MODULE I

INTRODUCTION

Contents

1. INTRODUCTION

1.1 What is an Algorithm?

Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

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.

This definition can be illustrated by a simple diagram:

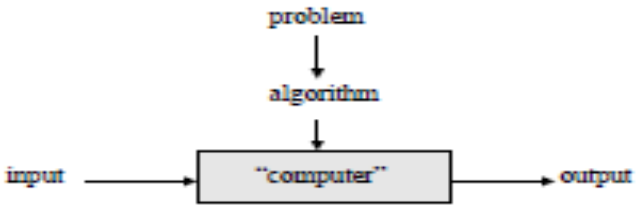


FIGURE: The notion of the algorithm

- The reference to "instructions" in the definition implies that there is something or someone capable of understanding and following the instructions given.
- Human beings and the computers nowadays are electronic devices being involved in performing numeric calculations.
- However, that although the majority of algorithms are indeed intended for eventual computer implementation, the notion of algorithm does not depend on such an assumption.

In addition, all algorithms should satisfy the following criteria or properties.

1. **INPUT** → Zero or more quantities are externally supplied.
2. **OUTPUT** → At least one quantity is produced.
3. **DEFINITENESS** → Each instruction is clear and unambiguous.
4. **FINITENESS** → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **EFFECTIVENESS** → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Fundamentals Of Algorithmic Problem Solving

- Algorithms are procedural solutions to problems. These solutions are not answers but specific instructions for getting answers.
- The following diagram briefly illustrates the sequence of steps one typically goes through in designing and analyzing an algorithm.
- It includes the following sequence of steps:
  - ✓ Understanding the problem
  - ✓ Deciding on: Computational means, Exact vs. approximate problem solving, data structure(s), Algorithm design techniques.
  - ✓ Design an algorithm
  - ✓ Prove correctness
  - ✓ Analyze the algorithm
  - ✓ Code the algorithm

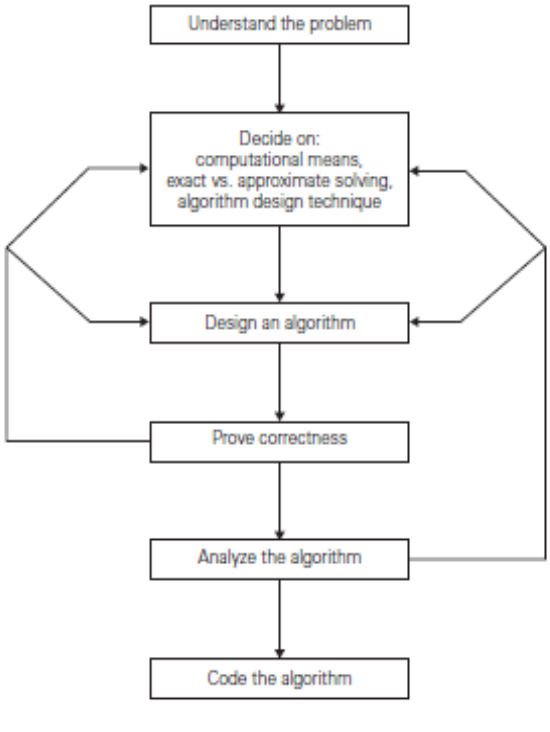


Figure: Algorithm design and analysis process.

- **Understanding the problem**
  - ✓ Before designing an algorithm the most important thing is to understand the problem given.
  - ✓ Asking questions, doing a few examples by hand, thinking about special cases, etc.
  - ✓ An Input to an algorithm specifies an instance of the problem the algorithm that it solves.

- ✓ Important to specify exactly the range of instances the algorithm needs to handle. Else it will work correctly for majority of inputs but crash on some "boundary" value.
- ✓ A correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

- **Ascertaining the capabilities of a computational device**
  - ✓ After understanding need to ascertain the capabilities of the device.
  - ✓ The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture.
  - ✓ Von Neumann architectures are sequential and the algorithms implemented on them are called sequential algorithms.
  - ✓ Algorithms which designed to be executed on parallel computers called parallel algorithms.
  - ✓ For very complex algorithms concentrate on a machine with high speed and more memory where time is critical.
- **Choosing between exact and approximate problem solving**
  - ✓ For exact result->exact algorithm
  - ✓ For approximate result->approximation algorithm.
  - ✓ Examples of exact algorithms: Obtaining square roots for numbers and solving non-linear equations.
  - ✓ An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.
- **Deciding on appropriate data structures**
  - ✓ Algorithms may or may not demand ingenuity in representing their inputs.
  - ✓ Inputs are represented using various data structures.
  - ✓ Algorithm + data structures=program.
- **Algorithm Design Techniques and Methods of Specifying an Algorithm**
  - ✓ 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.
  - ✓ The different algorithm design techniques are: brute force approach, divide and conquer, greedy method, decrease and conquer, dynamic programming, transform and conquer and back tracking.

- ✓ Methods of specifying an algorithm: Using natural language, in this method ambiguity problem will be there.
- ✓ The next 2 options are : pseudo code and flowchart.
- ✓ Pseudo code: mix of natural and programming language. More precise than NL
- ✓ Flow chart: method of expressing an algorithm by collection of connected geometric shapes containing descriptions of the algorithm's steps.
- ✓ This representation technique has proved to be inconvenient for large problems.
- ✓ The state of the art of computing has not yet reached a point where an algorithm's description—be it in a natural language or pseudo code—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language.
- ✓ Hence program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

- **Proving an Algorithm's Correctness**
  - ✓ After specifying an algorithm, we have to prove its correctness.
  - ✓ The correctness is to prove that the algorithm 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)*, the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.
  - ✓ For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.
  - ✓ A common technique for proving correctness is to use mathematical induction.
  - ✓ Proof by mathematical induction is most appropriate for proving the correctness of an algorithm.
- **Analyzing an algorithm**
  - ✓ After correctness, efficiency has to be estimated.
    - ➢ Time efficiency and space efficiency.
    - ➢ Time: how fast the algorithm runs?
    - ➢ Space: how much extra memory the algorithm needs?
  - ✓ Simplicity: how simpler it is compared to existing algorithms. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

- ✓ Generality: Generality of the problem the algorithm solves and the set of inputs it accepts.
- ✓ If not satisfied with these three properties it is necessary to redesign the algorithm.

