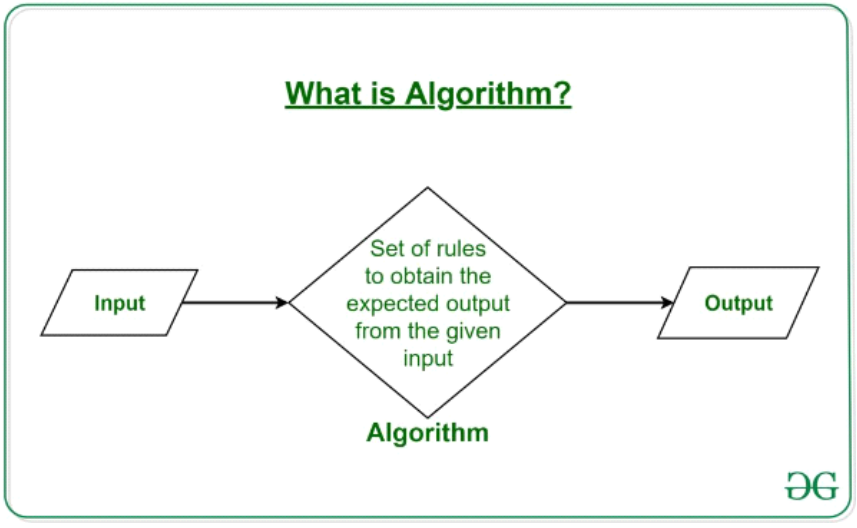
Analysis of Algorithms

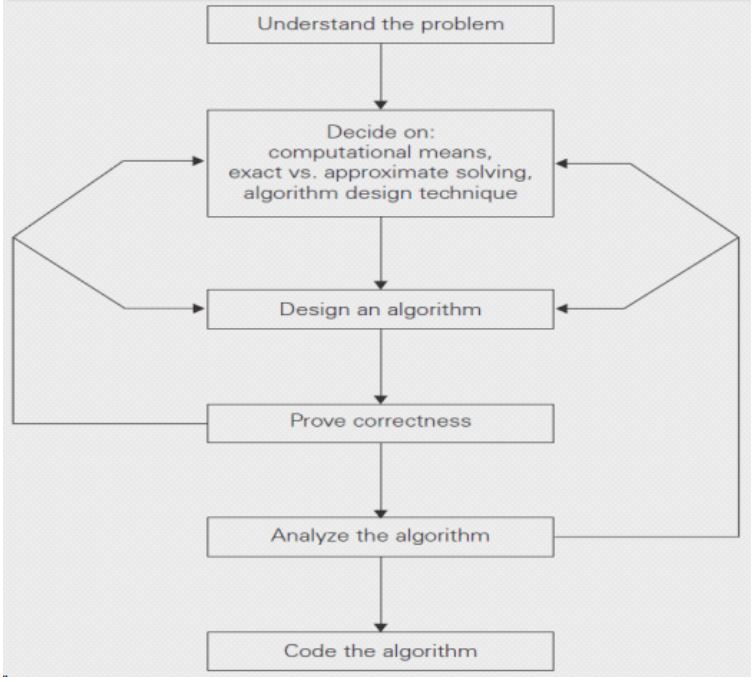
INTROUDCTION:

- What is an Algorithm?
  An algorithm is a step-by-step procedure or set of rules designed to perform a specific task or solve a particular problem. It is a sequence of well-defined, unambiguous instructions that can be executed by a computer or other processing device.



- **FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING**

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



(i) Understanding the Problem :

This is the first step in designing of algorithm.

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

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device.

