

# Lecture Notes on

## Design and Analysis of Algorithms

### 18CS42

#### Module-1: Introduction to Algorithm

##### Contents

1. Introduction
  - 1.1. What is an Algorithm?
  - 1.2. Algorithm Specification
  - 1.3. Analysis Framework
2. Performance Analysis
  - 2.1. Space complexity
  - 2.2. Time complexity
3. Asymptotic Notations
  - 3.1. Big-Oh notation
  - 3.2. Omega notation
  - 3.3. Theta notation
  - 3.4. Little-oh notation
  - 3.5. Mathematical analysis
4. Important Problem Types
  - 4.1. Sorting
  - 4.2. Searching
  - 4.3. String processing
  - 4.4. Graph Problems
  - 4.5. Combinatorial Problems
5. Fundamental Data Structures
  - 5.1. Linear Data Structures
  - 5.2. Graphs
  - 5.3. Trees
  - 5.4. Sets and Dictionaries.

## Module-1: Introduction

### 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.

**Input.** Zero or more quantities are externally supplied.

- Output.** At least one quantity is produced.
- Definiteness.** Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
- Finiteness.** If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- 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 criterion c; it also must be feasible.

#### 1.1.2. Algorithm Specification

An algorithm can be specified in

- 1) Simple English
- 2) Graphical representation like flow chart
- 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;  
    }  
}
```

### 1.1.3. Analysis Framework

#### Measuring an Input's Size

It is observed that almost all algorithms **run longer on 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**. The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

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  $n$  is prime). For such algorithms, computer scientists prefer measuring size by the number  $b$  of bits in the  $n$ 's binary representation:  $b = \lceil \log_2 n \rceil + 1$ . This metric usually gives a better idea about the efficiency of algorithms in question.

#### Units for Measuring Running time

To measure an algorithm's efficiency, we would like to have a **metric 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 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  $c_{op}$  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:

$$T(n) \approx c_{op}C(n)$$

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 count's **order of growth** to within a constant multiple for large-size inputs.

## 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.

## 1.2. Performance Analysis

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

- Time efficiency indicates 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 inspace. Therefore, we primarily concentrate on time efficiency.

### 1.2.1 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()**

```
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;  $S_{sum}(n) \geq (n+3); n$  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;
}
```

### 1.2.2 Time complexity

Usually, the execution time or run-time of the program is refereed as its time complexity denoted by  $t_p$ (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 program. So when each time original program executes, the **count** also incremented by the step count.

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