- **Code the algorithm**
  - ✓ Writing program by using programming language.
  - ✓ Selection of programming language should support the features mentioned in the design phase.
  - ✓ Program testing: If the inputs to algorithms belong to the specified sets then require no verification. But while implementing algorithms as programs to be used in actual applications, it is required to provide such verifications.
  - ✓ Documentation of the algorithm is also important.

1.2 Algorithm Specification

Algorithm can be described in three ways.

1. Natural language like English: When this way is choosed care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.

3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles programming language constructs.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.

2. Bocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

Node. Record { data type – 1 data-1; . . . data type – n data – n; node * link; }

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

- ✓ Logical Operators AND, OR, NOT
- ✓ Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed. For, while and repeat-until

While Loop:

While < condition > do

{ <statement-1> . . . <statement-n> }

For Loop:

For variable: = value-1 to value-2 step step do

{ <statement-1> . . . <statement-n> }

repeat-until:

repeat <statement-1> . . . <statement-n> until<condition>

8. A conditional statement has the following forms.

- ✓ If <condition> then <statement>
- ✓ If <condition> then <statement-1> Else <statement-1>
- ✓ **Case statement:**

Case

{ : <condition-1> : <statement-1>

. . .

: <condition-n> : <statement-n>

: else : <statement-n+1>

}

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:

- ✓ Algorithm, the heading takes the form, Algorithm Name (Parameter lists)
- ✓ As an example, the following algorithm fields & returns the maximum of „n" given numbers:

```
Algorithm Max(A,n)
// A is an array of size n
{
        Result := A[1];
        for i:= 2 to n do
         if A[i] > Result then
```

```
        Result :=A[I];
        return Result;
    }
```

In this algorithm (named Max), A & n are procedure parameters. Result & i are Local variables.

1.3 Analysis Framework

There are two kinds of efficiency:

♦ **Time efficiency** - indicates how fast an algorithm in question runs.

♦ **Space efficiency** - deals with the extra space the algorithm requires.

✓ **Measuring An Input Size**

♦ An algorithm's efficiency is investigated 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, finding the list's smallest element, and most other problems dealing with lists.

♦ For the problem of evaluating a polynomial **p(x)** of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

♦ There are situations, of course, where the choice of a parameter indicating an input size does matter.

- **Example** - computing the product of two n-by-n matrices.There are two natural measures of size for this problem.
- The matrix order n.
- The total number of elements N in the matrices being multiplied.

♦ Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.

♦ 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 size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime).

✓ **Units For Measuring Run Time**

♦ We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.

♦ There are obvious drawbacks to such an approach. They are

- ♦ 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.

♦ Since we are in need to measure algorithm efficiency, we should 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 difficult and unnecessary.

♦ The main objective 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.

♦ As a rule, it is not difficult to identify the basic operation of an algorithm.

For e.g., The basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

ALGORITHM sum_of_numbers (A[0… n-1])
// Functionality : Finds the Sum
// Input : Array of n numbers
// Output : Sum of 'n' numbers
i ← 0
sum ← 0
while i < n
    sum ← sum + A[i] ⟶ n
    i ← i + 1
return sum

♦ Consider the following example:

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

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

Total number of steps for basic operation execution, C (n) = n

✓ **Orders Of Growth**

♦A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.

♦When we have to compute, 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 two algorithms discussed in previous section or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important.

♦ For large values of *n*, it is the function's order of growth that counts: look at Table which contains values of a few functions particularly important for analysis of algorithms.

Table: Values (some approximate) 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}		

✓ **Worst-Case, Best-Case, Average Case Efficiencies**

♦Algorithm efficiency depends on the **input size n.** And for some algorithms efficiency not only on input size but also on the **specifics of particular or type of input.**

♦We have best, worst & average case efficiencies.

Worst-case efficiency: Efficiency (number of times the basic operation will be executed) **for the worst case input of size n.** *i.e.* The algorithm runs the longest among all possible inputs of size n.

Best-case efficiency: Efficiency (number of times the basic operation will be executed) **for the best case input of size n.** *i.e.* The algorithm runs the fastest among all possible inputs of size n.

Average-case efficiency: Average time taken (number of times the basic operation will be executed) **to solve all the possible instances (random) of the input.**

NOTE: It is not the average of worst and best case.

♦ **Example: Sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

♦ Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

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.

Worst case efficiency

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

♦ In the worst case the input might be in such a way that there are no matching elements or the first matching element happens to be the last one on the list, in this case the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$C_{worst}(n) = n$.

♦ The way to determine is quite straightforward

♦ To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$.

♦ The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$ its running time on the worst-case inputs.

Best case Efficiency

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

♦ We can analyze the best case efficiency as follows.

♦ First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)

♦ Then ascertain the value of C (n) on these most convenient inputs.

♦ Example- for sequential search, best-case inputs will be lists of size n with their first element equal to a search key; accordingly, $C_{best}(n) = 1$.

♦ The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency.

♦ But it is not completely useless. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast.

♦ Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Average case efficiency

♦ It yields the information about an algorithm about an algorithm‘s behaviour on a —typical and random input.

♦To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

♦The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.

♦It involves dividing all instances of size n .into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.

♦Then a probability distribution of inputs needs to be obtained or assumed so that the expected value of the basic operation's count can then be derived. The average number of key comparisons $C_{avg}(n)$ can be computed as follows,

♦Let us consider again sequential search. The standard assumptions are,

♦Let **p** be the probability of successful search.

♦ In the case of a successful search, the probability of the first match occurring in the i^{th} position of the list is **p/n** for every **i, where 1<=i<=n** and the number of comparisons made by the algorithm in such a situation is obviously **i**.

