# Module I

- **System Life Cycle**
- **Algorithms**
- **Performance Analysis**
  - **Space Complexity**
  - **Time Complexity,**
- **Asymptotic Notation**
- **Complexity Calculation of Simple Algorithms**

- **Algorithm**:
  - An algorithm is a finite set of instructions that accomplishes a particular task.
  - It is a step by step procedure to solve the problem.
  - It is the simplest representation of the program in our own language.
  - An algorithm can be abstract or quite detailed.
  - It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
  - Every step in an algorithm has its own logical sequence so it is easy to debug.
  - Algorithm does not follow any rules.

- **Properties of an Algorithm**
  - **Input:** Zero or more inputs are externally supplied.
  - **Output:** At least one output is produced.
  - **Definiteness:** Each instruction is clear and unambiguous.
    - "add 6 or 7 to x" , "compute 5/0" etc. are not permitted.
  - **Finiteness:** The algorithm terminates after a finite number of steps.
  - **Effectiveness:** Every instruction must be very basic so that it can be carried out by a person using only pencil and paper in a finite amount of time. It also must be feasible.
- **Computational Procedures**
  - Algorithms those are definite and effective.
  - Example: Operating system of a digital computer. (When no jobs are available, it does not terminate but continues in a waiting state until a new job is entered.)
- **Pseudo code:**
  - Pseudo code is an implementation of an algorithm
  - It is a more formal representation than an algorithm
  - Each step is very closer to the actual programming language
  - Acts as a bridge between the program and the algorithm.
  - Don't make the pseudo code abstract.
  - Don't be too generalized
  - The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.
  - Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
  - **Rules:**
    - An identifier begins with a letter
    - Blocks are indicated with matching braces: { and }
    - Assignment operator: =
    - Mathematical Operators: +,-,*,/,^,%
    - Boolean values: true, false.


Prepared By
Dona Jose, AP, CSE, VJCET

- Three logical operators: and, or, not
- Relational operators: $<$, $\leq$, $>$, $\geq$,==, $\neq$
- Array indices start at zero.
- **if** statement has the following forms:
  **if** <condition> **then** <statement-1>
  **if** <condition> **then** <statement-1> **else** <statement-2>
- The **while loop** takes the following form
  while < condition > do
  {
  <statements>
  }
- A **repeat-until** statement is constructed as follows
  repeat
  <statement 1>
  .
  <statement n>
  until <condition>
- The general form of a **for loop** is
  for variable=value1 to value2 step s do
  {
  <statement 1>
  .
  <statement n>
  }
- **break** statement is used to exit from innermost loop.
- **return** statement is used to exit from loops and functions
- **case** statement has the following form:
  case
  {
  :<condition 1>: <statements>
  .
  :<condition n>: <statements>
  :else: <statements>
  }
- An algorithm consists of a heading and a body.
  Algorithm Name (<parameterlist>)
  {
  Body of the algorithm
  }

- **Program:** It is the expression of an algorithm in a programming language
- **Recursive Algorithms**
  - A recursive function is a function that is defined in terms of itself.
  - An algorithm is said to be recursive if the same algorithm is invoked in the body.
  - Two types of recursive algorithms
    - **Direct Recursion**: An algorithm that calls itself is direct recursive.
    - **Indirect Recursion**: Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

- **<u>Performance Analysis</u>**
  - When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.
  - Performance analysis depends on **Space Complexity** and **Time Complexity**
  - **Space Complexity**
    - The space complexity of an algorithm is the amount of memory it needs to run to completion
    - Space Complexity = Fixed Part + Variable Part

