Data Structures and Algorithms (CSE2002)

Dr. PARAS JAIN
Senior Assistant Professor (Grade-2)
School of Computing Science and Engineering
VIT Bhopal University

CSE2002   DATA STRUCTURES AND ALGORITHMS   CT – LTP   C – 4

OBJECTIVES:
- To understand various types of fundamental data structures (standard and user defined).
- To learn about algorithm analysis for the run time complexities and the space requirements.
- To acquire knowledge of data structures and algorithms for implementing various real-world problems.

UNIT I Introduction to Algorithm and Data Structures   6
Algorithm: Introduction - Algorithm Design – Complexity- Asymptotic notations.
Data Structures: Introduction- Classification of Data structure - Abstract Data Type (ADT).
UNIT II Sorting and Searching   8
Brute force approach: General method -Sorting (bubble, selection, insertion) –
Searching (Sequential/Linear)
Divide and Conquer approach: General method - Sorting (merge, quick) –
Searching (Binary Search).
UNIT III List, Statck and Queue ADT   10
Linked List: Array Vs Linked List - Singly Linked List, Doubly Linked Lists –
Circular Linked Lists-implementation - application.
Stack and Queue: Introduction – implementation (static and dynamic) – application –
Circular queues-application.
UNIT IV Trees and Hashing   10
Linear Vs Non-Linear Data Structures -General Tree – Terminologies -Binary
Tree – Expression Tree - Traversals - Binary Search Tree – AVL Tree – Red
block Tree – Splay Tree – B Tree. - Hashing: Introduction – Hash Function-
Methods-Collision Resolution.
UNIT V Graph ADT   9
Graph: Introduction – Representations – Traversals - Topological Sorting –
Connected and Bi-Connected Components – Articulation Point - Shortestpath
algorithms (Dijkstra's and Floyd's algorithms) - Minimum spanning tree
(Prim's and Kruskal's algorithms).

                                                    TOTAL: 43 Hours
OUTCOMES:
The students will be able to:
- Apply the fundamental knowledge of various data structures and algorithms to analyze, design, formulate and implement algorithm for any real time problem.
- Apply current techniques in data structures and algorithmic principles for modeling and developing software systems.
- Choose an appropriate design paradigm that solves the given problem efficiently along with appropriate data structures.
- Map real-world problems to algorithmic solutions.

i. TEXT BOOK:
Thomas H. Cormen , Charles E. Leiserson , Ronald L. Rivest , Clifford Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

ii. REFERENCES:
1. Mark A. Weiss,Data Structures and Algorithm Analysis in C++, 4th Edition, Pearson, 2014
2. Aaron M. Tenenbaum, Yeedidyah Langsam, Moshe J. Augenstein, 'Data structures using C', Pearson Education, 2010.
3. D. E. Knuth, Art of computer programming, Volume 1: Fundamental algorithms, Addison Wesley, 2011.

iii. ONLINE RESOURCES:
1. http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html
2. http://nptel.ac.in/courses/106101060/
3. https://www.coursera.org/course/algo

UNIT I INTRODUCTION

1.1 NOTION OF AN ALGORITHM

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
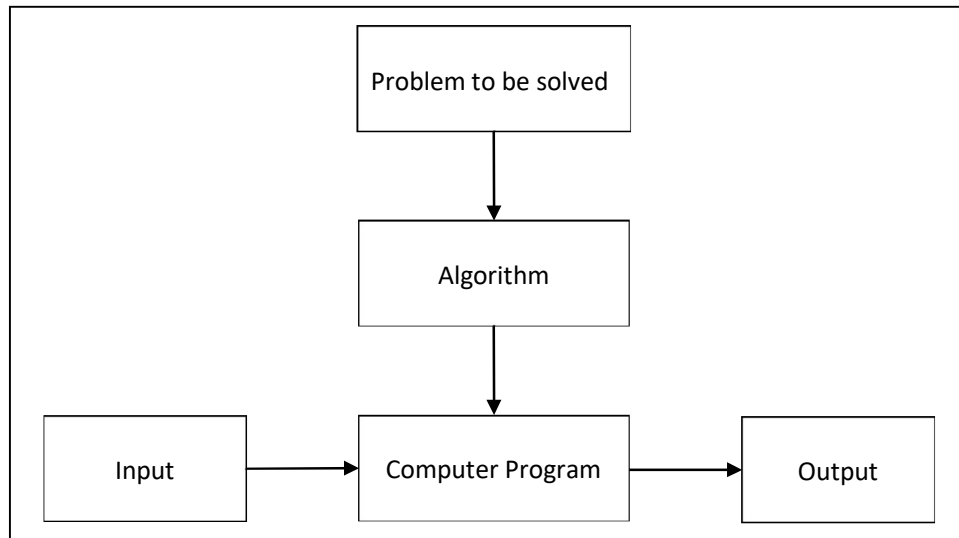

FIGURE 1.1 The notion of the algorithm.