```
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;
}
```

**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			$2mn + 2m + 1$

The above method is both excessively difficult and, 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.

### 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.

## 1.3. Asymptotic Notations

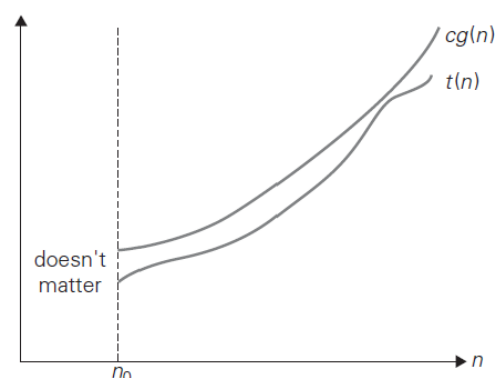
The efficiency analysis framework concentrates on the order of growth of an algorithm's

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:  $O$  (big oh),  $\Omega$  (big omega),  $\Theta$  (big theta) and  $o$  (little oh)

### 1.3.1. 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  $n_0$ .

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

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

Strategies to prove Big-O: Sometimes the easiest way to prove that  $f(n) = O(g(n))$  is to take  $c$  to be the sum of the positive coefficients of  $f(n)$ . We can usually ignore the negative coefficients.

**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  $100n + 5 \in O(n^2)$

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

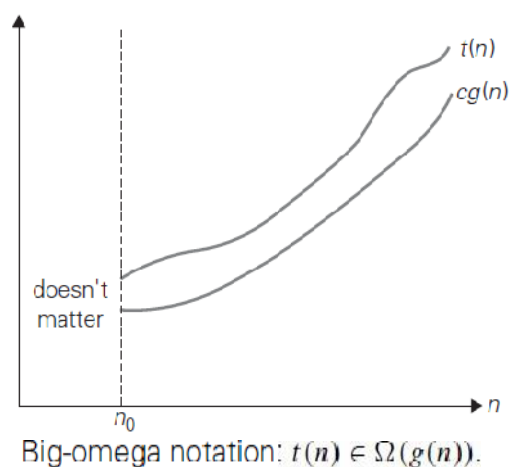
**Example:** To prove  $n^2 + n = O(n^3)$

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

i) Prove  $3n+2=O(n)$  ii) Prove  $1000n^2+100n-6 = O(n^2)$

### 1.3.2. Omega notation

**Definition:** 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  $n^3 \in \Omega(n^2)$ :  $n^3 \geq n^2$  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:** To prove  $n^3 + 4n^2 = \Omega(n^2)$

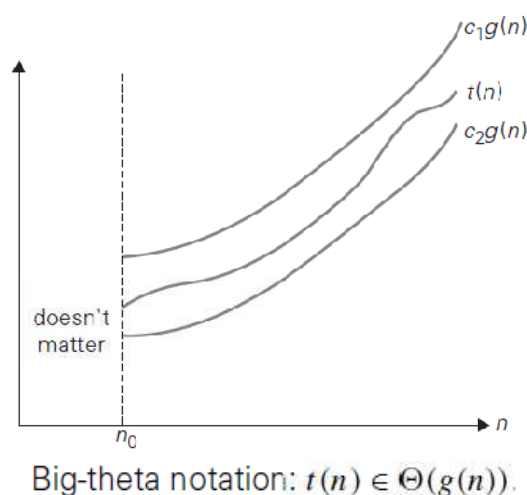
We see that, if  $n \geq 0$ ,  $n^3 + 4n^2 \geq n^3 \geq n^2$ . Therefore  $n^3 + 4n^2 \geq 1n^2$  for all  $n \geq 0$

Thus, we have shown that  $n^3 + 4n^2 = \Omega(n^2)$  where  $c = 1$  &  $n_0 = 0$

### 1.3.3. 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 some positive constants  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that

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





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  $n_0 = 2$ .

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

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- $\Theta$ , with  $n_0 = 1$ ,  $c_1 = 1$ , and  $c_2 = 13$ .)

### Strategies for $\Omega$ and $\Theta$

- Proving that a  $f(n) = \Omega(g(n))$  often requires more thought.
  - Quite often, we have to pick  $c < 1$ .
  - A good strategy is to pick a value of  $c$  which you think will work, and determine which value of  $n_0$  is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms of  $f(n)$  with the positive coefficients.
- The following theorem shows us that proving  $f(n) = \Theta(g(n))$  is nothing new:

Theorem:  $f(n) = \Theta(g(n))$  if and only iff  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .

Thus, we just apply the previous two strategies.

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

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

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

**Show that**  $\frac{1}{5}n^2 - 3n = \Theta(n^2)$

We need to find positive constants  $c_1$ ,  $c_2$ , and  $n_0$  such that

$$0 \leq c_1 n^2 \leq \frac{1}{5}n^2 - 3n \leq c_2 n^2 \text{ for all } n \geq n_0$$

Dividing by  $n^2$ , we get

$$0 \leq c_1 \leq \frac{1}{5} - \frac{3}{n} \leq c_2$$

$c_1 \leq \frac{1}{5} - \frac{3}{n}$  holds for  $n \geq 10$  and  $c_1 = 1/5$

$\frac{1}{5} - \frac{3}{n} \leq c_2$  holds for  $n \geq 10$  and  $c_2 = 1$ .

Thus, if  $c_1 = 1/5$ ,  $c_2 = 1$ , and  $n_0 = 10$ , then for all  $n \geq n_0$ ,

$$0 \leq c_1 n^2 \leq \frac{1}{5}n^2 - 3n \leq c_2 n^2 \text{ for all } n \geq n_0.$$

Thus we have shown that  $\frac{1}{5}n^2 - 3n = \Theta(n^2)$ .

**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  $\Omega$  and  $\Theta$  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  $t_1(n) \leq c_1 g_1(n)$  for all  $n \geq n_1$ .

Similarly, since  $t_2(n) \in O(g_2(n))$ ,  $t_2(n) \leq c_2 g_2(n)$  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:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3 [g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

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  $O$  definition being  $2c_3 = 2 \max\{c_1, c_2\}$  and  $\max\{n_1, n_2\}$ , respectively.

**3.4. 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

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