♦ In the case of an unsuccessful search, the number of comparisons is **n** with the probability of such a search being **(1 - p).** Therefore,

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

♦ Example, if p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.

♦ If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

Recapitulation of the Analysis Framework

1 Both time and space efficiencies are measured as functions of the algorithm's input size.

2 Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

3 The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.

4 The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

2. PERFORMANCE ANALYSIS

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()**

```
Algorithm abc(a,b,c)
{
   return a+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

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

2.2 Time Complexity

- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

✓ The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .

✓ The run time is denoted by tp(instance characteristics).

- Usually, the execution time or run-time of the program is refered 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 instruction 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**.

✓ A program step is loosely defined as syntactically or semantically meaning segment of the program that has an execution time that is independent of instance characteristics. For example comment has zero steps; assignment statement has one step and in an iterative statement such as the for, while & repeat-until statements, the step count is counted for the control part of the statement

- We can determine the **steps needed by a program** to solve a particular problem instance in two ways

✓ In the **first method** 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-1:** Consider the algorithm ***Sum( )***. After the introduction of the count the program will be as follows.

```
Algorithm Sum(a, n)
{
    s := 0.0;
    count := count + 1; // count is global; it is initially zero.
    for i := 1 to n do
    {
        count := count + 1; // For for
        s := s + a[i]; count := count + 1; // For assignment
    }
    count := count + 1; // For last time of for
    count := count + 1; // For the return
    return s;
}
```

From the above we can estimate that invocation of ***Sum()*** executes total number of **2n+3** steps.

✓ **Example-2:**Consider the algorithm that computes Sum of n numbers recursively

