UNIT – II

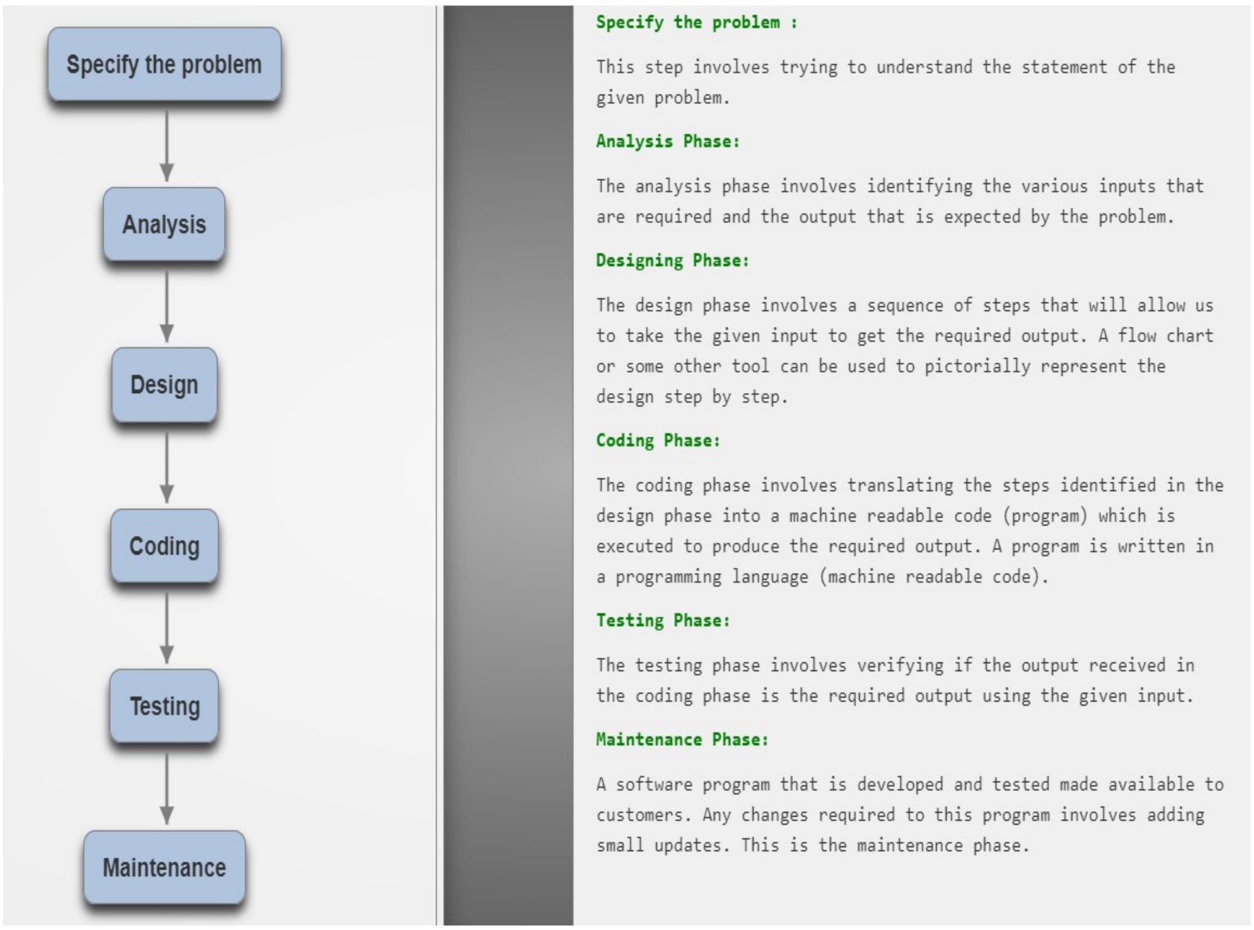
Introduction to computer problem solving: Introduction, the problem-solving aspect, topdown design, implementation of algorithms, the efficiency of algorithms, the analysis of algorithms.