      $S(P) = c + S_P$, Where P is any algorithm
      - A fixed part:
        - It is independent of the characteristics of the inputs and outputs.
        - Eg:
          - Instruction space(i.e., space for the code)
          - space for simple variables and fixed-size component variables
          - space for constants
      - A variable part:
        - It is dependent on the characteristics of the inputs and outputs.
        - Eg:
          - Space needed by component variables whose size is dependent on the particular problem instance being solved
          - Space needed by referenced variables
          - Recursion stack space.
  - **Time Complexity**
    - The time complexity of an algorithm is the amount of computer time it needs to run to completion. Compilation time is excluded.
    - Time Complexity = Frequency Count * Time for Executing one Statement
    - Frequency Count → Number of times a particular statement will execute
  - Eg1: Find the time and space complexity of matrix addition algorithm

| | **Step/Execution** | **Frequency Count** | **Total Frequency Count** |
|---|---|---|---|
| Algorithm Sum(A,n) | 0 | 0 | 0 |
| { | 0 | 0 | 0 |
| s=0 | 1 | 1 | 1 |
| for i=0 to n-1 do | 1 | n+1 | n+1 |
| S=s+A[i] | 1 | n | n |
| return s | 1 | 1 | 1 |
| } | 0 | 0 | 0 |
| | | | **2n +3** |

**Time Complexity = 2n + 3**

Space Complexity = Space for parameters and Space for local variables

A[]→n n→1 s→1 i→1

**Space complexity = n + 3**

- Eg2: Find the time and space complexity of matrix addition algorithm

| | **Step/Execution** | **Frequency Count** | **Total Frequency Count** |
|---|---|---|---|
| Algorithm mAdd(A,B,C,m,n) | 0 | 0 | 0 |
| { | 0 | 0 | 0 |
| for i=0 to m-1 do | 1 | m+1 | m+1 |
| for j=0 to n-1 do | 1 | m(n+1) | mn+m |
| C[i,j] := A[i,j] + B[i,j]; | 1 | mn | mn |
| } | 0 | 0 | 0 |
| | | | **2mn + 2m +1** |

**Time Complexity = 2mn + 2m + 1**

Space Complexity = Space for parameters and Space for local variables

m→1 n→1 a[]→mn b[]→mn c[]→mn i→1 j→1

**Space complexity = 3mn + 4**

- Eg3: Find the time and space complexity of recursive sum algorithm

| | Step/Execution | Frequency Count | | Total Frequency Count | |
|---|---|---|---|---|---|
| | | **n≤0** | **n>0** | **n≤0** | **n>0** |
| Algorithm RSum(A,n) | 0 | 0 | 0 | 0 | 0 |
| { | 0 | 0 | 0 | 0 | 0 |
| if n ≤ 0 then | 1 | 1 | 1 | 1 | 1 |
| return 0 | 1 | 1 | 0 | 1 | 0 |
| Else | 0 | 0 | 0 | 0 | 0 |
| return A[n] + RSum(A,n-1) | 1 + T(n-1) | 0 | 1 | 0 | 1 + T(n-1) |
| } | 0 | 0 | 0 | 0 | 0 |
| | | | | **2** | **2 + T(n-1)** |

**Time Complexity** = T(n) = $\begin{cases} 2 & \text{if } n<=0 \\ 2 + T(n-1) & \text{Otherwise} \end{cases}$

T(n) = 2 + T(n-1)
=2 + 2 + T(n-2)
=2 + 2 + 2+ T(n-3)
=2x3 + T(n-3)
. . .
=2xn +T(n-n)
**=2n + 2**

**Space Complexity** = Space for Stack
= Space for parameters + Space for local variables + Space for return address

For each recursive call the amount of stack required is 3

Space for parameters: A→1 n→1

Space for local variables: No local variables

Space for return address: 1

Total number of recursive call = n+1

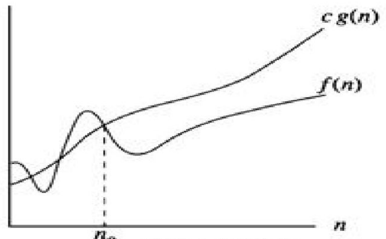
**Space complexity = 3(n+1)**

- **<u>Best Case, Worst Case and Average Case Complexity</u>**
  - In certain case we cannot find the exact value of frequency count. In this case we have 3 types of frequency counts
    - Best Case : It is the minimum number of steps that can be executed for a given parameter
    - Worst Case: It is the maximum number of steps that can be executed for a given parameter
    - Average Case: It is the average number of steps that can be executed for a given parameter
  - Eg: Linear Search
    - Best Case: Search data will be in the first location of the array.
    - Worst Case: Search data does not exist in the array
    - Average Case: Search data is in the middle of the array.