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

(b) Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an **exact algorithm.**
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an **approximation algorithm.** i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

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

(iii) Methods of Specifying an Algorithm

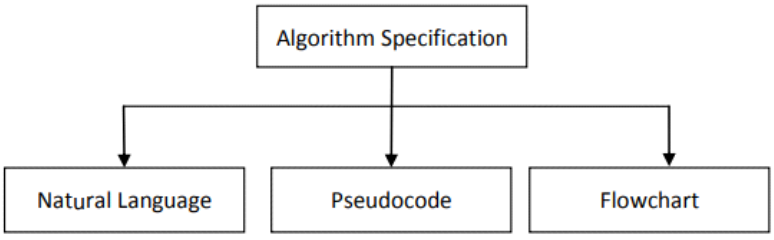
There are three ways to specify an algorithm.

They are:

a. Natural language

b. Pseudocode

c. Flowchart



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

a. Natural Language

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

Example: An algorithm to perform addition of two numbers

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

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

b. Pseudocode

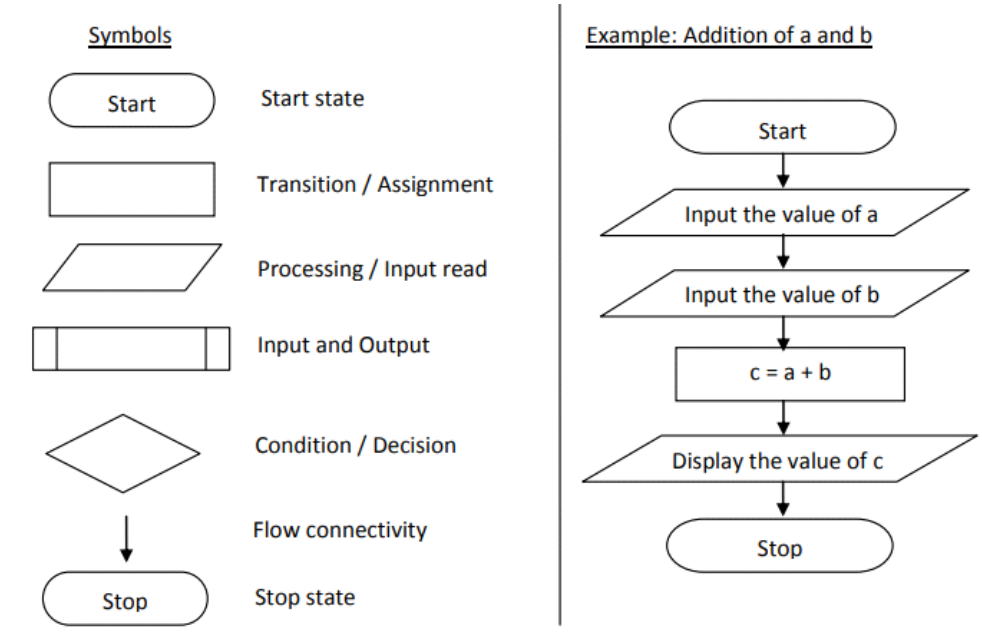
- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.
- For Assignment operation left arrow "←", for comments two slashes "//"**,if** condition, **for**, **while** loops are used

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

This specification is more useful for implementation of any language.

c. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.
Flowchart is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps



(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its correctness must be proved.
- An algorithm must yields a required result for every legitimate input in a finite amount of time
- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
- A common technique for proving correctness is to use **mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

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

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

- **Fundamentals of the Analysis of Algorithm Efficiency:**

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

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

- **Analysis Framework**
- There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:
- **Time efficiency,** indicating how fast the algorithm runs, and
- **Space efficiency**, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