```
Algorithm RSum(a, n)
{
    count := count + 1; // For the if conditional
    if (n ≤ 0) then
    {
        count := count + 1; // For the return
        return 0.0;
    }
    else
    {
        count := count + 1;  // For the addition, function
                             // invocation and return
        return RSum(a, n − 1) + a[n];
    }
}
```

- When analyzing a recursive program for its step count, we often obtain arecursive formula for the step count, for example

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

- These recursive formulas are referred to as recurrence relations. One way of solving is to make repeated substitutions for each occurrence of the function t_{RSum} on the right-hand side until all such occurrences disappear

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

- So, the step count for RSum is 2n+2

✓ The **second method** to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

- First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

✓ **Example 1:** Consider the algorithm ***sum()***.

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			$2n+3$

Table: Step table for the Sum() algorithm

✓ **Example 2:** Consider the algorithm that computes Sum of n numbers recursively

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table: Step table for the RSum() algorithm

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), Ω(big omega), and *Θ* (big theta).

Informal Introduction

Informally, *O(g(n))* is the set of all functions with a lower or same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity). Thus, to give a few examples, the following assertions are all true:

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

Indeed, the first two functions are linear and hence have a lower order of growth than $g(n) = n^2$, while the last one is quadratic and hence has the same order of growth as n^2. On the other hand,

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

Indeed, the functions n^3 and $0.00001n^3$ are both cubic and hence have a higher order of growth than n^2, and so has the fourth-degree polynomial $n^4 + n + 1$.

The second notation, Ω *(g(n))*, stands for the set of all functions with a higher or same order of growth as *g(n)* (to within a constant multiple, as *n* goes to infinity).

For example,

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

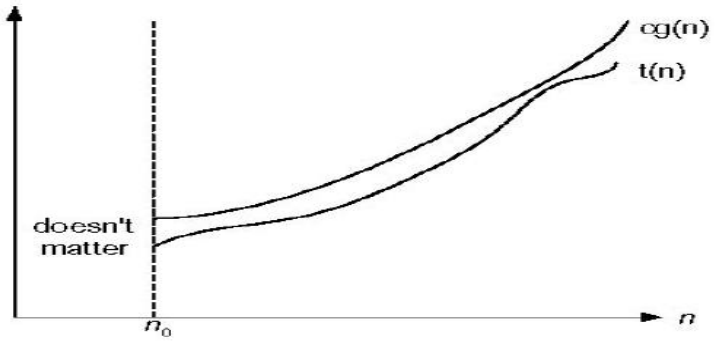
Finally, Θ *(g(n))* is the set of all functions that have the same order of growth as *g(n)*.

Big Oh- O 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

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

The definition is illustrated in the following figure.



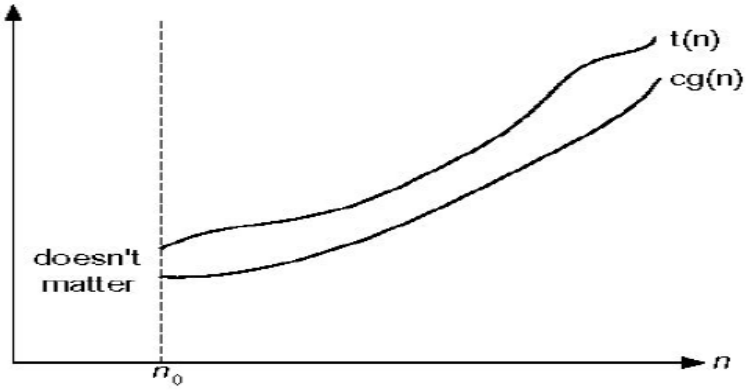
Big-oh notation: $t(n) \in O(g(n))$

Big Omega- Ω 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

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

The definition is illustrated in the following figure.



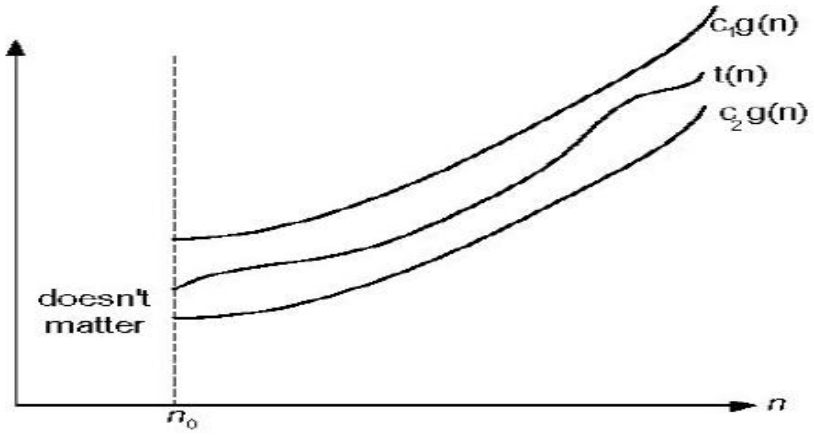
Big-omega notation: $t(n) \in \Omega(g(n))$

Big Theta- Θ notation

DEFINITION 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

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated in the following figure.



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

Examples: Refer class notes

- **Useful property involving the asymptotic notations**

  Using the formal definitions of the asymptotic notations, we can prove their general properties. 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 $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

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

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

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

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

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

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

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. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

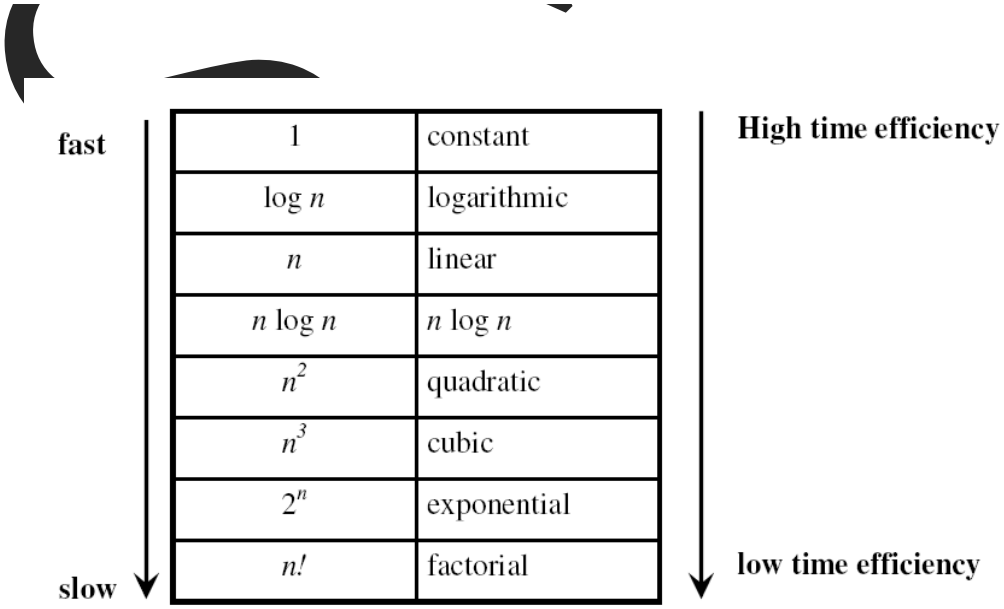
$$\left.\begin{array}{|l|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

For example, we can check whether an array has equal elements by the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n-1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

❖ Basic efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.

Class	Name	Comments
1	*constant*	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.
$\log n$	*logarithmic*	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	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	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	*cubic*	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	*exponential*	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.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.



4. MATHEMATICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating the **input size of the algorithm.**
2. Identify algorithm's **basic operation.**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, then investigate **worst, average, and best case efficiency** separately.
4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas(**Refer class notes for formulas used for simplifying).**

Example: Finding the largest element in a 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 $maxval$

Analysis:

1. Input size: the number of elements = n (size of the array)
2. Two operations can be considered to be as basic operation i.e.,
   a) **Comparison : A[i]>maxval.**
   b) Assignment :: maxval←A[i].

Here the comparison statement is considered to be the basic operation of the algorithm.

3. No best, worst, average cases- because the number of comparisons will be same for all arrays of size n and it is not dependent on type of input.
4. Let C(n) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and n – 1.

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

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

$$C(n) = \sum_{i=1}^{n-1} 1 \qquad 1 + 1 + 1 + \ldots + 1 \quad [(n-1) \text{ number of times}]$$

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

Example: Element uniqueness problem-Checks whether the elements in the array are distinct or not.

Algorithm *UniqueElements (A[0..n-1])*
//Checks 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***

Analysis

1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison
3. Best, worst, average cases exists

Worst case input is an array giving largest comparisons.

- Array with no equal elements
- Array with last two elements are the only pair of equal elements

4. Let C (n) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits i + 1 and n – 1; and this is repeated for each value of the outer loop i.e, for each value of the loop's variable i between its limits *0 and n – 2*.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

5.Simplify the summation using standard formulas as follows:

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

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

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

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

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

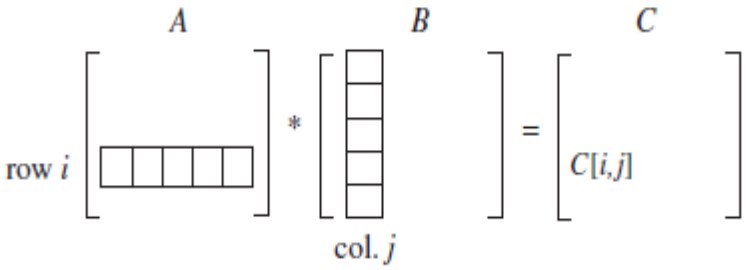
$$C(n) = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$C(n) = (n-1)\left((n-1) - \frac{(n-2)}{2}\right)$$

$$C(n) = (n-1)\frac{(2n-2-n+2)}{2}$$

$$C(n) = (n-1)(n)/2$$
$$= (n^2-n)/2$$
$$= (n^2)/2 - n/2$$
$$C(n) \in \Theta(n^2)$$

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



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

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
```

Analysis:

1. We measure an input's size by matrix order *n*.

2. There are two arithmetical operations in the innermost loop here—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation. Actually, we do not have to choose between them, because on each repetition of the innermost loop each of the two is executed exactly once. So by counting one we automatically count the other. Hence, we consider multiplication as the basic operation.

3. 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.)