Example: The function  $3n + 2 = o(n^2)$  since  $\lim_{n \rightarrow \infty} \frac{3n+2}{n^2} = 0$ .  $3n + 2 = o(n \log n)$ .  $3n + 2 = o(n \log \log n)$ .  $6 * 2^n + n^2 = o(3^n)$ .  $6 * 2^n + n^2 = o(2^n \log n)$ .  $3n + 2 \neq o(n)$ .  $6 * 2^n + n^2 \neq o(2^n)$ .  $\square$

For comparing the order of growth limit is used

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

If the case-1 holds good in the above limit, we represent it by little-oh.

**EXAMPLE 1** Compare the orders of growth of  $\frac{1}{2}n(n-1)$  and  $n^2$ . (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically,  $\frac{1}{2}n(n-1) \in \Theta(n^2)$ . ■

**EXAMPLE 2** Compare the orders of growth of  $\log_2 n$  and  $\sqrt{n}$ . (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero,  $\log_2 n$  has a smaller order of growth than  $\sqrt{n}$ . (Since  $\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = 0$ , we can use the so-called *little-oh notation*:  $\log_2 n \in o(\sqrt{n})$ . Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.) ■

### 1.3.5. Basic asymptotic efficiency Classes

Class	Name	Comments
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.

### 1.3.6. Mathematical Analysis of Non-recursive & Recursive Algorithms

#### Analysis of 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. (As a rule, it is located in 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 closedform

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  $A$

$maxval \leftarrow A[0]$

**for**  $i \leftarrow 1$  **to**  $n - 1$  **do**

**if**  $A[i] > maxval$

$maxval \leftarrow A[i]$

**return**  $maxval$

Here comparison is the basic operation. Note that number of comparisons will be same for all arrays of size  $n$ . Therefore, no need to distinguish worst, best and average cases. Total number of basic operations are,

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

**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**  $i \leftarrow 0$  **to**  $n - 2$  **do**

**for**  $j \leftarrow i + 1$  **to**  $n - 1$  **do**

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

**return** true

Here basic operation is comparison. The maximum no. of comparisons happen in the worst

case. (i.e. all the elements in the array are distinct and algorithms return *true*).

Total number of basic operations (comparison) in the worst case are,



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

Other than the worst case, the total comparisons are **less than**  $\frac{1}{2}n^2$ . For example if the first two elements of the array are equal, only one comparison is computed.

So in general  $C(n) = O(n^2)$

### Example-3: To perform matrix multiplication

**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**  
 $C[i, j] \leftarrow 0.0$   
**for**  $k \leftarrow 0$  **to**  $n-1$  **do**  
    $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$   
**return**  $C$

Number of basic operations  
(multiplications) is

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

Total running time:  $T(n) \approx c_m M(n) = c_m n^3$

Suppose if we take into account of addition; Algorithm also have same number of additions  
 $A(n) = n^3$

Total running time:  $T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$

### Example-4: 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  
 $count \leftarrow 1$   
**while**  $n > 1$  **do**  
    $count \leftarrow count + 1$   
    $n \leftarrow \lfloor n/2 \rfloor$   
**return**  $count$

The basic operation is  $\text{count} = \text{count} + 1$  repeats  $\lfloor \log_2 n \rfloor + 1$  no. of times

### Analysis of 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. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
4. 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**  $F(n-1) * n$

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

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

Such equations are called **recurrence relations**

Condition that makes the algorithm stop **if**  $n = 0$  **return** 1. Thus recurrence relation and initial condition for the algorithm's number of multiplications  $M(n)$  can be stated as

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

$$M(0) = 0.$$

We can use backward substitutions method to solve this

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

**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, which is illustrated in Figure.

1. If  $n = 1$ , we move the single disk directly from the source peg to the destination peg.
2. 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).

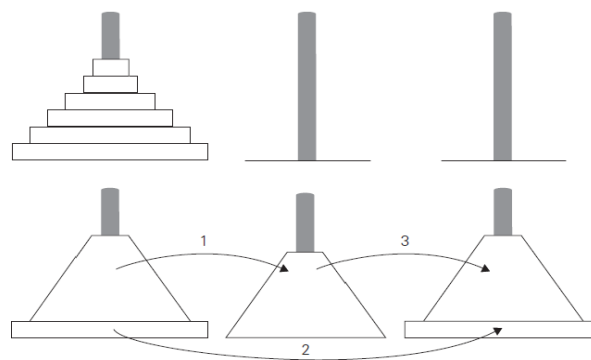


Figure: Recursive solution to the Tower of Hanoi puzzle