It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

The notion of the algorithm illustrates some important points:
- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Characteristics of an algorithm:

Input: Zero / more quantities are externally supplied**.**

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous.

Finiteness: If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.

Efficiency: Every instruction must be very basic and runs in short time.

Steps for writing an algorithm:
1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is **body**.
2. The Head section consists of keyword **Algorithm** and Name of the algorithm with parameter list. E.g. Algorithm name1(p1, p2,…,p3)

   The head section also has the following:

   > **//Problem Description:**
   > **//Input:**
   > **//Output:**
3. In the body of an algorithm various programming constructs like **if, for, while** and some statements like assignments are used.
4. The compound statements may be enclosed with **{** and **}** brackets. **if**, **for**, **while** can be closed by **end if**, **end for**, **end while** respectively. Proper indention is must for block.
5. Comments are written using // at the beginning.
6. The **identifier** should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
7. The left arrow "**←**" used as assignment operator. E.g. v←10
8. **Boolean** operators (TRUE, FALSE), **Logical** operators (AND, OR, NOT) and **Relational** operators ($<$,$<=$, $>$, $>=$,$=$, $\neq$, $<>$) are also used.
9. Input and Output can be done using **read** and **write**.
10. **Array[]**, **if then else condition**, **branch** and **loop** can be also used in algorithm.

Example:

   The greatest common divisor(GCD) of two nonnegative integers m and n (not-both-zero), denoted gcd*(m, n)*, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality gcd*(m, n)* = gcd*(n, m* mod *n)*, where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd*(m, 0)* = *m*, the last value of m is also the greatest common divisor of the initial m and n.

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)* in simple steps

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.

Euclid's algorithm for computing gcd*(m, n)* expressed in pseudocode

ALGORITHM *Euclid_gcd(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** $n \neq 0$ **do**
> > $r \leftarrow m$ mod n
> > $m \leftarrow n$
> > $n \leftarrow r$
>
> **return** m

1.2 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

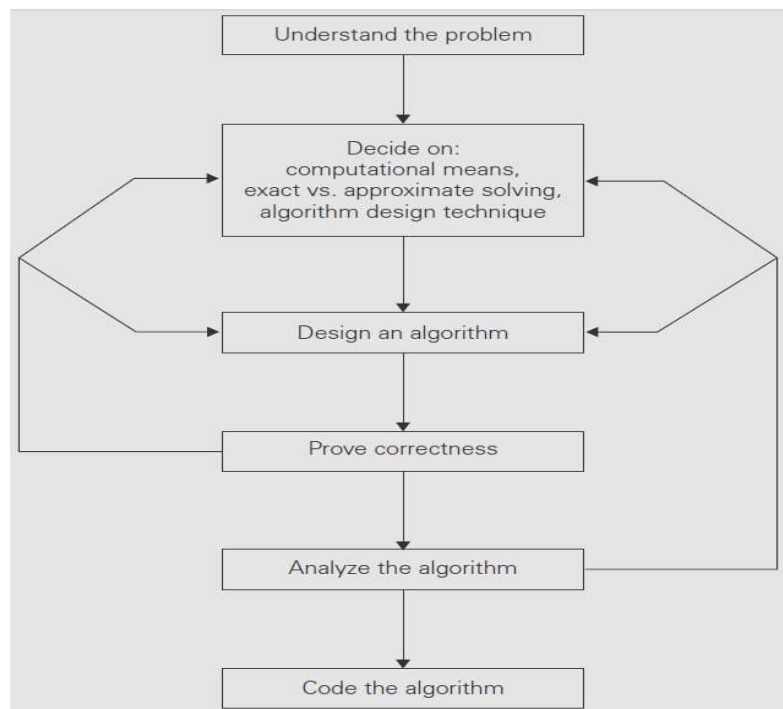A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

FIGURE 1.2 Algorithm design and analysis process.