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

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

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

Now, we can compute this sum by using formula (S1) and rule (R1). Starting with the innermost sum, which is equal to n, we get

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

If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

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

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

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

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 ← ⌊n/2⌋
return count
```

- ✓ First, notice that the most frequently executed operation here is not inside the **while** loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed.
- ✓ Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.
- ✓ A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
- ✓ The exact formula for the number of times the comparison $n>1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$ the number of bits in the binary representation of n.

5. MATHEMATICAL ANALYSIS(TIME EFFICIENCY) OF RECURSIVE ALGORITHMS

General plan for analyzing efficiency of recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. **Solve** the recurrence.

Example 1: Factorial function

Definition: $n! = 1 * 2 * \ldots * (n-1) * n$ **for** $n \geq 1$ **and** $0! = 1$

Recursive definition of $n!$**:** $F(n) = F(n-1) * n$ **for** $n \geq 1$ **and**

$F(0) = 1$

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

Analysis:

1. Input size: given number = n
2. Basic operation: multiplication
3. No best, worst, average cases.
4. Let M(n) denotes number of multiplications

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

Where: $M(n-1)$: to compute Factorial $(n-1)$
1 :to multiply Factorial $(n-1)$ by n

5. Solve the recurrence relation using backward substitution method:

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

…

The general formula for the above pattern for some 'i' is as follows:

= M(n-i) + i

By taking the advantage of the initial condition given i.e.,M(0)=0, we now substitute i=n in the patterns formula to get the ultimate result of the backward substitutions.

=M(n-n)+n

= M(0) + n

M(n) = n

$$M(n) \in \Theta(n)$$

The number of multipilcations to compute the factorial of 'n' is n where the time complexity is linear.

Example 2: Tower of Hanoi Problem

In this puzzle, we have *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.

- ✓ 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 i.e., nth 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).
- ✓ Of course, if $n = 1$, we simply move the single disk directly from the source peg to the destination peg.

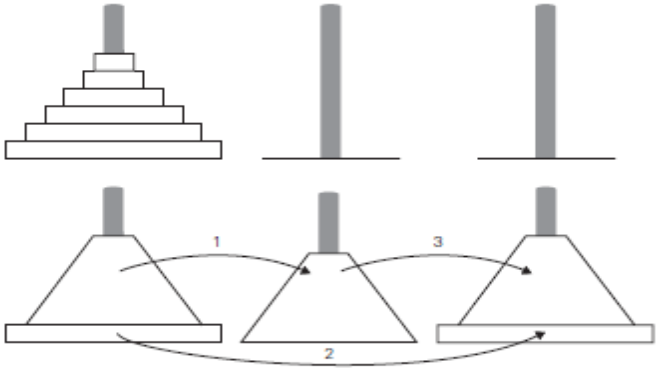


Figure: Tower of hanoi problem

Analysis:

Let us apply the general plan outlined above to the Tower of Hanoi problem.

1.The number of disks *n* is the obvious choice for the input's size indicator.

2.Moving one disk as the algorithm's basic operation.

3. Clearly, 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,

M(*n*) = 2M(*n*-1) + 1, M(1) = 1 and

The initial condition is:

M(1) = 1 for n=1

5. Solve the recurrence relation using backward substitution method:

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 + 2^0 sub. $M(n-2) = 2M(n-3) + 1$

= 2^2*(2M(n-3) + 1) + 2^1 + 2^0

= 2^3*M(n-3) + 2^2 + 2^1 + 2^0

= …

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

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

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

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

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of *n* .This is not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is the problem's intrinsic difficulty that makes it so computationally hard.

EXAMPLE 3 A recursive version of the algorithm that computes number of binary digits in *n*'s 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 $BinRec(\lfloor n/2 \rfloor) + 1$

- ✓ Let us set up a recurrence and an initial condition for the number of additions *A(n)* made by the algorithm.
- ✓ 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

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

- ✓ Since the recursive calls end when *n* is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0. \text{ for } n=1$$