```

Algorithm: TowerOfHanoi(n, source, dest, aux)
  If n == 1, THEN
    move disk from source to dest
  else
    TowerOfHanoi (n - 1, source, aux, dest)
    move disk from source to dest
    TowerOfHanoi (n - 1, aux, dest, source)
  End if
  
```

### Computation of 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  $M(n)$ :

$$M(n) = 2M(n - 1) + 1 \quad \text{for } n > 1$$

$$M(1) = 1.$$

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

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

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

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

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

#### 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

The recurrence relation can be written as

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

Also note that  $A(1) = 0$ .

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

$$A(2^0) = 0.$$

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  $n$ .

Now backward substitutions encounter no problems:

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

Thus, we end up with

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

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

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

## 1.4. Important Problem Types

In this section, we are going to introduce the most important problem types: Sorting, Searching, String processing, Graph problems, Combinatorial problems.

### 1.4.1. Sorting

The sorting problem is to rearrange the items of a given list in **non-decreasing order**. As a practical matter, we usually need to sort lists of numbers, characters from an alphabet or character strings. Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations. Some of the algorithms are simple but relatively slow, while others are faster but more complex; some work better on randomly ordered inputs, while others do better on almost-sorted lists; some are suitable only for lists residing in the fast memory, while others can be adapted for sorting large files stored on a disk; and so on.

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.

### 1.4.2. Searching

The searching problem deals with finding a given value, called a search key, in a given set. (or a multiset, which permits several elements to have the same value). 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.

### 1.4.3. String Processing

In recent decades, the rapid proliferation of applications dealing with non-numerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms. A string is a sequence of characters from an alphabet. String-processing

algorithms have been important for computer science in conjunction with computer languages and compiling issues.

### 1.4.4. Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a graph can be thought of as a collection of points called **vertices**, some of which are connected by line segments called **edges**. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games. Studying different technical and social aspects of the Internet in particular is one of the active areas of current research involving computer scientists,

economists, and social scientists.

### 1.4.5. Combinatorial Problems

Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint. Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

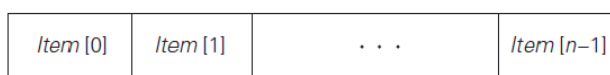
## 1.5. Fundamental Data Structures

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

### 1.5.1. 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 is 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.

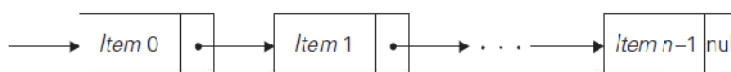


