) + 1 \\ &= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 && \text{sub. } M(n - 2) = 2M(n - 3) + 1 \\ &= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n - 4) + 2^3 + 2^2 + 2 + 1$ and, generally, after i substitutions, we get

$$M(n) = 2^iM(n - i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 = 2^iM(n - i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned} M(n) &= 2^{n-1}M(n - (n - 1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Example-3: To count bits of a decimal number in its binary representation

Example 3

ALGORITHM *BinRec(n)*

```
//Input: A positive decimal integer n  
//Output: The number of binary digits in n's binary representation  
if n = 1 return 1  
else return BinRec([n/2]) + 1
```

- Basic operation is **Addition**
- The recurrence relation can be written as

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$

- Assuming $n = 2^k$

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 \quad \text{for } k > 0, \\ A(2^0) &= 0. \end{aligned}$$

The standard approach to solving such a recurrence is to **solve it only for $n=2^k$** and then take advantage of the theorem called the **smoothness rule** which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n .

Recurrence relation for basic operation

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 \quad \text{for } k > 0, \\ A(2^0) &= 0. \end{aligned}$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
 A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
 &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
 &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
 &\quad \dots \\
 &= A(2^{k-i}) + i && \dots \\
 &\quad \dots \\
 &= A(2^{k-k}) + k.
 \end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

Solve the following recurrence relation.

- a) $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$
- b) $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$
- c) $x(n) = x(n-1) + n$ for $n > 1$, $x(0) = 0$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA - DAA - UNIT-I

$$x(n) = x(n-1) + 5, n \geq 1$$

$$x(1) = 0$$

$$x(n) = x(n-1) + 5$$

$$= [x(n-2) + 5] + 5$$

$$= x(n-2) + 10$$

$$= [x(n-3) + 5] + 10$$

$$= x(n-3) + 15$$

$$= x(n-3) + 5 \times 3$$

$$= x(n-i) + 5 \times i \quad i = n-1$$

$$= x(n-(n-1)) + 5(n-1)$$

$$= x(n-n+1) + 5n-5$$

$$= x(1) + 5n-5$$

$$= 0 + 5n + 5 = \underline{\underline{O(n)}}$$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA - DAA - UNIT-I

$$= 3^3 x(n-3)$$

$$= 3^i x(n-i) \quad i = n-1$$

$$= 3^{n-1} x(n-(n-1))$$

$$= 3^{n-1} x(n-n+1)$$

$$= 3^{n-1} x(1) = 3^{n-1} \cdot 4$$

$$= 3^n \Rightarrow O(3^n)$$

$$x(n) = 3x(n-1)$$

$$x(1) = 4$$

$$x(n) = 3x(n-1)$$

$$= 3[3x(n-2)]$$

$$= 9x(n-2)$$

$$= 9[3x(n-3)]$$

$$= 27x(n-3)$$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA - DAA - UNIT-I

$$x(n) = x(n-1) + n$$

$$x(0) = 0$$

$$x(n) = x(n-1) + n$$

$$= x(n-2) + n-1 + n$$

$$= x(n-2) + 2n - 1$$

$$= x(n-3) + n-2 + 2n - 1$$

$$= x(n-3) + 3n - 3.$$

$$= x(n-i) + n \times i - i$$

i = 1 to n

$$= x(n-n) + n \times n - n$$

$$= x(0) + n^2 - n$$

$$= 0 + n^2 - n$$

$$= \underline{\underline{O(n^2)}}$$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

TWO MARKS QUESTIONS

1. What is an Algorithm? What are the criteria for writing an algorithms?
2. What are the methods of specifying an algorithms?
3. List the steps of Algorithm design and analysis process
4. What is exact algorithm and approximation algorithm? Give example.
5. List the important Problem Types.
6. Define the different methods for measuring algorithm efficiency.
7. Write the Euclid algorithm to find the GCD of 2 numbers.
8. What are combinatorial problems? Give an example.
9. What are following data structures?
 - a) Single linked list
 - b) double linked list
 - c) stack
 - d) queue
 - e) graph
 - f) tree
10. Explain the terms (w.r.t graph):
 - a) Directed graph
 - b) undirected graph
 - c) adjacency matrix
 - d) adjacency lists
 - e) weighted graph
 - f) path
 - g) cycle
11. Explain the terms (w.r.t trees)
 - a) free tree
 - b) forest
 - c) rooted tree
 - d) ordered tree
 - e) binary search tree
12. Define Sets and Dictionaries.
13. Define the two types of efficiencies used in algorithm.
14. What are Best case and Worst case in algorithm?
15. Why order growth necessary in algorithm analysis?