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

(i) Measuring an Input's Size

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

(iii) Orders of Growth

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

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

n	$\sqrt{n}$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	**1**	**0**	**1**	**0**	**1**	**1**	**2**	**1**
2	**1.4**	**1**	**2**	**2**	**4**	**4**	**4**	**2**
4	**2**	**2**	**4**	**8**	**16**	**64**	**16**	**24**
8	**2.8**	**3**	**8**	$\mathbf{2.4 \cdot 10^1}$	**64**	$\mathbf{5.1 \cdot 10^2}$	$\mathbf{2.6 \cdot 10^2}$	$\mathbf{4.0 \cdot 10^4}$
10	**3.2**	**3.3**	**10**	$\mathbf{3.3 \cdot 10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^3}$	$\mathbf{3.6 \cdot 10^6}$
16	**4**	**4**	**16**	$\mathbf{6.4 \cdot 10^1}$	$\mathbf{2.6 \cdot 10^2}$	$\mathbf{4.1 \cdot 10^3}$	$\mathbf{6.5 \cdot 10^4}$	$\mathbf{2.1 \cdot 10^{13}}$
$\mathbf{10^2}$	**10**	**6.6**	$\mathbf{10^2}$	$\mathbf{6.6 \cdot 10^2}$	$\mathbf{10^4}$	$\mathbf{10^6}$	$\mathbf{1.3 \cdot 10^{30}}$	$\mathbf{9.3 \cdot 10^{157}}$
10^3	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$\mathbf{3.2 \cdot 10^2}$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

(iv) Worst-Case, Best-Case, and Average-Case Efficiencies

Consider Sequential Search algorithm some search key K

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

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

Clearly, the running time of this algorithm can be quite different for the same list size n.

In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

Worst-case efficiency

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

Best case efficiency

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

Average case efficiency

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

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

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

1.6 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

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

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

- O - Big oh notation
- Ω - Big omega notation
- Θ - Big theta notation

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

Example 1:

$n \in O(n^2)$,	$100n + 5 \in O(n^2)$,	$\frac{1}{2}n(n-1) \in O(n^2)$.
$n^3 \notin O(n^2)$,	$0.00001n^3 \notin O(n^2)$,	$n^4 + n + 1 \notin O(n^2)$.
$n^3 \in \Omega(n^2)$,	$\frac{1}{2}n(n-1) \in \Omega(n^2)$,	but $100n + 5 \notin \Omega(n^2)$.

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

(i) O - Big oh notation

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

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

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

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

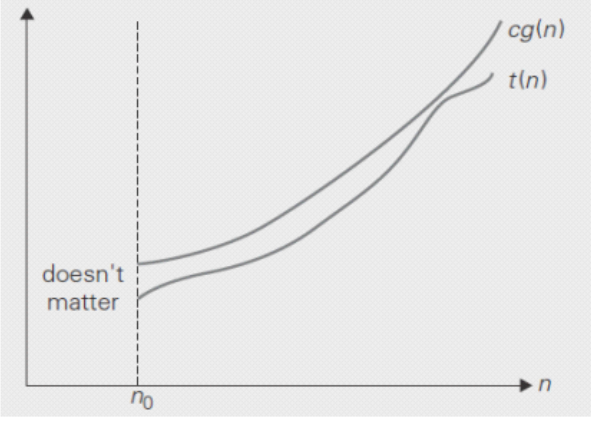


FIGURE 1.5 Big-oh notation: $t\ (n)\ \in\ O(g(n))$.

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

Proof: $100n + 5 \leq 100n + n$ (for all $n \geq 5$)

$= 101n$

$\leq 101n^2$ ($\because n \leq n^2$)

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** and $\mathbf{n_0}$. We have c=101 and n_0=5

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

Proof: $100n + 5 \leq 100n + 5n$ (for all $n \geq 1$)

$= 105n$

i.e., $100n + 5 \leq 105n$

i.e., $t(n) \leq cg(n)$

$\therefore 100n + 5 \in O(n)$ with c=105 and n_0=1

(ii) Ω - Big omega notation

A function *t(n)* is said to be in *Ω(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

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

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

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

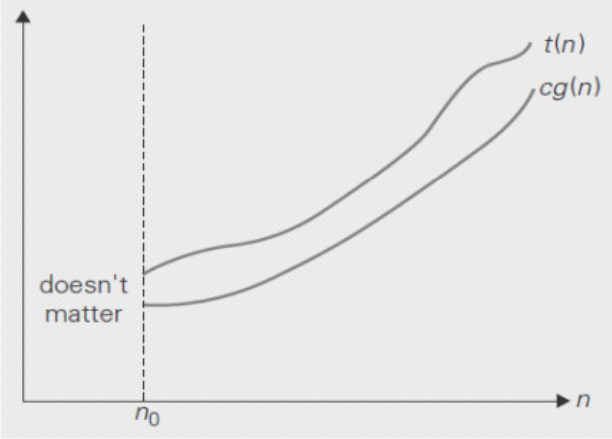


FIGURE 1.6 Big-omega notation: *t (n)* ∈ Ω *(g(n))*.