- ✓ The presence of $\lfloor n/2 \rfloor$ in the function's argument makes the method of backward substitutions stumble on values of *n* that are not powers of 2.
- ✓ Therefore, 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*. (Alternatively, after getting a solution for powers of 2, we can sometimes fine-tune this solution to get a formula valid for an arbitrary *n.)*
- ✓ Applying the same to the recurrence, for $n = 2^k$ *which* takes the form

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

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 $k = \log_2 n$,

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

6. IMPORTANT PROBLEM TYPES

The most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems

6.1 Sorting

- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing order.
- As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees.
- In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose 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***.
- 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:

✓ A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where i < j, then in the sorted list they have to be in positions i` and j` respectively, such that i`< j`

✓ 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.

6.2 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 straightforward 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.
- For searching, too, there is no single algorithm that fits all situations best.
- Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on.

6.3 String Processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A,C, G, T}.

- One particular problem searching for a given word in a text—has attracted special attention from researchers referred to as ***string matching***.

6.4 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 are an interesting subject to study, for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of applications, including transportation, communication, social and economic networks, project scheduling, and games.
- Basic graph algorithms include graph-traversal algorithms (how can one reach all the points in a network?), shortest-path algorithms (what is the best route between two cities?), and topological sorting for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?).
- Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem.
- The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through *n* cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray crystallography, and genetic engineering.
- The ***graph-coloring problem*** seeks to assign the smallest number of colors to the vertices of a graph so that no two adjacent vertices are the same color. This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled at the same time, a solution to the graph-coloring problem yields an optimal schedule.

6.5 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.

7. 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.

7.1 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.

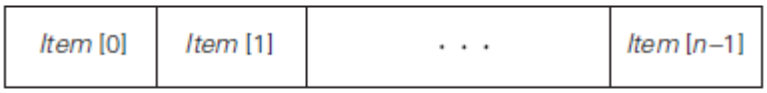


Figure : Array of *n* elements.

- 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.

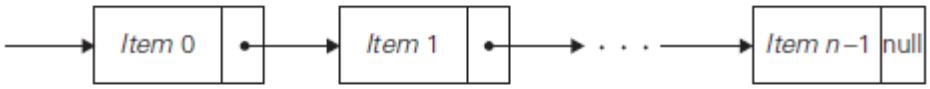


Figure : Singly linked list of *n* elements.

- 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.

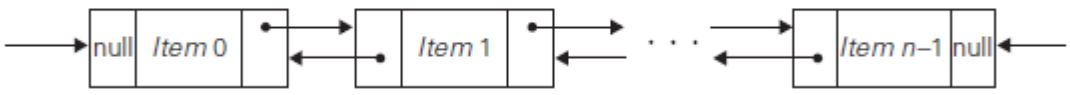


Figure : Doubly linked list of *n* elements.

- 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.

7.2 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)}.

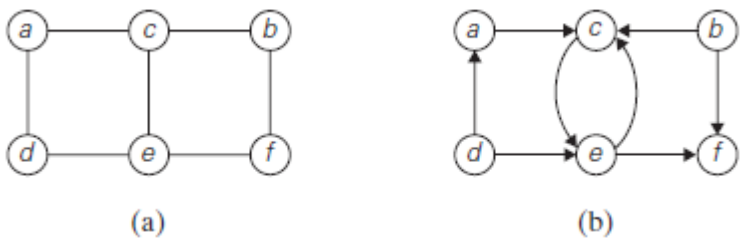


Figure : (a) Undirected graph. (b) Digraph.

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 x n** boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i^{th} row and the j^{th} column is equal to 1 if there is an edge from the i^{th} vertex to the j^{th} vertex, and equal to 0 if there is no such edge.
- ✓ The **adjacency lists** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge).

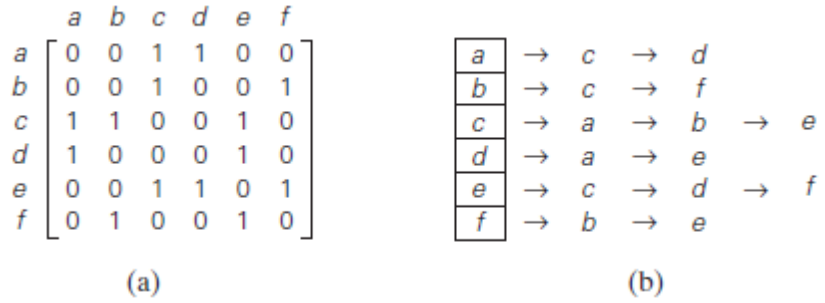


Figure : (a) Adjacency matrix and (b) adjacency lists of the graph in Figure (a)

Weighted Graphs

- ✓ A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***.
- ✓ An interest in such graphs is motivated by numerous real-world applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.
- ✓ Both principal representations of a graph can be easily adopted to accommodate weighted graphs.