(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (*instance*) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device
- In *random-access machine* (*RAM*), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called *sequential algorithms*.
- In some newer computers, operations are executed **concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called *parallel algorithms*.
- Choice of computational devices like Processor and memory is mainly based on **space and time efficiency**

(b) Choosing between Exact and Approximate Problem Solving
- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an **exact algorithm**.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an **approximation algorithm**. i.e., produces an

approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

- 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.
- **Algorithms+** *Data Structures = Programs*
- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- **Implementation** of algorithm is possible only with the help of Algorithms and Data Structures
- **Algorithmic strategy / technique / paradigm** are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are**:**

     a. **Natural language**
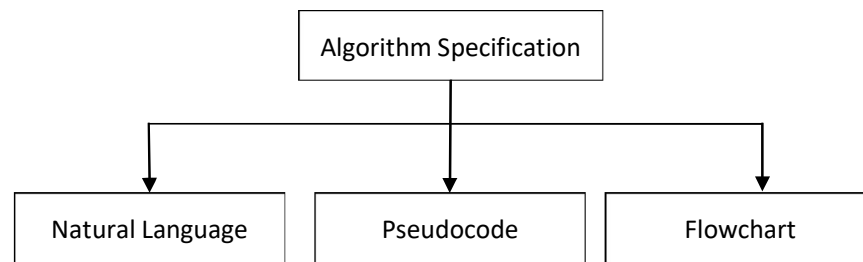   b. **Pseudocode**
     c. **Flowchart**

FIGURE 1.3 Algorithm Specifications

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

Step 1: Read the first number, say a.
Step 2: Read the first number, say b.
Step 3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b. Pseudocode

- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.
- For Assignment operation left arrow "←", for comments two slashes "//",**if** condition**, for, while** loops are used.

> **ALGORITHM** *Sum(a,b)*
> //Problem Description: This algorithm performs addition of two numbers
> //Input: Two integers a and b
> //Output: Addition of two integers
> c←a+b
> return c

This specification is more useful for implementation of any language.

c. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a *flowchart*, this representation technique has proved to be inconvenient.

Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
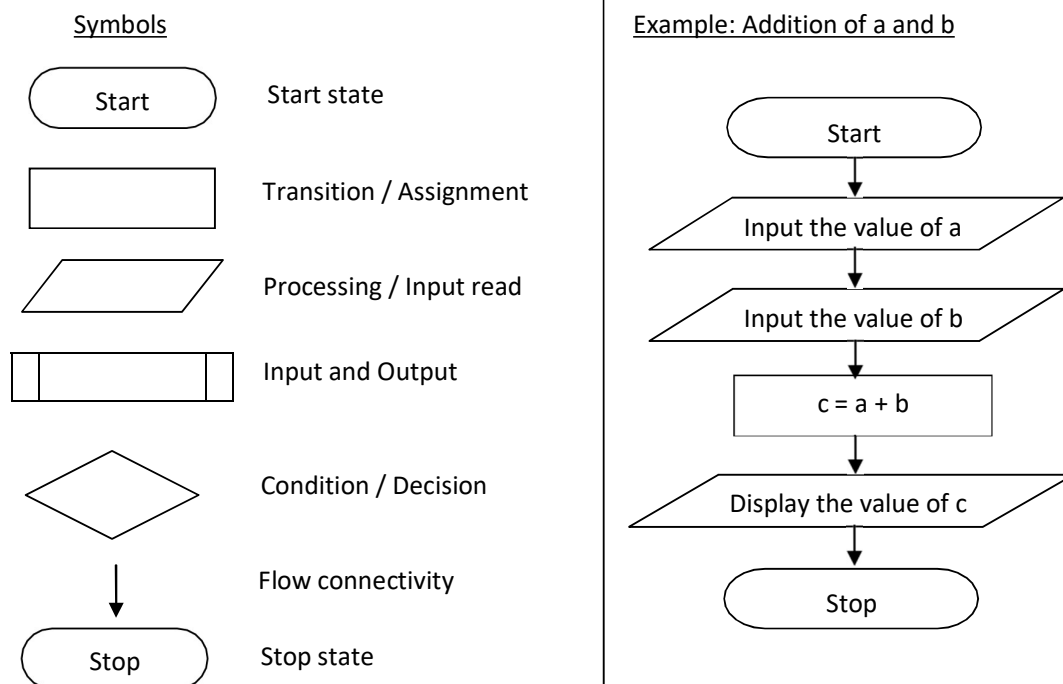

FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its *correctness* must be proved.
- An algorithm must yields a required **result** for every legitimate input in a finite amount of time.

- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)*.
- A common technique for proving correctness is to use **mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

- For an algorithm the most important is *efficiency*. In fact, there are two kinds of algorithm efficiency. They are:
- *Time efficiency*, indicating how fast the algorithm runs, and
- *Space efficiency*, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
- So factors to analyze an algorithm are:
  - Time efficiency of an algorithm
  - Space efficiency of an algorithm
  - Simplicity of an algorithm
  - Generality of an algorithm

(vi) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduced by inefficient implementation.
- Standard tricks like computing a **loop's invariant** (an expression that does not change its value) outside the loop, collecting **common subexpressions**, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by **orders of magnitude**. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an **optimized code (efficient code)** to reduce the burden of compiler.

1.3 IMPORTANT PROBLEM TYPES

The most important problem types are:

- (i).   Sorting
- (ii).   Searching
- (iii). String processing
- (iv).   Graph problems
- (v).   Combinatorial problems
- (vi).   Geometric problems
- (vii). Numerical problems

(i) Sorting

- The *sorting problem* is to rearrange the items of a given list in non decreasing (ascending) order.
- Sorting can be done on numbers, characters, strings or records.
- To sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a *key*.
- An algorithm is said to be **in-place** if it does not require extra memory, E.g., Quick sort.
- A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input.

(ii) Searching

- The *searching problem* deals with finding a given value, called a *search key*, in a given set.
- E.g., Ordinary Linear search and fast binary search.

(iii) String processing

- A *string* is a sequence of characters from an alphabet.
- Strings comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}. It is very useful in bioinformatics.
- Searching for a given word in a text is called string matching.

(iv) Graph problems

- A *graph* is a collection of points called vertices, some of which are connected by line segments called edges.
- Some of the graph problems are graph traversal, shortest path algorithm, topological sort, traveling salesman problem and the graph-coloring problem and so on.

(v) 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.
- A desired combinatorial object may also be required to have some additional property such s a maximum value or a minimum cost.
- In practical, the combinatorial problems are the most difficult problems in computing.
- The traveling salesman problem and the graph coloring problem are examples of *combinatorial problems*.

(vi) Geometric problems

- *Geometric algorithms* deal with geometric objects such as points, lines, and polygons.
- Geometric algorithms are used in computer graphics, robotics, and tomography.
- The *closest-pair problem and t*he *convex-hull problem are comes under this category.*

(vii) Numerical problems

- *Numerical problems* are problems that involve mathematical equations, systems of equations, computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problems can be solved only approximately.

1.4 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

        a. Analysis Framework.
        b. Asymptotic Notations and its properties.
        c. Mathematical analysis for Recursive algorithms.
        d. Mathematical analysis for Non-recursive algorithms.

1.5 Analysis Framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

- ***Time efficiency***, indicating how fast the algorithm runs, and
- ***Space efficiency***, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

(i) Measuring an Input's Size

- An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching.
- For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- In computing the product of two $n \times n$ matrices, the choice of a parameter indicating an input size does matter.
- Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
- The input size by the number b of bits in the n's binary representation is b=($\log_2$ n)+1.

(ii) Units for Measuring Running Time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

    Drawbacks

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

So, we need metric to measure an *algorithm*'s efficiency that does 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 excessively difficult.

The most important operation (+, -, *, /) of the algorithm, called the ***basic operation***. Computing the number of times the basic operation is executed is easy. The total running time is determined by basic operations count.

Orders of Growth
- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
- For large values of *n*, it is the function's order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms.