Fundamental algorithms: Exchanging the values of two variables, counting, summation of a set of numbers, factorial computation, sine function computation, generation of the Fibonacci sequence, reversing the digits of an integer.

2.1 Introduction to Problem Solving:

Programming is a problem solving activity. If you are a good problem solver, you have the potential to become a good programmer.

Software development involves many stages. Each stage has different functionality and outcome. These stages form the Software Development Life Cycle, which is popularly known as SDLC.



Specify the problem :

This step involves trying to understand the statement of the given problem.

Analysis Phase:

The analysis phase involves identifying the various inputs that are required and the output that is expected by the problem.

Designing Phase:

The design phase involves a sequence of steps that will allow us to take the given input to get the required output. A flow chart or some other tool can be used to pictorially represent the design step by step.

Coding Phase:

The coding phase involves translating the steps identified in the design phase into a machine readable code (program) which is executed to produce the required output. A program is written in a programming language (machine readable code).

Testing Phase:

The testing phase involves verifying if the output received in the coding phase is the required output using the given input.

Maintenance Phase:

A software program that is developed and tested made available to customers. Any changes required to this program involves adding small updates. This is the maintenance phase.

2.2 What is an Algorithm:

An algorithm can be defined as “a sequence of steps to be followed for getting the desired output for a given input.”

The sequence of steps of an algorithm can be stated in a human readable language (English) or in pseudo-code (i.e, - a notation resembling a simplified programming language).

After the analysis stage of SDLC, producing an algorithm is the first step towards solving the given problem.

There may be several ways to solve a given problem, hence there may be several algorithms for a given problem.

Before attempting to write an algorithm, one should identify the expected inputs and outputs for the given problem.

An algorithm has the following properties:

1. Finiteness: An algorithm must always terminate after a finite number of steps.
2. Definiteness: Each step of an algorithm must be precisely defined, i.e., the step should perform a clearly defined task without much complication.
3. Input and Output values: The steps of the algorithm are designed to work on the input values and then produce the desired output in the final step.
4. Effectiveness: The efficiency of the steps and the accuracy of the output determine the effectiveness of the algorithm.

2.3 Problem Solving Aspect:

General Problem Solving Strategies:

There are number of general and powerful strategies which are used in problem solving. If the problem is phrased in one of these strategies, we can achieve considerable gains. The most widely used is divide-and-conquer strategy.

Divide- and- conquer strategy: In this, the original problem is divided into smaller problems which can be solved more efficiently. This type of strategy is mostly used in sorting, searching and selection algorithms.

Dynamic programming: This method is mostly used where we want to build up a solution for a problem via intermediate steps. This method relies on the idea that a good solution to a large problem can sometimes be built up from a good solution to smaller problems.

2.4 What is Top-Down Design:

The primary goal in computer problem solving is an algorithm which is capable of being implemented as a correct and efficient computer program. Once we have defined the problem to be solved and we have at least a vague idea on how to solve it, we can design algorithms using different techniques.

Top-down design or stepwise refinement is one such technique for designing algorithms. Top-down design is a strategy that we can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program implementation. It allows us to build our solutions to a problem in a stepwise fashion.

The following steps may be considered:

1) Breaking a problem into subproblems
The top-down design suggests that we take the general statements that we have about the solution, and break them down into a set of more precisely defined subtasks. The process of repeatedly breaking a task down into subtasks should continue until we eventually end up with subtasks that can be implemented as program statements.

2) Choice of a suitable data structure
All programs operate on data and consequently, the way the data is organized can have a profound effect on every aspect of the final solution. The things we must be aware of in setting up data structures are: can it be easily searched, updated, can we recover to an earlier state of the computation, etc.,

2.5 Implementation of Algorithms:

For a properly designed algorithm, the path of execution should flow in a straight line i.e from top to bottom. Programs implemented in this way are easy to understand, debug and modify.