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

16. What are asymptotic notation? Why it is required?
17. What is Big O notation? Give an example
18. What is Big Omega notation? Give an example
19. Define Big Theta notation. Give an example
20. Define Little Oh notation. Give an example
21. What is recurrence relation? Give an example.
22. Prove the following statements.
 - a) $100n + 5 = \Theta(n^2)$
 - b) $n^2 + 5n + 7 = \Theta(n^2)$
 - c) $n^2 + n = O(n^3)$
 - d) $\frac{1}{2} n(n-1) = \Theta(n^2)$
 - e) $5n^2 + 3n + 20 = O(n^2)$
 - f) $\frac{1}{2} n^2 + 3n = \Theta(n^2)$
 - g) $n^3 + 4n^2 = \Omega(n^2)$

23. Algorithm Sum(n)

```
S ← 0  
For i← 1 to n do  
    S ← S + i  
Return S
```

- a) What does this algorithm compute?
- b) What are its basic operations?
- c) How many times is the basic operation executed?
- d) What is the efficiency class of this algorithm?

Long Answers Questions (THREE, FOUR OR FIVE Marks Questions)

1. What is an Algorithm? Explain the various criteria for writing an algorithms with example?
2. Explain Euclid Algorithm with example to find the GCD of two numbers.
3. Explain Consecutive integer checking methods of find the GCD of two numbers.
4. Explain Algorithm design and analysis process with flow diagram.
5. Explain any FIVE Problem types.
6. Explain following
 - a. Graph problem
 - b. Combinatorial problems
 - c. Geometrical problems.
7. Explain the fundamentals of data structure.
8. Write a note on Graph data structure.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

9. Write a note on following data structures.
 - a. Tree
 - b. Sets
 - c. Dictionary
10. Explain Space complexity and Time complexity with example.
11. Write an algorithm find sum of two matrixes also calculate its time complexity.
12. Write an algorithm find the sum of n numbers also calculate its space and time complexity.
13. Explain the following w.r.t algorithm efficiency.
 - a. Measuring input size
 - b. Unit for measuring the run time
 - c. Order growth
14. Explain Worst case, Best case and average case with example
15. Write an algorithm to perform sequential search and also calculate its Worst case, Best case and average case complexity.
16. Explain Big O notation with example.
17. Explain Big Omega notation with example.
18. Explain Big Theta notation with example.
19. Explain asymptotic notations Big O, Big Ω and Big θ that are used to compare the order of growth of an algorithm with example.
20. Define Big O notation and prove
 - a) $100n + 5 = O(n^2)$
 - b) $5n^2 + 3n + 20 = O(n^2)$
 - c) $n^2+n = O(n^3)$
 - d) $3n+2=O(n)$
 - e) $1000n^2+100n-6=O(n^2)$
16. Define Big Omega notation and prove
 - a) $n^3 \in \Omega(n^2)$
 - b) Prove that $2n + 3 = \Omega(n)$

c) $\frac{1}{2}n(n - 1) \in \Omega(n^2)$.

d) Prove that $n^3 + 4 n^2 = \Omega(n^2)$
17. Define Big Theta notation and prove
 - a) $n^2+5n+7 = \Theta(n^2)$
 - b) $1/2n^2 + 3n = \Theta(n^2)$

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-I

c) $\frac{1}{2}n(n - 1) \in \Theta(n^2)$.

18. Explain with example mathematical analysis of non-recursive algorithm.
19. Write an algorithm to Find the largest element in an array and also perform mathematical analysis.
20. Write an algorithm to Checking for Unique elements in an array and also perform mathematical analysis.
21. Write an algorithm to perform matrix multiplication and also perform mathematical analysis.
22. Write a non-recursive algorithm to Count the number of bits in a number. And also perform mathematical analysis.
23. List the steps for analyzing the time efficiency of recursive algorithm.
23. Explain with example mathematical analysis of recursive algorithm.
24. Write an algorithm to find the factorial of a number using recursion and also perform mathematical analysis.
25. Write an algorithm to perform Towers of Hanoi using recursion and also perform mathematical analysis.
26. State the recursive algorithm to count the **bits of a decimal number** in its binary representation. Give its mathematical analysis.
27. Consider the following algorithm.

```
Algorithm GUESS (A[ ] [ ])  
for i ← 0 to n – 1  
    for j ← 0 to i  
        A [i] [j] ← 0
```

- i) What does the algorithm compute?
- ii) What is its basic operation?
- iii) What is the efficiency of this algorithm?

27. Solve the following recurrence relation.
 - a) $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$
 - b) $x(n) = 3x(n-1)$ for $n > 1$ $x(1)=4$
 - c). $x(n)=x(n-1)+n$ for $n>1$ $x(0)=0$
28. Find the time complexity of below algorithm.

BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA - DAA - UNIT-I

```
algorithm Fibonacci(n)
{
    if n <= 1 then
        output 'n'
    else
        f2 = 0;
        f1 = 1;

        for i = 2 to n do
        {
            f = f1 + f2;
            f2 = f1;
            f1 = f;
        }
        output 'f'
}
```