TABLE 1.1 Values (approximate) of several functions important for analysis of algorithms

n	$\sqrt{n}$	*log₂n*	*n*	*n log₂n*	n^2	n^3	2^n	*n!*
1	1	0	1	0	1	1	2	1
2	1.4	1	2	2	4	8	4	2
4	2	2	4	8	16	64	16	24
8	2.8	3	8	$2.4 \cdot 10^1$	64	$5.1 \cdot 10^2$	$2.6 \cdot 10^2$	$4.0 \cdot 10^4$
10	3.2	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
16	4	4	16	$6.4 \cdot 10^1$	$2.6 \cdot 10^2$	$4.1 \cdot 10^3$	$6.5 \cdot 10^4$	$2.1 \cdot 10^{13}$
10^2	10	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	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

(iv) **Worst-Case, Best-Case, and Average-Case Efficiencies**
Consider Sequential Search algorithm some search key *K*
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 in *A* that matches *K* or -1 if there are no
        //   matching elements
        i ←0
        while *i* < *n* **and** *A*[*i*] ≠ *K* **do**
                i ←*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, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

Worst-case efficiency
- The *worst-case efficiency* of an algorithm is its efficiency for the worst case input of size *n*.
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size *n,* the running time is $C_{worst}(n) = n$.

Best case efficiency
- The ***best-case efficiency*** of an algorithm is its efficiency for the best case input of size n.
- The algorithm runs the fastest among all possible inputs of that size n.
- In sequential search, If we search a first element in list of size n. *(i.e.* first element equal to a search key), then the running time is $C_{best}(n) = 1$

Average case efficiency
- The Average case efficiency lies between best case and worst case.
- To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.
- The standard assumptions are that
   - The probability of a successful search is equal to p $(0 \leq p \leq 1)$ and
   - The probability of the first match occurring in the ith position of the list is the same for every i.

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$
$$= \frac{p}{n} \frac{n(n + 1)}{2} + n(1 - p) = \frac{p(n + 1)}{2} + n(1 - p).$$

Yet another type of efficiency is called ***amortized efficiency***. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.

1.6 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program.

The efficiency analysis framework 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, they are:
- O - Big oh notation
- Ω - Big omega notation
- Θ - Big theta notation

Let *t(n)* and *g(n)* can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time *t(n)* usually indicated by its basic operation count *C(n),* and *g(n),* some simple function to compare with the count.

Example 1:

$$n \in O(n^2), \quad\quad 100n + 5 \in O(n^2), \quad\quad \frac{1}{2}n(n - 1) \in O(n^2).$$

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

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

where $g(n) = n^2$.

(i) O - Big oh notation

A function t*(n)* is said to be in *O(g(n))*, denoted t*(n)* ∈ *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

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

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

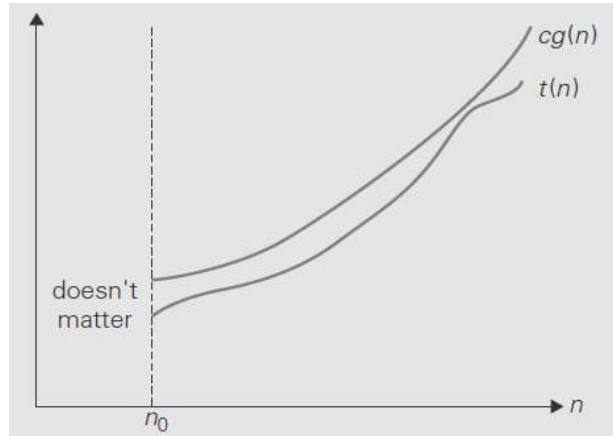O = Asymptotic upper bound = Useful for worst case analysis = Loose bound

FIGURE 1.5 Big-oh notation: t*(n)* ∈ *O(g(n))*.

Example 2:   Prove the assertions $100n^2 + 5 \in O(n^2)$.

We have given $t(n) = 100n^2 + 5$ and $g(n) = n^2$

Proof: $100n^2 + 5 \leq 101n^2$ for all $n \geq 3$

i.e., $t(n) \leq cg(n)$ for all $n \geq n_0$

∴ $100n^2 + 5 \in O(n^2)$ with c=101 and n_0=3

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** and **n_0**. We have c=101 and n_0=3

Example 3:   Prove the assertions $100n + 5 \in O(n)$.