The following steps should be followed while implementing the algorithm:

Use of Procedures to emphasize modularity: It is usually helpful to modularize the program along the lines of Top-down design. This practice allows us to implement a set of independent procedures to perform specific and well-defined tasks.

Choice of Variable names: To make the program meaningful and easier to understand, the names of the variables and constants should be appropriate. For example, if we have to make the changes on the day of the week it is better to use a variable name as **day** instead of using single letter **a** or **b**. A clear definition of all variables and constants at the start of each procedure can also be helpful.

Documentation of Programs: We need to associate brief and accurate comments for each program. The program must specify exactly what responses it requires from the user during execution. A good programming practice is to always write programs so that they can be executed and used by other people.

Program Testing: While testing the program, all the boundary conditions should be verified. It is often not possible to write programs that handle all input conditions that may be given to a particular problem. It is always a good practice to write an algorithm that should satisfy a wide range of possible inputs.

2.6 Efficiency of Algorithms:

An algorithm contains a sequence of steps to be followed to get the solution for a particular problem.

Efficiency of an algorithm mainly depends on its **design** and **implementation**. Since every algorithm uses computer resources to run, execution time and internal memory usage are important considerations while analyzing the efficiency of an algorithm.

Following are some simple methods that can be useful in designing efficient algorithms:

Redundant computation:

Most of the inefficiencies of an algorithm are mainly caused by redundant computation or unnecessary use of storage. The effect of these computations gets serious if they are embedded in a loop that gets executed multiple times. So these types of calculations must be avoided while designing efficient algorithms.

Inefficiency due to late termination :

Another place of inefficiency is taking more steps than required to solve the given problem. For example, In an alphabetically sorted dictionary of names, when searching for a specific name an inefficient example of implementation would be when the search is continued even after a point when it is certain that the name cannot be found.

Early detection of desirable output conditions:

One should able to identify the desired input and output conditions for the given problem. The algorithm should be simple and easy to modify, and also the efficiency of the algorithm should be independent of programming language, development environment.

2.7 Analysis of Algorithms:

Analysis Of Algorithms: Good algorithms generally possess the following qualities and capabilities:

- They are simple and easily understandable by others
- They can be modified easily if necessary
- They have correct and clear defined solutions
- They are economical in the use of computer time, storage and peripherals
- They should run independent of computer
- The solution should be pleasing and satisfied by the designer

2.7.1 Understanding the Time Complexity:

Time complexity is the computational complexity that measures or estimates the time taken to run an algorithm.

The total time required by the algorithm to run till its completion depends on the number of (machine) instructions the algorithm executes during its running time.

The actual time taken differs from machine to machine as it depends on several factors like CPU speed, memory access speed, OS, processor, etc.

So, we will take the number of operations executed by as the time complexity.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Thus, the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

2.7.2 Understanding Big Θ:

Three types of asymptotic notations are used to represent the growth of any algorithm, as input increases:

1. Big-Theta (Θ) : denotes "the same as" iterations (represents the average case).
2. Big-Oh (O) : denotes "fewer than or the same as" iterations (represents the worst case)
3. Big-Omega (Ω) : denotes "more than or the same as" iterations (represents the best case)

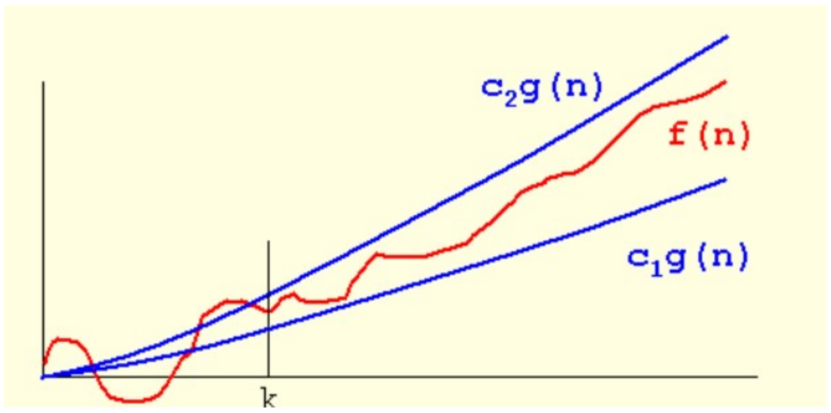
If the time complexity is represented by the Big-Θ notation, it is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression 20n2 + 8n, and we use the Big Theta (Θ) notation to represent this, then the time complexity would be Θ(n2), ignoring the constant coefficient and removing the insignificant part, which is 8n.

