CONTROL STRUCTURES USED IN ALGORITHMS

- An algorithm has a finite number of steps.
- Some steps may involve decision-making and repetition.
- An algorithm may employ one of the following control structures:

  (a) sequence (Figure 2.10),

  (b) decision (Figure 2.11), and

  (c) repetition (Figure 2.12).

```
Step 1: [INITIALIZE] SET I = 1, N = 10
Step 2: Repeat Steps 3 and 4 while I<=N
Step 3: PRINT I
Step 4: SET I = I+1
        [END OF LOOP]
Step 5: END
```

Figure 2.12 Algorithm to print the first 10 natural of

TIME AND SPACE COMPLEXITY

- Analyzing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.
- The time complexity of an algorithm is basically the running time of a program as a function of the input size.
- Similarly, the space complexity of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

TIME AND SPACE COMPLEXITY

Worst-case, Average-case, Best-case

- Worst-case running time: This denotes the behavior of an algorithm with respect to the worst possible case of the input instance.
- Average-case running time The average-case running time of an algorithm is an estimate of the running time for an 'average' input.
- Best-case running time The term 'best-case performance' is used to analyze an algorithm under optimal conditions. For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

TIME AND SPACE COMPLEXITY

Expressing Time and Space Complexity

- The time and space complexity can be expressed using a function f(n) where n is the input size for a given instance of the problem being solved.
- Expressing the complexity is required when there are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.
- The most widely used notation to express this function f(n) is the Big O notation. It provides the upper bound for the complexity.

TIME AND SPACE COMPLEXITY

Algorithm Efficiency

- If a function is linear, the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- However, if an algorithm contains loops, then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.

TIME AND SPACE COMPLEXITY

Algorithm Efficiency

- Let us consider different cases in which loops determine the efficiency of an algorithm.
- *Linear Loops:*
  - for(i=0;i<100;i++)
    - f(n) = n
  - for(i=0;i<100;i+=2)
    - f(n) = n/2
- Logarithmic Loops
  - for(i=1;i<1000;i*=2)
  - for(i=1000;i>=1;i/=2)
    - f(n) = log n

TIME AND SPACE COMPLEXITY

Nested loops

- for(i=0;i<10;i++)

  for(j=1; j<10;j*=2) statement block;

  - n log n.
- for(i=0;i<10;i++)

  for(j=0; j<10;j++) statement block;

  - n^2
- for(i=0;i<10;i++)

  for(j=0; j<=i;j++) statement block;

  - n (n + 1) / 2

TIME AND SPACE COMPLEXITY

Quadratic loop

- In a quadratic loop, the number of iterations in the inner loop is equal to the number of iterations in the outer loop.
- Consider the following code in which the outer loop executes 10 times and for each iteration of the outer loop, the inner loop also executes 10 times.
- Therefore, the efficiency here is 100.

```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)  statement block;
```

- The generalized formula for quadratic loop can be given as $f(n) = n^2$.
- Dependent quadratic loop In a dependent quadratic loop, the number of iterations in the inner loop is dependent on the outer loop.

TIME AND SPACE COMPLEXITY

Quadratic loop

- Consider the code given below:

```
for(i=0;i<10;i++)
    for(j=0; j<=i;j++)  statement block;
```

- In this code, the inner loop will execute just once in the first iteration, twice in the second iteration, thrice in the third iteration, so on and so forth.
- In this way, the number of iterations can be calculated as 1 + 2 + 3 + ... + 9 + 10 = 55
- If we calculate the average of this loop (55/10 = 5.5), we will observe that it is equal to the number of iterations in the outer loop (10) plus 1 divided by 2.
- In general terms, the inner loop iterates (n + 1)/2 times.
- Therefore, the efficiency of such a code can be given as f(n) = n (n + 1)/2