Proof:

$100n + 5 \leq 101n$ for all $n \geq 5$

i.e. $t(n) \leq cg(n)$ for all $n \geq n_0$

∴ $100n + 5 \in O(n)$ with c=101 and n_0=5

(ii) Ω - Big omega notation

A function *t(n)* is said to be in *Ω(g(n))*, denoted *t(n)* ∈ *Ω(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

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

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

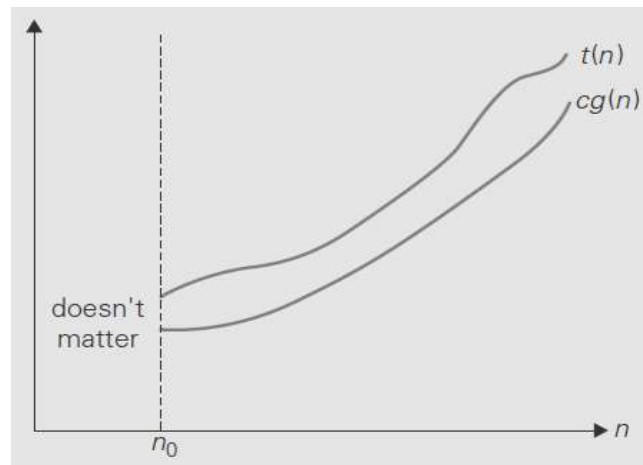Ω = Asymptotic lower bound = Useful for best case analysis = Loose bound

FIGURE 1.6 Big-omega notation: $t(n) \in \Omega(g(n))$.

Example 4: Prove the assertions $n^3+10n^2+4n+2 \in \Omega(n^2)$.
Proof: $n^3+10n^2+4n+2 \geq n^2$ (for all $n \geq 0$)
i.e., by definition $t(n) \geq cg(n)$, where $c=1$ and $n_0=0$

(iii) Θ - Big 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 some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

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

Where $t(n)$ and $g(n)$ are nonnegative functions defined on the set of natural numbers.

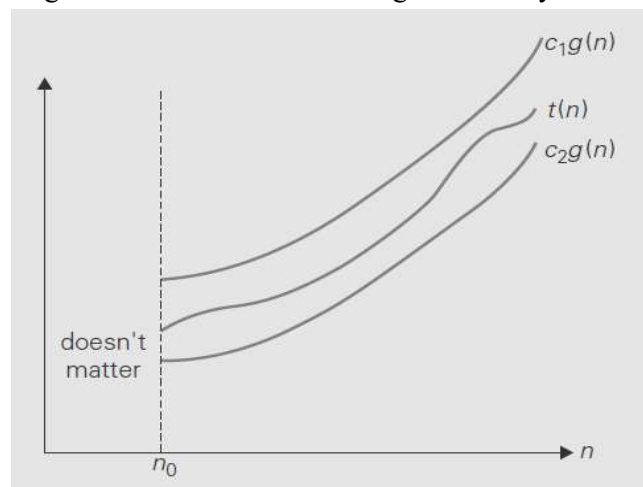Θ = Asymptotic tight bound = Useful for average case analysis

FIGURE 1.7 Big-theta notation: $t(n) \in \Theta(g(n))$.

Example 5: Prove the assertions $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Proof: First prove the right inequality (the upper bound):

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

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

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \left[\frac{1}{2}n\right]\left[\frac{1}{2}n\right] \text{ for all } n \geq 2.$$

$$\therefore \quad \frac{1}{2}n(n-1) \geq \frac{1}{4}n^2$$

$$\text{i.e.,} \quad \frac{n^2}{4} \leq \quad \frac{1}{2} n(n-1) \leq \frac{1}{2}n^2$$

Hence $\frac{1}{2}n(n-1) \in \Theta(n^2)$, where $C_2 = \frac{1}{4}$, $C_1 = \frac{1}{2}$ and $n_0 = 2$

Note: asymptotic notation can be thought of as "relational operators" for functions similar to the corresponding relational operators for values.

$= \Rightarrow \Theta()$, $\qquad \leq \Rightarrow O()$, $\qquad \geq \Rightarrow \Omega()$, $\qquad < \Rightarrow o()$, $\qquad > \Rightarrow \omega()$

Useful Property Involving the Asymptotic Notations

The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$. (The analogous assertions are true for the Ω and Θ notations as well.)

PROOF: The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) \quad &\leq \; c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &= c_3[g_1(n) + g_2(n)] \\ &\leq c_3 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the definition O being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

The property implies that the algorithm's overall efficiency will be determined by the part with a higher order of growth, i.e., its least efficient part.