i.e., 20n2 + 8n = Θ(n2)

Here in the above example, if we say f(n) = 20n2 + 8n and g(n) = (n2), it means that there are positive constants c1, c2, and k, such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq k$.

The values of c1, c2, and k must be fixed for the function f and must not depend on n.



2.7.3 Understanding Big-Oh (O) Notation:

This notation is known as the upper bound of the algorithm, or a worst case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

Why do we need this representation when we already have the Big-Θ notation, which represents the tightly bound running time for any algorithm.

Let us take a small example to understand this.

Consider a linear search algorithm, in which we traverse an array elements from starting element to the last element, one by one to search a given number.

In worst case, we need to search till the end of the array to conform that the element is not found in the array.

But it can also happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.

Now in this case, saying that the Big-Θ or tight bound time complexity for linear search is T(n), will mean that the time required will always be related to n, as this is the right way to represent the average time complexity.

But when we use the Big-O notation, we mean to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it considers the worst case of the running algorithm.

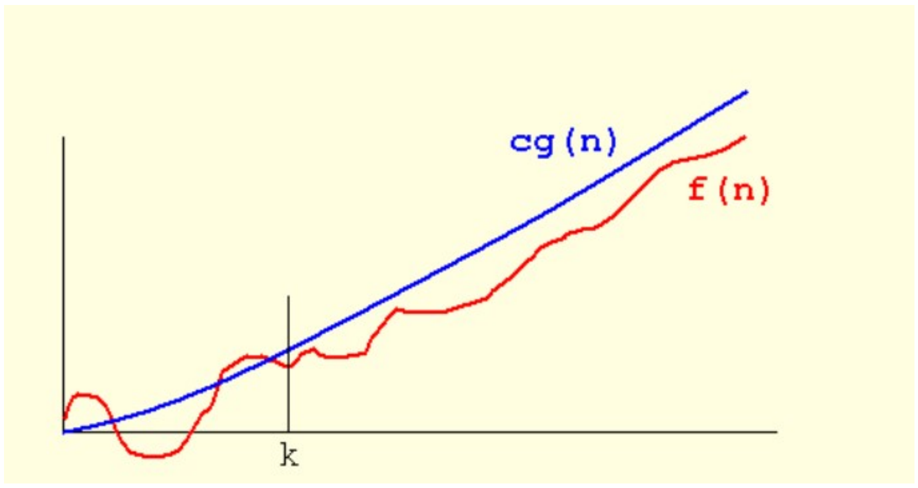
For example, if for some algorithm the time complexity is represented by the expression 20n2 +

8n, and we use the Big-Oh (O) notation to represent this, then the time complexity would be O(n2), ignoring the constant coefficient and removing the insignificant part, which is 8n.

i.e., 20n2 + 8n = O(n2)

Here, in the example above, if we say f(n) = 20n2 + 8n and g(n) = n2 it means there are positive constants c and k, such that $0 \le f(n) \le cg(n)$ for all n = k.

The values of c and k must be fixed for the function f and must not depend on n.



2.7.4 Understanding Big –Omega (Ω) Notation:

Big-Omega (Ω) notation is used to define the lower bound of an algorithm or we can say it as the best case of that algorithm.