| | Best Case | | | Worst Case | | | Average Case | | |
|---|---|---|---|---|---|---|---|---|---|
| | S/E | FC | TFC | S/E | FC | TFC | S/E | FC | TFC |
| Algorithm Search(a,n,x) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| for i:=1 to n do | 1 | 1 | 1 | 1 | n+1 | n+1 | 1 | n/2 | n/2 |
| if a[i] ==x then | 1 | 1 | 1 | 1 | n | n | 1 | n/2 | n/2 |
| return i; | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| return -1; | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| } | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | **3** | | | **2n + 2** | | | **n+1** |

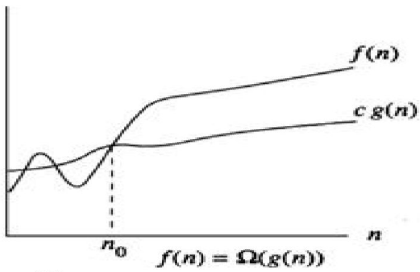
Best Case Complexity = **3**
Worst Case Complexity = **2n + 2**
Average Case Complexity= **n+1**

- **Asymptotic Notations**
  - It is the mathematical notations to represent frequency count. 5 types of asymptotic notations
    - **Big Oh (O)**
      - The function f(n) = O(g(n)) iff there exists 2 positive constants c and $n_0$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$
        - It is the measure of longest amount of time taken by an algorithm(Worst case).
        - It is asymptotically tight upper bound
        - O(1) : Computational time is constant
        - O(n) : Computational time is linear
        - $O(n^2)$ : Computational time is quadratic
        - $O(n^3)$ : Computational time is cubic
        - $O(2^n)$ : Computational time is exponential



$f(n) = O(g(n))$

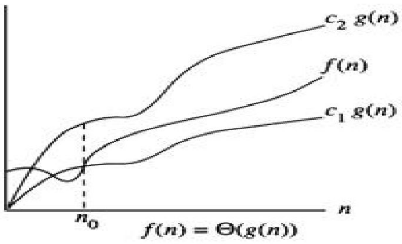
    - **Omega (Ω)**
        - The function f(n) = Ω (g(n)) iff there exists 2 positive constant c and $n_0$ such that $f(n) \geq c\, g(n) \geq 0$ for all $n \geq n_0$
        - It is the measure of smallest amount of time taken by an algorithm(Best case).
        - It is asymptotically tight lower bound



$f(n) = \Omega(g(n))$

    - **Theta (Θ)**
        - The function f(n) = Θ (g(n)) iff there exists 3 positive constants $c_1$, $c_2$ and $n_0$ such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$
        - It is the measure of average amount of time taken by an algorithm(Average case).


Prepared By
Dona Jose, AP, CSE, VJCET




- **Little Oh (o)**
  - The function f(n) = o(g(n)) iff for any positive constant c>0, there exists a constant $n_0$>0 such that $0 \le f(n) < c\,g(n)$ for all $n \ge n_0$
  - It is asymptotically loose upper bound

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

  g(n) becomes arbitrarily large relative to f(n) as n approaches infinity

- **Little Omega (ω)**
  - The function f(n) = ω(g(n)) iff for any positive constant c>0, there exists a constant $n_0$>0 such that $f(n) > c\,g(n) \ge 0$ for all $n \ge n_0$
  - It is asymptotically loose lower bound