Example 4: Prove the assertions $n^3+10n^2+4n+2 \in \Omega(n^2)$.

Proof: $n^3+10n^2+4n+2 \geq n^2$ (for all $n \geq 0$)

i.e., by definition $t(n) \geq cg(n)$, where c=1 and n_0=0

(iii) Θ - Big theta notation

A function t(n) is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

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

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

Θ = Asymptotic tight bound = Useful for average case analysis

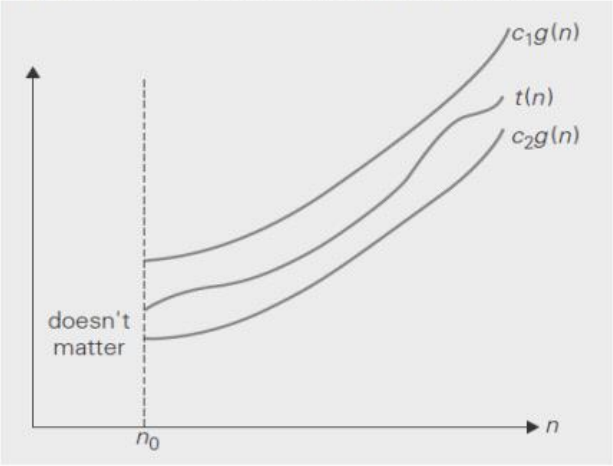


FIGURE 1.7 Big-theta notation: $t(n) \in \Theta(g(n))$.

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

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

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

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

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

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

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

Hence, $\frac{1}{2}n(n-1) \in \Theta(n^2)$, where c2= $\frac{1}{4}$, c1= $\frac{1}{2}$ and n_0=2

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

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

Useful Property Involving the Asymptotic Notations

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

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

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

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

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

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

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

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

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

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

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

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

Basic rules of sum manipulation

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

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

Summation formulas

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

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

Recurrence relations

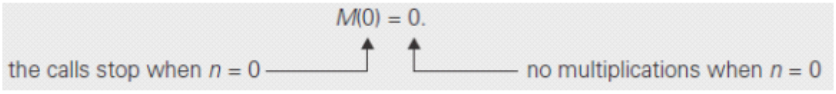
The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

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

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

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

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



Thus, the recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

M(n) = M(n − 1) + 1 for n > 0,

M(0) = 0 for n = 0.

Method of backward substitutions

$M(n)$	$= M(n-1) + 1$	substitute $M(n-1) = M(n-2) + 1$
	$= [M(n-2) + 1] + 1$	
	$= M(n-2) + 2$	substitute $M(n-2) = M(n-3) + 1$
	$= [M(n-3) + 1] + 2$	
	$= M(n-3) + 3$	
	…	
	$= M(n-i) + i$	
	…	
	$= M(n-n) + n$	
	$= n.$	

Therefore $M(n)=n$

EXAMPLE 2: consider educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third