ö $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

Basic rules of sum manipulation

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

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

Summation formulas

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

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

1.7 MATHEMATICAL ANALYSIS FOR RECURSIVE ALGORITHMS
General Plan for Analyzing the Time Efficiency of Recursive Algorithms
1. Decide on a *parameter* (or 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* 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•. . . . • (n − 1) • n = (n − 1)! • n*, for n ≥ 1 and 0!= 1 by definition, we can compute
F(n) = F(n − 1) • 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*

Algorithm analysis
- For simplicity, we consider *n* itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n)*. Since the function *F(n)* is computed according to the formula *F(n) = F(n −1)•n* for *n >* 0.
- The number of multiplications *M(n)* needed to compute it must satisfy the equality

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

To compute To multiply
F(n-1)   F(n-1) by n

M(n − 1) multiplications are spent to compute *F(n − 1),* and one more multiplication is needed to multiply the result by *n*.

Recurrence relations
The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solve the recurrence relation M(n) = M(n − 1) + 1, i.e., to find an explicit formula for *M(n)* in terms of *n* only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if *n* = 0 **return** 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.

$$M(0) = 0.$$

the calls stop when $n = 0$ ——————   —————— no multiplications when $n = 0$
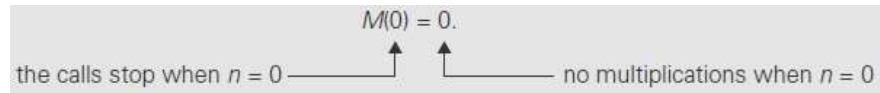
Thus, the recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

M(n) = M(n − 1) + 1 for n > 0,
M(0) = 0   for n = 0.

Method of backward substitutions

M(n) = *M(n − 1) + 1*	substitute *M(n − 1) = M(n − 2) + 1*
= [*M(n − 2) + 1*]+ 1	
= *M(n − 2) + 2*	substitute *M(n − 2) = M(n − 3) + 1*
= [*M(n − 3) + 1*]+ 2	
= *M(n − 3) + 3*	
…	
= M(n − i) + i	
…	
= M(n − n) + n	
= n.	

Therefore *M(n)=n*

EXAMPLE 2: consider educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). 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.
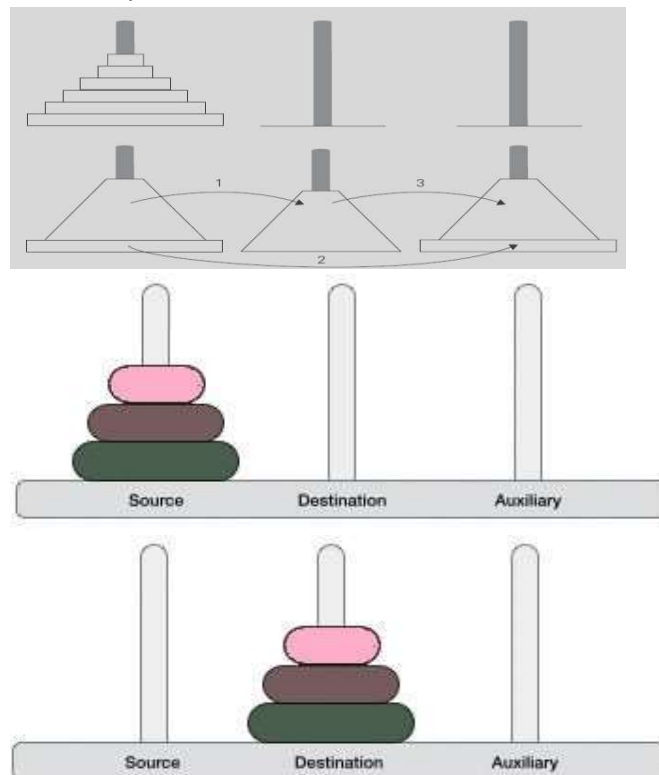

FIGURE 1.8 Recursive solution to the Tower of Hanoi puzzle.

ALGORITHM TOH(n, A, C, B)

      //Move disks from source to destination recursively

      //Input: *n disks and* 3 pegs A, B, and C

      //Output: Disks moved to destination as in the source order.

      if n=1

            Move disk from A to C

      else

        //Move top n-1 disks from A to B using C

          TOH(n - 1, A, B, C)

          Move top n-1 disks from B to C using A

          TOH(n - 1, B, C, A)

Algorithm analysis

      The number of moves *M(n)* depends on *n* only, and we get the following recurrence equation for it: $M(n) = M(n − 1) + 1 + M(n − 1)$ for $n > 1$.