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

  f(n) becomes arbitrarily large relative to g(n) as n approaches infinity

- **Examples:**

1. Find the O notation of the following functions
   a) f(n) = 3n + 2
      $3n + 2 \le 4n$ for all $n \ge 2$
      Here f(n)= 3n + 2   g(n)=n   c=4   $n_0$=2
      Therefore   3n + 2= **O(n)**
   b) $f(n) = 4n^3 + 2n + 3$
      $4n^3 + 2n + 3 \le 5n^3$ for all $n \ge 2$
      Here $f(n)= 4n^3 + 2n + 3$   $g(n)= n^3$   c=5   $n_0$=2
      Therefore   $4n^3 + 2n + 3=$ **$O(n^3)$**
   c) $f(n) = 2^{n+1}$
      $2^{n+1} \le 2\,2^n$ for all $n \ge 1$
      Here $f(n)= 2^{n+1}$   $g(n)= 2^n$   c=2   $n_0$=1
      Therefore   $2^{n+1} =$ **$O(2^n)$**
   d) $f(n) = 2^n + 6n^2 + 3n$
      $2^n + 6n^2 + 3n \le 7\,2^n$ for all $n \ge 5$
      Here $f(n)= 2^n + 6n^2 + 3n$   $g(n)= 2^n$   c=7   $n_0$=5
      Therefore   $2^n + 6n^2 + 3n =$ **$O(2^n)$**
   e) $f(n) = 10n^2 + 7$
   f) $f(n) = 5n^3 + n^2 + 6n + 2$
   g) $f(n) = 6n^2 + 3n + 2$
   h) f(n) = 100n + 6

2. Is $2^{2n} = O(2^n)$?
   $2^{2n} \le c\,2^n$
   $2^n \le c$
   There is no value for c and $n_0$ that can make this true.
   Therefore   $2^{2n}$ != **$O(2^n)$**

3. Is $2^{n+1} = O(2^n)$?
   $2^{n+1} \le c\, 2^n$
   $2x2^n \le c\, 2^n$
   $2 \le c$
   $2^{n+1} \le c\, 2^n$ is True if c=2 and n≥1.
   Therefore $2^{n+1} = \mathbf{O(2^n)}$

4. Find the **Ω** notation of the following functions
   a) $f(n) = 27\,n^2 + 16n + 25$
      $27\,n^2 + 16n + 25 \ge 27\,n^2$ for all $n \ge 1$
      Here c=27 $n_0=1$ $g(n)= n^2$
      $27\,n^2 + 16n + 25 = \mathbf{\Omega(n^2)}$
   b) $f(n) = 5\,n^3 + n^2 + 3n + 2$
      $5\,n^3 + n^2 + 3n + 2 \ge 5\,n^3$ for all $n \ge 1$
      Here c=5 $n_0=1$ $g(n)= n^3$
      $5\,n^3 + n^2 + 3n + 2 = \mathbf{\Omega(n^3)}$
   c) $f(n) = 3^n + 6n^2 + 3n$
      $3^n + 6n^2 + 3n \ge 5.3^n$ for all $n \ge 1$
      Here c=5 $n_0=1$ $g(n)= 3^n$
      $3^n + 6n^2 + 3n = \mathbf{\Omega(3^n)}$
   d) $f(n) = 4\,2^n + 3n$
   e) $f(n) = 3n + 30$
   f) $f(n) = 10\,n^2 + 4n + 2$

5. Find the **Θ** notation of the following functions
   a) $f(n) = 3n + 2$
      $3n + 2 \le 4\,n$ for all n≥2
      $3n + 2 = O(n)$

      $3n + 2 \ge 3\,n$ for all n≥1
      $3n + 2 = \Omega\,(n)$

      $3\,n \le 3n + 2 \le 4\,n$ for all n≥2
      $3n + 2 = \mathbf{\Theta(n)}$
   b) $f(n) = 3\,2^n + 4n^2 + 5n + 2$
      $3x2^n + 4n^2 + 5n + 2 \le 10x2^n$ for all n≥1
      $3x2^n + 4n^2 + 5n + 2 = O(2^n)$

      $3x2^n + 4n^2 + 5n + 2 \ge 3x2^n$ for all n≥1
      $3x2^n + 4n^2 + 5n + 2 = \Omega\,(2^n)$

      $3x2^n \le 3x2^n + 4n^2 + 5n + 2 \le 10\,2^n$ for all n≥1
      $3x2^n + 4n^2 + 5n + 2 = \mathbf{\Theta(2^n)}$
   c) $f(n) = 2\,n^2 + 16$
   d) $f(n) = 27n^2 + 16$