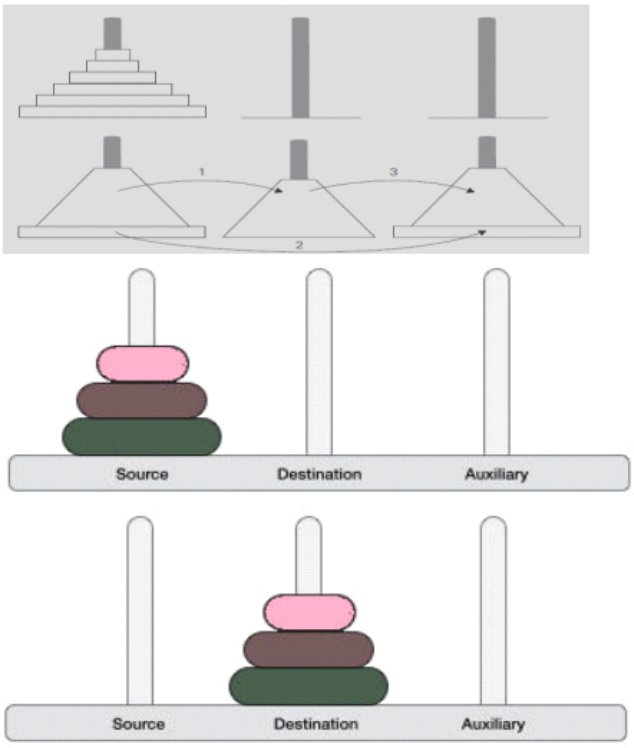


FIGURE 1.8 Recursive solution to the Tower of Hanoi puzzle.

ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively

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

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

if n=1

Move disk from A to C

else

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

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

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

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

Algorithm analysis

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

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

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

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

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

$= 2[2M(n-2) + 1] + 1$

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

$= 2^2[2M(n-3) + 1] + 2 + 1$

$= 2^3M(n-3) + 2^2 + 2 + 1$ sub. $M(n-3) = 2M(n-4) + 1$

$= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1$

...

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

. . .

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

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

Thus, we have an exponential time algorithm

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

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

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

Algorithm analysis

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

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

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

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

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

$A(2^0) = 0$.

backward substitutions

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

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

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

. . .

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

. . .

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

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

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

Algorithm analysis

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n − 1.
- Therefore, the number of multiplications made for every pair of specific values of variables i and j is

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

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

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

Now, we can compute this sum by using formula (S1) and rule (R1)

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

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

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

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

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

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

ALGORITHM Binary(n)

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

Algorithm analysis

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
- The exact formula for the number of times.
- The comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$.

2.1 BRUTE FORCE

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.

Examples:

1. Computing a^n : a * a * a * … * a (n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Selection Sort

- First scan the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.
- Then scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position in the sorted list.
- Generally, on the ***i***th pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with A_i:

$$A_0 \le A_1 \le \ldots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions | the last n – i elements

- After n − 1 passes, the list is sorted.

ALGORITHM SelectionSort(A[0..n − 1])

//Sorts a given array by selection sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in nondecreasing order

for *i* ← *0 to n − 2* **do**

min ← *i*

for j ← *i* + 1 to *n* − 1 do

if A[*j*]<A[*min*] *min* ← *j*

swap A[*i*] and A[*min*]

89	45	68	90	29	34	**17**
17	|	45	68	90	**29**	34	89
17	29	|	68	90	45	**34**	89
17	29	34	|	90	**45**	68	89
17	29	34	45	|	90	**68**	89
17	29	34	45	68	|	90	**89**

17 29 34 45 68 89 | 90

The sorting of list 89, 45, 68, 90, 29, 34, 17 is illustrated with the selection sort algorithm.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

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

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Note: The number of key swaps is only $\Theta(n)$, or, more precisely n – 1.

Bubble Sort

The bubble sorting algorithm is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i ($0 \le i \le n - 2$) of bubble sort can be represented by the following: $A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$

ALGORITHM BubbleSort(A[0..n − 1])

//Sorts a given array by bubble sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in nondecreasing order

for $i \leftarrow 0$ **to** n − 2 **do**

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

    if A[j + 1]<A[j] swap A[j] and A[j + 1]

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example.

89 ?↔ 45 68 90 29 34 17
45 89 ?↔ 68 90 29 34 17
45 68 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 90 ?↔ 34 17
45 68 89 29 34 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 29 89 ?↔ 34 17 | 90
45 68 29 34 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

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

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons.