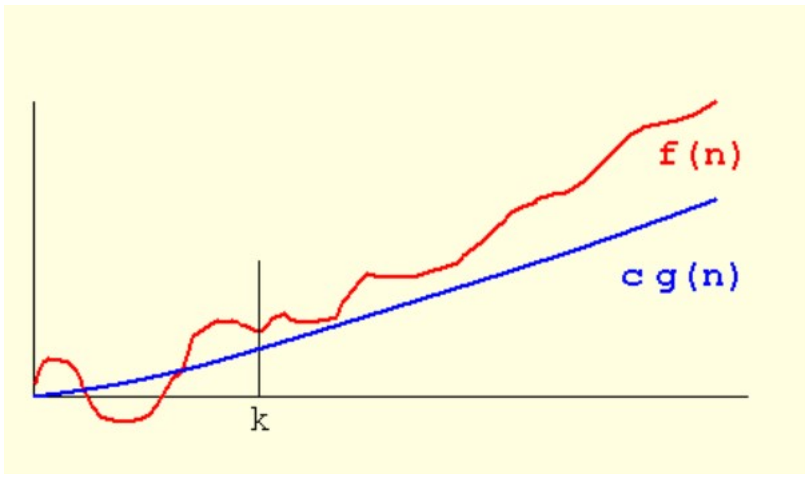
This always indicates the minimum time required to an algorithm for all the input values, which is the best case of that algorithm.

For example, if for some algorithm the time complexity is represented by the expression 20n2 + 8n, and we use the Big-Omega (Ω) notation to represent this, then the time complexity would be Ω(n2), ignoring the constant coefficient and removing the insignificant part, which is 8n.

i.e., 20n2 + 8n = Ω(n2)

Here, in the example above, if we say f(n) = 20n2 + 8n and g(n) = (n2) it means there are positive constants c and k, such that $0 \le f(n) \le cg(n)$ for all $n \ge k$.

The values of C and K must be fixed for the function f and must not depend on n



2.8 Fundamental Algorithms:

2.8.1 Exchanging the values of Two variables:

After reading the problem statement, we need to identify our input and output requirements. In this problem, our input requires two variables let us say first_num and second_num and we need to exchange the values of both.

Let us write an algorithm for this:

1. Start
2. Initialize first_num, second_num and temp to 0
3. Prompt the user to "enter the first number" and store the value to first_num.
4. Prompt the user to "enter the second number" and store the value to second_num.
5. Print the values of first_num and second_num before swapping.
6. Save the original value of second_num in temp
7. Assign original value of first_num to second_num
8. Assign value of temp to first_num
9. Print the values of first_num and second_num after swapping.
10. Stop.

Let us consider pseudo code for the same

```
procedure exchange(first_num, second_num)
   set temp to 0
   temp = second_num
   second_num =  first_num // second_num holds the value of first_num
   first_num = temp // first_num holds the value of a stored in temp (which is second_num's old value)
end procedure
```

Here we are using an intermediate variable temp, this allows the exchange of two variables to be processed correctly.

2.8.2 Algorithm for Counting:

Write an algorithm to count the number of students who passed the given examination. The pass marks are greater than or equal to 50.

Generally counting is made, to know the number of times a particular condition is satisfied. In our problem statement, we have to find the count of students who passed the given examination.

Let us consider an example.

1. Consider a set of marks given as 55, 35, 42, 77.
2. Initially consider the first one 55 and compare it with the condition >=50. If the condition is satisfied, the student has passed the exam. Add one to count.
3. The second one 35, does not satisfy the given condition and count remains the same.
4. The third one 42, also does not satisfy the given condition and count remains the same.
5. This process continues till the last one.

Algorithmic steps for above example:

1. Start
2. Initialize two variables num_students and pass_count to 0.
3. Prompt the user to enter the number of students and save the value in num_students
4. For each student do the following steps :
    - Prompt the user to enter the marks obtained by the student and save the value as marks.
    - if marks is greater than or equal to 50, then increment pass_count by 1.
5. Write the pass_count to the user.
6. Stop

2.8.3 Summation of a set of numbers:

Given a set of n numbers, design an algorithm which adds these numbers and returns
the resultant sum. Assume n is greater than or equal to zero.