  - If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the ith to the jth vertex if there is such an edge and a special symbol, e.g.,∞, if there is no such edge. Such a matrix is called the ***weight matrix*** or ***cost matrix***.
  - This approach is illustrated in below figure (b) for the weighted graph in figure (a). (For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.)

- Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge (Figure c).

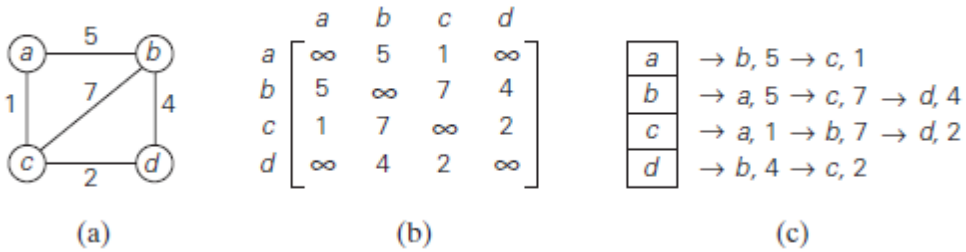
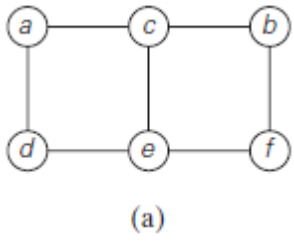


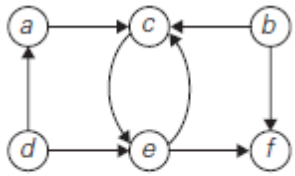
Figure : (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Paths and Cycles

- ✓ 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.
- ✓ If all vertices of a path are distinct, the path is said to be ***simple***. The ***length*** of a path is the total number of vertices in the vertex sequence defining the path minus 1, which is the same as the number of edges in the path.
- ✓ For example, a, c, b, f is a simple path of length 3 from a to f in the graph in below figure (a), whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f.



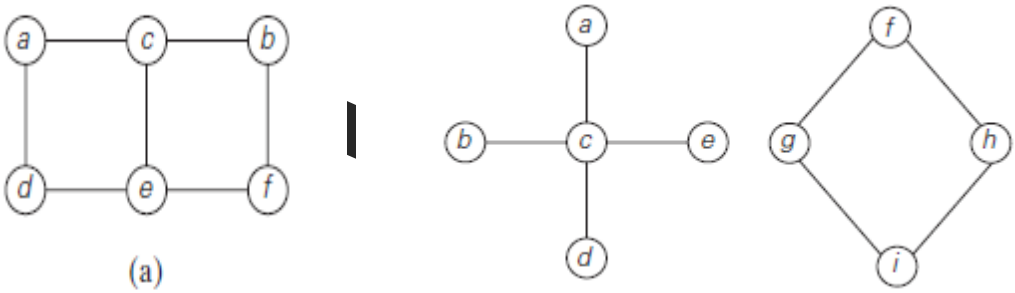
- ✓ In the case of a directed graph, we are usually interested in directed paths. A ***directed path*** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next.
- ✓ For example, a, c, e, f is a directed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



(b)

✓

- ✓ 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*. If we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece.
- ✓ If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph.
- ✓ Formally, a ***connected component*** is a maximal (not expandable by including another vertex and an edge) connected subgraph2 of a given graph. For example, the graphs in following figures (a) is connected, whereas the graph in figure (b) is not, because there is no path, for example, from *a* to *f*.



(a)

✓

- ✓ The graph in figure (c) has two connected components with vertices {*a, b, c, d, e*} and {*f, g, h, i*}, respectively.
- ✓ A ***cycle*** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, *f , h, i, g, f* is a cycle in the graph in figure (c).
- ✓ A graph with no cycles is said to be ***acyclic***.

7.3 Trees

- ✓ A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph as in figure (a) below.
- ✓ A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree as in figure (b) below

✓ 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.

\

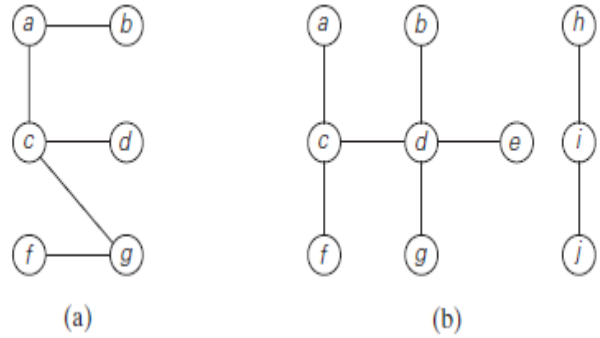


Figure : (a) Tree. (b) Forest.

❖ **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 root still below (level 2), and so on. Figure below presents such a transformation from a free tree to a \rooted tree.

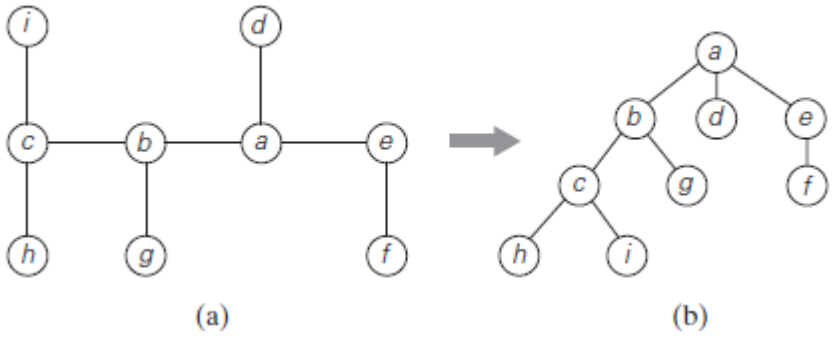


Figure : (a) Free tree. (b) Its transformation into a rooted tree.

✓ For any vertex *v* in a tree *T* , all the vertices on the simple path from the root to that vertex are called ***ancestors*** of *v*.

✓ The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as the set of ***proper ancestors***.

- ✓ If *(u, v)* is the last edge of the simple path from the root to vertex *v* (and $u \neq v$), *u* is said to be the ***parent*** of *v* and *v* is called a ***child*** of *u*; vertices that have the same parent are said to be ***siblings***.
- ✓ A vertex with no children is called a ***leaf*** ; a vertex with at least one child is called ***parental***.
- ✓ All the vertices for which a vertex *v* is an ancestor are said to be ***descendants*** of *v*; the ***proper descendants*** exclude the vertex *v* itself.
- ✓ All the descendants of a vertex *v* with all the edges connecting them form the ***subtree*** of *T* rooted at that vertex.
- ✓ Thus, for the tree in Figure (b) above, the root of the tree is *a*; vertices *d, g, f, h,* and *I* are leaves, and vertices *a, b, e,* and *c* are parental; the parent of *b* is *a*; the children of *b* are *c* and *g*; the siblings of *b* are *d* and *e*; and the vertices of the subtree rooted at *b* are {*b, c, g, h, i*}.
- ✓ The ***depth*** of a vertex *v* is the length of the simple path from the root to *v*.
- ✓ The ***height*** of a tree is the length of the longest simple path from the root to a leaf.
- ✓ For example, the depth of vertex *c* of the tree in Figure (b) above is 2, and the height of the tree is 3.