$C_{worst}(n) \in \Theta(n^2)$

- **Linear/sequential Search**

Linear Search is the most basic searching algorithm. It is also known as Sequential Search. Linear Search is a brute force algorithm. It sequentially compares each element of the array/list to the element we want to search until a match is found or the whole list has been searched.

It doesn't depend on the arrangement of elements in an array, In simple words, the array need not be sorted.

We traverse each element of the array using a loop and compare it with the search value. See the following illustration for better understanding.

Code implementation :

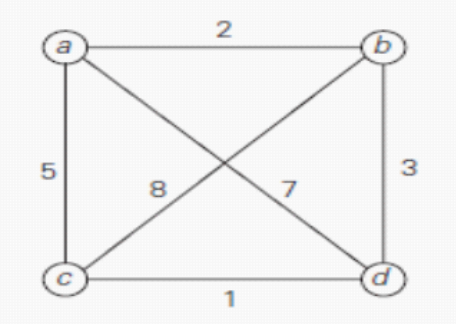
```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
int main()
{
  int array[50],i,target,num;
  printf("How many elements do you want in the array");
  scanf("%d",&num);
  printf("Enter array elements:");
  for(i=0;i<num;++i)
    scanf("%d",&array[i]);
  printf("Enter element to search:");
  scanf("%d",&target);
  for(i=0;i<num;++i)
    if(array[i]==target)
      break;
  if(i<num)
  printf("Target element found at location %d",i);
  else
    printf("Target element not found in an array");
  return 0;
}
```

2.4 TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $vi_0, vi_1, \ldots, vi_{n-1}, vi_0$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	Length
a ---> b ---> c ---> d ---> a	I = 2 + 8 + 1 + 7 = 18
a ---> b ---> d ---> c ---> a	**I = 2 + 3 + 1 + 5 = 11 optimal**
a ---> c ---> b ---> d ---> a	I = 5 + 8 + 3 + 7 = 23
a ---> c ---> d ---> b ---> a	**I = 5 + 1 + 3 + 2 = 11 optimal**
a ---> d ---> b ---> c ---> a	I = 7 + 3 + 8 + 5 = 23
a ---> d ---> c ---> b ---> a	I = 7 + 1 + 8 + 2 = 18

Time efficiency

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. **(n - 1)!**
- Consider two intermediate vertices, say, b and c, and then only permutations in which b precedes c. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of n.

```c
#include <stdio.h>
int tsp_g[10][10] = {
    {12,
     30, 33, 10, 45},
    {56, 22, 9, 15, 18},
    {29, 13, 8, 5, 12},
    {33, 28, 16, 10, 3},
    {1, 4, 30, 24, 20}};
int visited[10], n, cost = 0;

/* creating a function to generate the shortest path */
void travellingsalesman(int c)
{
    int k, adj_vertex = 999;
    int min = 999;

    /* marking the vertices visited in an assigned array */
    visited[c] = 1;

    /* displaying the shortest path */
    printf("%d ", c + 1);

    /* checking the minimum cost edge in the graph */
    for (k = 0; k < n; k++)
    {
        if ((tsp_g[c][k] != 0) && (visited[k] == 0))
        {
            if (tsp_g[c][k] < min)
            {
                min = tsp_g[c][k];
            }
            adj_vertex = k;
        }
    }
    if (min != 999)
    {
        cost = cost + min;
    }
    if (adj_vertex == 999)
    {
        adj_vertex = 0;
        printf("%d", adj_vertex + 1);
        cost = cost + tsp_g[c][adj_vertex];
        return;
    }
    travellingsalesman(adj_vertex);
}

/* main function */
int main()
{
    int i, j;
    n = 5;
    for (i = 0; i < n; i++)
    {
        visited[i] = 0;
    }
    printf("Shortest Path: ");
    travellingsalesman(0);
    printf("\nMinimum Cost: ");
    printf("%d\n", cost);
    return 0;
}
```

2.5 KNAPSACK PROBLEM

Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Real time examples:

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

Subset	Total weight	Total value
Φ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65 (Maximum-Optimum)**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{ 2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

- **Knapsack problem using greedy approach**

```
#include <stdio.h>
int main()
{
    float weight[50], profit[50], ratio[50], totalvalue, temp, capacity,
amount;
    int n, j, i;
    printf("Enter the number of items");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        printf("Enter the weight and profit for item%d:", i);
        scanf("%f%f", &weight[i], &profit[i]);
    }
    printf("Enter the capacity of knapsack");
    scanf("%f", &capacity);
```

```
    for (i = 0; i < n; i++)
        ratio[i] = profit[i] / weight[i];
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (ratio[i] < ratio[j])
            {
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                temp = weight[i];
                weight[j] = weight[i];
                weight[i] = temp;

                temp = profit[i];
                profit[j] = profit[i];
                profit[i] = temp;
            }
    printf("Knapsack problem using greedy algorithm\n");
    for (i = 0; i < n; i++)
    {
        if (weight[i] > capacity)
            break;
        else
        {
            totalvalue = totalvalue + profit[i];
            capacity = capacity - weight[i];
        }
    }
    if (i < n)
    {
        totalvalue = totalvalue + ratio[i] * capacity;
        printf("\n The minimum value is %f\n", totalvalue);
    }
    return 0;
}
```

```
Enter the number of items
5
Enter the weight and profit for item1:10
15
Enter the weight and profit for item2:20
25
Enter the weight and profit for item3:30
45
Enter the weight and profit for item4:40
55
Enter the weight and profit for item5:60
65
Enter the capacity of knapsack10
Knapsack problem using greedy algorithm

 The minimum value is 12.500000
PS C:\Users\test\Desktop\DAA_programs>
```

DFS (Depth First Search) algorithm:

DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or

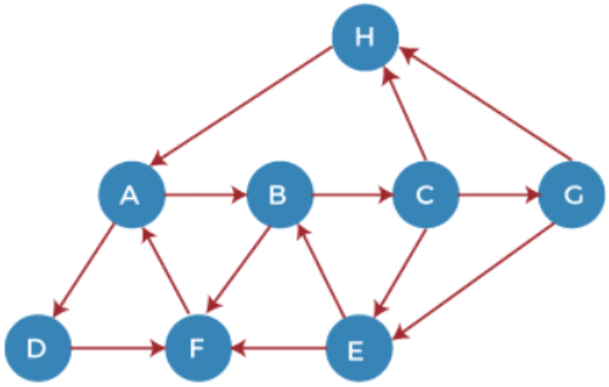
the node with no children.
Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm. The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Algorithm:
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H

BFS algorithm

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.
There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a

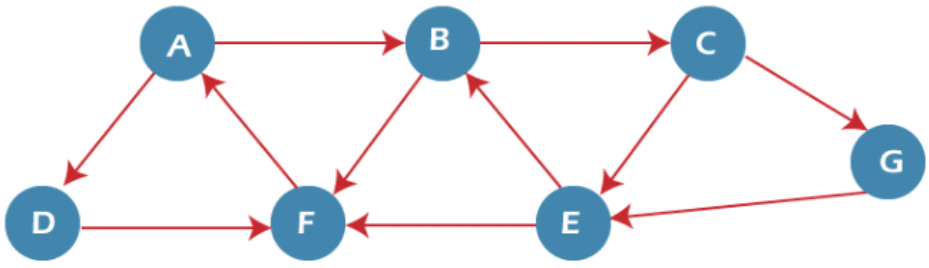
tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT

Example of BFS algorithm

Now, let's understand the working of BFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

- Decrease-and-conquer:

➢ "Decrease and Conquer" is a problem-solving technique and algorithmic design strategy that involves breaking down a problem into smaller subproblems, solving them individually, and then combining the solutions to address the original problem

➢ The basic idea behind "Decrease and Conquer" is to reduce the size of the problem at each step until a base case is reached, making it easier to solve