FIGURE 1.4 Singly linked list of  $n$  elements.

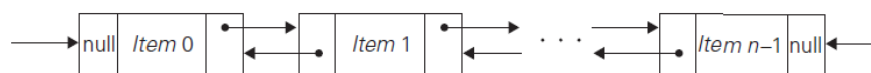


FIGURE 1.5 Doubly linked list of  $n$  elements.

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 not horizontally but vertically—akin to a stack of plates whose “operations” it mimics very closely.

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 a “first-in–first-out” (FIFO) fashion—akin 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.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a **priority queue**. A priority queue is a collection of data items from a totally ordered universe (most often, integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element.

### 1.5.2. Graphs

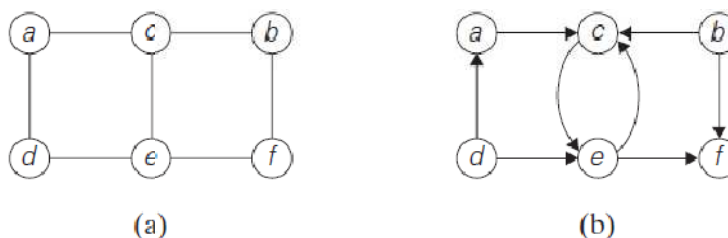
A graph is informally thought of as a collection of points in the plane called “vertices” or nodes,” some of them connected by line segments called “edges” or “arcs.” A graph  $G$  is called **undirected** if every edge in it is undirected. A graph whose every edge is directed is called **directed**. 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*.

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  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column is equal to 1 if there is an edge from the  $i^{\text{th}}$  vertex to the  $j^{\text{th}}$  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).

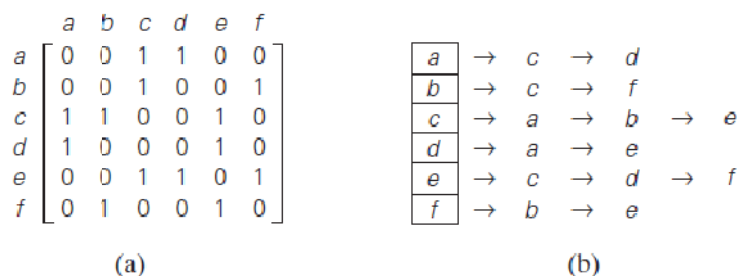


FIGURE 1.5.2 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.5.1

**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.

Among the many 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  $v$ .

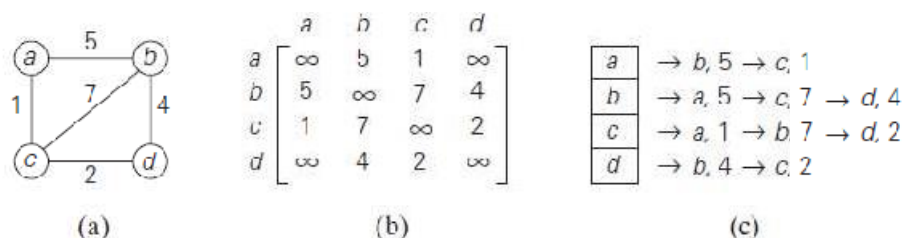
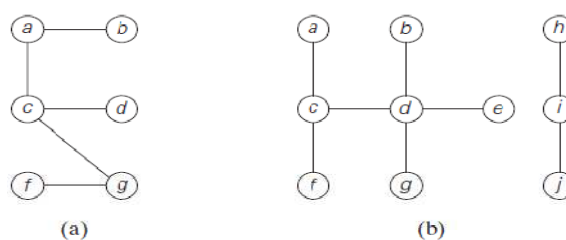


FIGURE 1.5.3 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

A graph is said to be **connected** if for every pair of its vertices  $u$  and  $v$  there is a path from  $u$  to  $v$ . Graphs with several connected components do happen in real-world applications. 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.

### 1.5.3. Trees

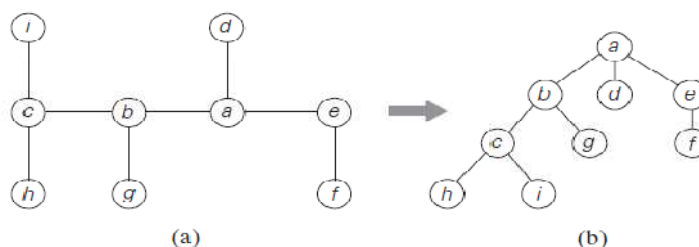
A **tree** (more accurately, a free tree) is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a **forest**: each of its connected components is a tree. Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices:  $|E| = |V| - 1$



(a) Tree. (b) Forest.

**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 roots till below (level 2), and so on.

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.

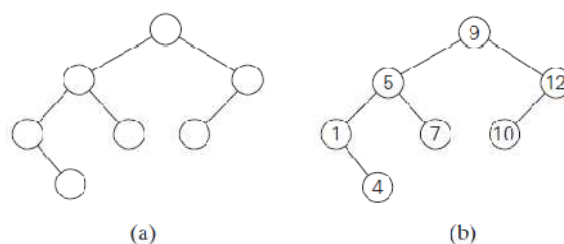


1.11 (a) Free tree. (b) Its transformation into a rooted tree.

**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; a binary tree may also be empty.

If 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.



1.2 (a) Binary tree. (b) Binary search tree.

#### 1.5.4. 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.

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 bitstring of size  $n$ , called a **bit vector**, in which the  $i^{\text{th}}$  element is 1 if and only if the  $i^{\text{th}}$  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 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 set or 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**. An efficient implementation of a dictionary has to strike a compromise between the efficiency

of searching and the efficiencies of the other two operations. They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees.

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