- ✓ Thus, if we count tree levels top down starting with 0 for the root's level, the depth of a vertex is simply its level in the tree, and the tree's height is the maximum level of its vertices

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.
- ✓ An example of a binary tree is given in Figure (a) below.

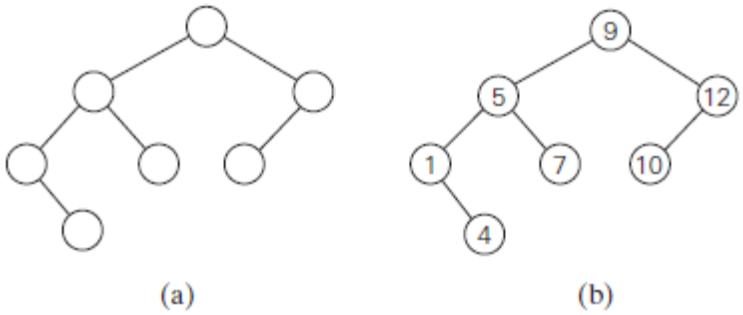


Figure : (a) Binary tree. (b) Binary search tree.

- ✓ The binary tree with its root at the left (right) child of a vertex in a binary tree is called the ***left*** (***right***) ***subtree*** of that vertex. Since left and right subtrees are binary trees as well, a binary tree can also be defined recursively. This makes it possible to solve many problems involving binary trees by recursive algorithms.
- ✓ In Figure (b), some numbers are assigned to vertices of the binary tree in Figure (a) above.
- ✓ Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called ***binary search trees***.
- ✓ A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively.
- ✓ Figure below illustrates such an implementation for the binary search tree in Figure (b) above.

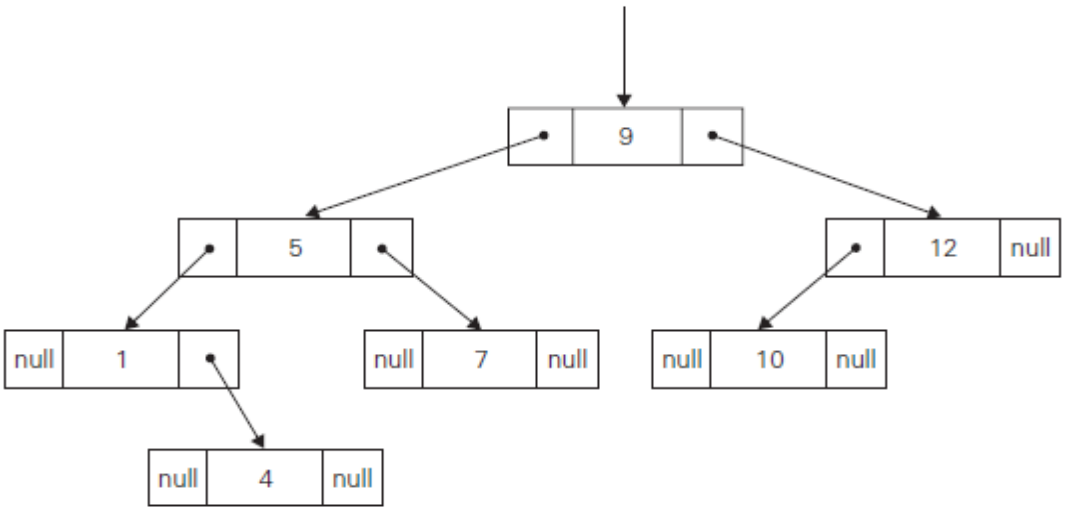


Figure : Standard implementation of the binary search tree in Figure (b) above.

- ✓ A computer representation of an arbitrary ordered tree can be done by simply providing a parental vertex with the number of pointers equal to the number of its children.
- ✓ This representation may prove to be inconvenient if the number of children varies widely among the nodes.
- ✓ We can avoid this inconvenience by using nodes with just two pointers, as we did for binary trees. Here, however, the left pointer will point to the first child of the vertex, and the right pointer will point to its next sibling. Accordingly, this representation is called the ***first child–next sibling representation***.
- ✓ Thus, all the siblings of a vertex are linked via the nodes' right pointers in a singly linked list, with the first element of the list pointed to by the left pointer of their parent.
- ✓ Figure (a) below illustrates this representation for the tree in Figure (b) above It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree.
- ✓ We get this representation by "rotating" the pointers about 45 degrees clockwise (see Figure b)

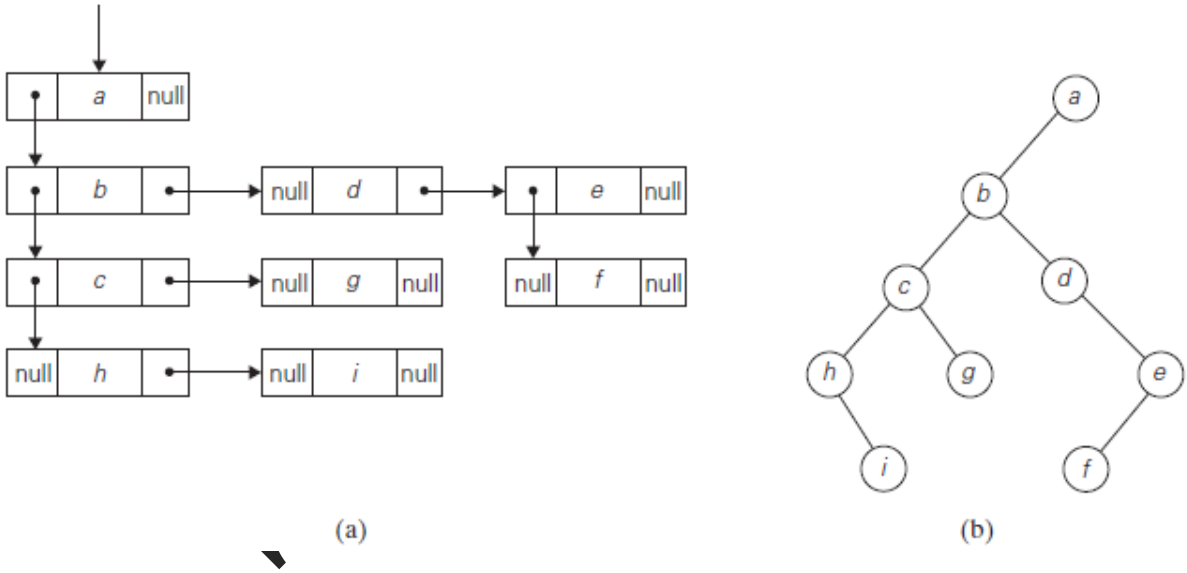


Figure : (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.

7.4 Sets and Dictionaries

- ✓ The notion of a set plays a central role in mathematics. 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., $S = \{n: n$ 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 bit string of size *n*, called a ***bit vector***, in which the *i*th element is 1 if and only if the *i*th element of *U* is included in set *S*.
- ✓ Thus, to continue with our example, if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, then $S = \{2, 3, 5, 7\}$ is represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast, but at the expense of potentially using a large amount of storage.
- ✓ 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. Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need. Note, however, the two principal points of distinction between sets and lists. First, a set cannot contain identical elements; a list can. This requirement for uniqueness is sometimes circumvented by the introduction of a ***multiset***, or ***bag***, an unordered collection of items that are not necessarily distinct.
- ✓ Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set.
- ✓ A list, defined as an ordered collection of items, is exactly the opposite. This is an important theoretical distinction, but fortunately it is not important for many applications.
- ✓ It is also worth mentioning that if a set is represented by a list, depending on the application at and, it might be worth maintaining the list in a sorted order.
- ✓ 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***. Note the relationship between this data structure and the problem of searching, we are dealing here with searching in a dynamic context. Consequently, an efficient implementation of a

dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations. There are quite a few ways a dictionary can be implemented. 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 *n*-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***.