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves *M(n)*:

$$M(n) = 2M(n − 1) + 1 \text{ for } n > 1,$$
$$M(1) = 1.$$

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

$M(n) = 2M(n − 1) + 1$	sub. $M(n − 1) = 2M(n − 2) + 1$
$= 2[2M(n − 2) + 1] + 1$	
$= 2^2 M(n − 2) + 2 + 1$	sub. $M(n − 2) = 2M(n − 3) + 1$
$= 2^2[2M(n − 3) + 1] + 2 + 1$	
$= 2^3 M(n − 3) + 2^2 + 2 + 1$	sub. $M(n − 3) = 2M(n − 4) + 1$
$= 2^4 M(n − 4) + 2^3 + 2^2 + 2 + 1$	

…

$= 2^i M(n − i) + 2^{i−1} + 2i^{−2} + . . . + 2 + 1 = 2^i M(n − i) + 2^i − 1.$

…

    Since the initial condition is specified for $n = 1$, which is achieved for $i = n − 1$,

$M(n) = 2^{n−1} M(n − (n − 1)) + 2^{n−1} − 1 = 2^{n−1} M(1) + 2^{n−1} − 1 = 2^{n−1} + 2^{n−1} − 1 = 2^n − 1.$

Thus, we have an exponential time algorithm

EXAMPLE 3: An investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

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 $BinRec(\lfloor n/2 \rfloor) + 1$

Algorithm analysis

      The number of additions made in computing $BinRec\lfloor n/2 \rfloor$) is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$.

      Since the recursive calls end when *n* is equal to 1 and there are no additions made

then, the initial condition is $A(1) = 0$.

The standard approach to solving such a recurrence is to solve it only for $n = 2^k$

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

backward substitutions

$A(2^k) = A(2^{k-1}) + 1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$ . . .

. . .

$= A(2^{k-i}) + i$

. . .

$= A(2^{k-k}) + k$.

Thus, we end up with $A(2^k) = A(1) + k = k$, or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

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

1.8 MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS
General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation* (in the innermost loop).
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 least, establish its *order of growth*.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

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
                        maxval←A[i]
        return maxval

Algorithm analysis

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body:
    o The comparison A[i]> maxval and
    o The assignment maxval←A[i].

- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. 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, inclusive. Therefore, the sum for C(n) is calculated as follows:

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

i.e., Sum up 1 in repeated n-1 times

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

EXAMPLE 2: Consider the **element uniqueness problem**: check whether all the Elements in a given array of n elements 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**

   **if** A[i]= A[j] **return false**

    return true

Algorithm analysis

- The natural measure of the input's size here is again n (the number of elements in the array).
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

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

-

EXAMPLE 3: Consider matrix multiplication. Given two n × 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 × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:
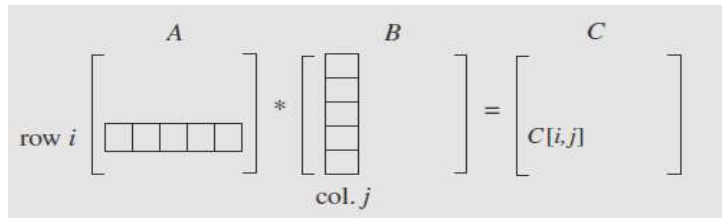
where C[i, j]= A[i, 0]B[0, j]+ . . . + A[i, k]B[k, j]+ . . . + A[i, n − 1]B[n − 1, j] for every pair of indices $0 \leq i, j \leq n - 1$.

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 × n matrices A and B

    //Output: Matrix C = AB

    for i ←0 **to** n − 1 **do**

        for j ←0 **to** n − 1 **do**

            C[i, j]←0.0

            for k←0 **to** n − 1 **do**

                C[i, j]←C[i, j]+ A[i, k] ∗ B[k, j]

    return C

Algorithm analysis

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- 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

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

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)

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

The running time of the algorithm on a particular machine m, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

If we consider, time spent on the additions too, then the total time on the machine is

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3$$

EXAMPLE 4 The following algorithm finds the number of binary digits in the **binary representation** of a positive decimal integer.

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← $\lfloor n/2 \rfloor$

       return count

Algorithm analysis

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- 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 $\lfloor \log2\ n \rfloor + 1$.