Let us discuss how to write algorithm for this:

1. Start
2. Initialize the variables values_count and sum to 0.
3. Prompt the user to enter the number of values to be summed and store it in values_count.
4. Initialize a variable i to 1.
5. Repeat the following steps until i is greater than values_count.
    - Prompt the user to enter the value.
    - Increment the value of sum by the value entered.

6. Write the resultant sum of the numbers
7. Stop

2.8.4 Algorithm to find the factorial of a number:

The below algorithm is used to find the factorial of a given number.

Factorial of any positive integer n is denoted as n!. It is the product of all integers less than or equal to n

For Example:

```
5! = 5*4*3*2*1 = 120
```

For n! it can be written as 1*2*3*........... (n-1)*n

Algorithmic steps:

```
1. Start
2. Prompt the user to enter the number to find its factorial and save the value into number.
3. Initialize i = 1 and factorial_value = 1
4. Repeat the following steps until i is greater than number.
     o Set factorial_value = factorial_value * i
     o i = i + 1
5. Print the factorial_value as the factorial of the given number
6. Stop
```

2.8.5 Sine Function Computation:

Design an algorithm to evaluate the function sin(x) as defined by the infinite series expansion sin(x) = x/1! - x3/3! + x5/5! - x7/7!+ ....

Algorithmic Steps:

```
1. Start
2. Declare variables x, n, sin_x, term, d1, d2, and denominator
3. Read the values of x and n from user.
4. Set x = x * (3.142 /180) for converting x from degrees to radians.
5. Set term = x
6. Set sin_x = x
7. Set i = 1
8. Repeat the following steps until i < n
     1. Set d1 = 2*i
     2. Set d2 = d1 + 1
     3. Set denominator = d1 * d2
     4. Set term = term * -1 * x * x / denominator
     5. Set sin_x = sin_x + term
```

```
9. Display sin_x
10. STOP
```

2.8.6 Generation of the Fibonacci Series

Write an algorithm to generate and print the first n terms of the Fibonacci series, where n >= 1.

The first few terms are 0, 1, 1, 2, 3, 5, 7,.....

Fibonacci series is a series of numbers in which each number is the sum of the two preceding numbers. From the above sequence we can identify that

new term = preceding term + term before preceding term

Algorithmic Steps:

```
1. Start
2. Declare variables n, fib0, fib1, fibNext and count
3. Read value of n from user
4. Initialize variables fib0 = 0, fib1 = 1, count = 1
5. Repeat following statements until count <= n
    1. Display fib0
    2. fibNext = fib0 + fib1
    3. fib0 = fib1
    4. fib1 = fibNext
    5. count = count + 1
6. Stop
```

2.8.7 Algorithm for Reversing the Digit of an integer:

The below algorithm accepts a positive integer and reverses the order of its digits.

Let us consider an example.

If the input is 27953. To access the individual digits we will start from the least significant digit. In order to get digit 3, we use the mod function

27953 mod 10 gives 3

Therefore we can apply the following steps

remainder = n mod 10 i.e., 3

number = n div 10 i.e., 2795

The reversed integer can be calculated using

previous value of dreverse = n mod 10

dreverse = (previous value of dreverse) * 10 + (most recently extracted rightmost digit)

For the above example it would be as follows:

dreverse = dreverse * 10 + 3 i.e., 3

dreverse = dreverse * 10 + 5 i.e., 35

dreverse = dreverse * 10 + 9 i.e., 359

Algorithm description:

```
1. Start
2. Read the number number which is to be reversed, from the user
3. Declare two variables rev_number, digit and initialize them to 0
4. Declare a variable temp_number and initialize it with number
5. Perform the following steps until temp_number is equal to 0.
       o Set digit to temp_number mod 10.
       o Set rev_number = digit + rev_number * 10.
       o Set temp_number = floor (temp_number / 10 )
6. Print the result as the reverse of number is rev_number.
7. Stop
```