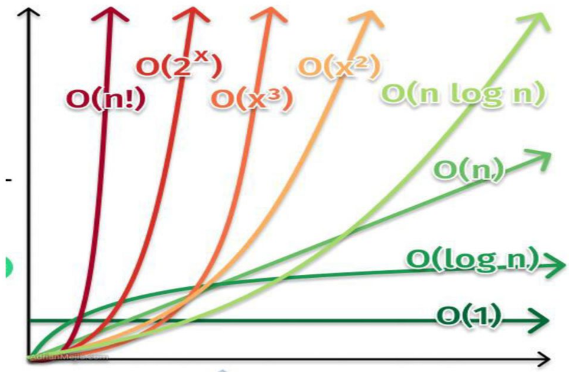
- **<u>Common Complexity Functions</u>**
  - Constant Time
    - An algorithm is said to be constant time if the value of f(n) is bounded by a value that does not depend on the size of input.
    - Computational time is constant
    - Eg: O(1)

- Logarithmic Time
  - An algorithm is said to be logarithmic time if $f(n) = O(\log n)$
- Linear Time
  - If $f(n) = O(n)$, then the algorithm is said to be linear time .
- Quadratic Time
  - If $f(n) = O(n^2)$, then the algorithm is said to be quadratic time .
- Polynomial Time
  - If $f(n) = O(n^k)$, then the algorithm is said to be polynomial time .
- Exponential Time
  - If $f(n) = O(2^n)$, then the algorithm is said to be exponential time .
- Factorial Time
  - If $f(n) = O(n!)$, then the algorithm is said to be factorial time

- **Running Time Comparison (Order of Growth)**
  - Logarithmic functions are very slow
  - Exponential functions and factorial functions are very fast growing

| n | log n | n | n log n | $n^2$ | $n^3$ | $2^n$ | n! |
|---|---|---|---|---|---|---|---|
| 10 | 3.3 | 10 | 3.3 x 10 | $10^2$ | $10^3$ | $10^3$ | 3.6 x $10^{60}$ |
| $10^2$ | 6.6 | $10^2$ | 6.6 x $10^2$ | $10^4$ | $10^6$ | 1.3 x $10^{30}$ | 9.3 x $10^{157}$ |
| $10^3$ | 10 | $10^3$ | 10 x $10^3$ | $10^6$ | $10^9$ | . | . |
| $10^4$ | 13 | $10^4$ | 13 x $10^4$ | $10^8$ | $10^{12}$ | . | . |
| $10^5$ | 17 | $10^5$ | 17 x $10^5$ | $10^{10}$ | $10^{15}$ | . | . |
| $10^6$ | 20 | $10^6$ | 20 x $10^6$ | $10^{12}$ | $10^{18}$ | . | . |

**$O(1) < O(\log n) < O(n) < O(n^k) < O(2^n) < O(n!)$**





Prepared By
Dona Jose, AP, CSE, VJCET

- **Time Complexity Calculation: Examples**
  1. Find the time complexity of Binary Search

```
Algorithm BinarySearch(A, low, high, search_data)
{
        flag=0
        while low<=high do
        {
                mid = (low + high)/2
                if A[mid]= search_data then
                {
                        flag = 1
                        break
                }
                else if A[mid] > search_data then
                        high=mid-1
                else
                        low=mid+1
        }
        if flag=0 then
                Print "Search data not found"
        else
                Print "Search_data found at index " mid
}
```

  - **Best Case Time Complexity of Binary Search**
    - The search data is at the middle index.
    - So total number of iterations required is 1
    - Therefore, Time complexity = **O(1)**

  - **Worst Case Time Complexity of Binary Search**
    - Assume that length of the array is n
    - At each iteration, the array is divided by half.
    - At Iteration 1, Length of array = n
    - At Iteration 2, Length of array = $n/2$
    - At Iteration 3, Length of array = $(n/2)/2 = n/2^2$
    - At Iteration k, Length of array = $n/2^{k-1}$
    - After k divisions, the length of array becomes 1

      $n/2^{k-1} = 1$

      $n = 2^{k-1}$
    - Applying log function on both sides:

      $\log_2 (n) = \log_2 (2^{k-1})$

      $\log_2 (n) = (k-1) \log_2 (2)$

      $k = \log_2 (n) + 1$
    - Hence, the time complexity **= O( $\log_2$ (n) )**

  - **Average case Time Complexity of Binary Search**
    - Total number of iterations required = $k/2 = (\log_2 (n)+1)/2$
    - Hence, the time complexity **= O( $\log_2$ (n) )**

  2. What is the time complexity of the following code

```
for(i=0; i<n; i++)
        s=s+i;
```

**Answer:**

- The for loop will execute n+1 times. It is the most frequently executing statement.
- So the time complexity = n+1 = **O(n)**

3. What is the time complexity of the following code

```
for(i=0; i<n*n; i++)
        s=s+i;
```

**Answer:**

- The for loop will execute $n^2+1$ times. It is the most frequently executing statement.
- So the time complexity = $n^2+1$ = **O($n^2$)**

4. What is the time complexity of the following code

```
i=1
while(i<=n)
{
        s=s+i;
        i=i*2;
}
```

**Answer:**

- The while loop will execute log n times.
- So the time complexity = log n = **O(log n)**

5. What is the time complexity of the following code

```
s=0
for(i=0; i<m; i++)
        for(j=0; j<n;j++)
                s=s+i*j;
```

**Answer:**

- The outer for loop will successfully execute m times
- For each successful case of outer for loop, the inner loop will successfully execute n times
- So the time complexity = m n = **O(mn)**

6. Calculate the frequency count of the statement x=x+1

```
for(i=1; i<=n; i++)
        for(j=1; j<=n; j=j*2)
                x=x+1;
```

**Answer:**

- The outer for loop will successfully execute n times
- For each successful case of outer for loop, the inner loop will successfully execute log n times
- So the frequency count of x=x+1 statement is **n log n**

7. Calculate the frequency count of the statement j=j*2

```
i=1;
while(i<=n)
{
        j=1
        while(j<=n)
        {
                j=j*2;
        }
        i=i+1;
}
```

**Answer:**

- The outer while loop will successfully execute n times
- For each successful case of outer while loop, the inner loop will successfully execute log n times
- So the frequency count of j=j*2 statement is **n log n = O(n log n)**

8. What is the time complexity of the following code

```
s=0
for(i=1; i<=n; i++)
        for(j=1; j<=i; j++)
                s=s+i*j;
```

**Answer:**

- When i=1, the inner loop will execute 1 time
- When i=2, the inner loop will execute 2 time
- When i=n, the inner loop will execute n time
- So the innermost statement will execute 1+2+3+…..+ n = n(n+1)/2 times
- So the time complexity = **n(n+1)/2** = **O($n^2$)**

9. What is the time complexity of the following code

```
s=0
for(i=1; i<=n; i++)
        for(j=i; j<0; j++)
                s=s+i*j;
```

**Answer:**

- The inner for loop will not execute at all. The frequency count of inner for loop is 0.
- The outer for loop will execute n times.
- So the time complexity = **n** = **O(n)**

10. Calculate the frequency count of the statement1

```
for(i=k; i<n; i=i*m)
        Statement1;
```

**Answer:**

- The for loop will successfully execute $\lceil \log_m (n/k) \rceil$ times
- So the frequency count of statement1 is $\mathbf{\lceil \log_m (n/k) \rceil = O(\lceil \log_m (n/k) \rceil)}$

11. Calculate the frequency count of the statement1

```
for(i=k; i<=n; i=i*m)
        Statement1;
```

**Answer:**

- The for loop will successfully execute $\lfloor \log_m (n/k) + 1 \rfloor$ times
- So the frequency count of statement1 is $\mathbf{\lfloor \log_m (n/k) + 1 \rfloor = O(\lfloor \log_m (n/k) \rfloor)}$

12. What is the time complexity of the following code

```
switch(key)
{
        case 1:  for(i=0;i<n;i++)
                         s=s+A[i]
                 break;
        case 2:  for(i=0;i<n;i++)
                         for(j=0;j<n;j++)
                                 s=s+B[i][j]
                 break;
}
```

**Answer:**

- Case 1 complexity=O(n)
- Case 2 complexity=$O(n^2)$
- The overall complexity **= $O(n^2)$**


Prepared By
Dona Jose, AP, CSE, VJCET