# Module I

- **Introduction to Algorithm Analysis**
  - **Characteristics of Algorithms**
  - **Criteria for Analysing Algorithms**
    - **Time and Space Complexity**
  - **Best, Worst and Average Case Complexities**
  - **Asymptotic Notations**
    - **Big-Oh (O), Big- Omega (Ω), Big-Theta (Θ), Little-oh (o) and Little- Omega (ω) and their properties.**
    - **Classifying functions by their asymptotic growth rate**
  - **Time and Space Complexity Calculation of simple algorithms**
  - **Analysis of Recursive Algorithms:**
    - **Recurrence Equations**
    - **Solving Recurrence Equations**
      - **Iteration Method**
      - **Recursion Tree Method**
      - Substitution method
      - **Master's Theorem**

- **Algorithm**: An algorithm is a finite set of instructions that accomplishes a particular task.
- **Characteristics 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.)
- **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
    1. **Direct Recursion**: An algorithm that calls itself is direct recursive.
    2. **Indirect Recursion**: Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.

- **Performance Analysis(Criteria for Analysing Algorithms)**
  - 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

1. 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
2. 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 mAdd(m,n,a,b,c) | 0 | 0 | 0 |
| { | 0 | 0 | 0 |
| for i=1 to m do | 1 | m+1 | m+1 |
| for j=1 to n 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**

- Eg2: 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)** $= \begin{cases} \mathbf{2} & \textbf{if n<=0} \\ \mathbf{2 + T(n-1)} & \textbf{Otherwise} \end{cases}$

T(n) = 2 + T(n-1)
=2 + 2 + T(n-2)
=2 + 2 + 2+ T(n-3)

$=2 \times 3 + T(n-3)$

. . .

$=2 \times n + T(n-n)$

$=\mathbf{2n + 2}$

**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)**

- **Examples:**

1. Discuss the time complexity of the following two functions

```
int fun1(int n)
{       if(n ≤ 1)
                return n;
        return 2xfun1(n-1);
}
int fun2(int n)
{       if(n ≤ 1)
                return n;
        return fun2(n-1)xfun2(n-1);
}
```

2. Analyze the complexity of the following program

```
main()
{       for(i=1; i<=n; i=i*2)
                sum = sum + i + func(i);
}
void func(int m)
{       for(j=1; j<=m; j++)
                Statement with O(1) complexity
}
```

3. Analyse the complexity of the following function

```
void function(int n)
{       int count=0;
        for(int i=n/2; i<=n; i++)
                for(int j=1; j<=n; j=2*j)
                        for(int k=1; k<=n; k=k*2)
                                count++;
}
```

4. Analyse the complexity of the following functions

```
function(int n)
{       if(n==1) return;
        for(i=1; i<=n; i++)
                for(int j=1; j<=n; j++)
                {       printf("*");
                        break;
                }
}
```

```
void function(int n)
{       int i=1; s=1;
        while(s<=n)
        {       i++;
                s+=i;
                printf("*");
        }
}
```

5. Express the return value of the function "mystery" in ɵ notation

```
int mystery(int n)
{
        int j=0,total=0;
        for (int i=1;j<=n;i++)
        {
                ++total;
                j+=2*i;
        }
        return total;
}
```

6. Consider the following C function

```
int check(int n)
{
    int i,j;
    for (i=1;i<=n;i++)
    {
        for (j=1;j<n;j+=i)
        {
            printf("%d",i+j);
        }
    }
}
```

   Find the time complexity of **check** in terms of ɵ notation

7. Write an algorithm to find sum of elements of an array. Find time and space complexity.
8. Write an algorithm to find $n^{th}$ Fibonacci number. Find time and space complexity.
9. Write an algorithm to find $2^k$ using recursion. Find time and space complexity
10. Write an algorithm to return largest element from an array. Find time and space complexity
11. Write an algorithm to find factorial of a number using recursion. Find time and space complexity
12. Write an algorithm for Bubble sort. Find time and space complexity
13. Write an algorithm for Insertion sort. Find time and space complexity
14. Write an algorithm for Selection sort. Find time and space complexity

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

- **Example:**

1. Determine the Best case and Worst case time complexity of the following function

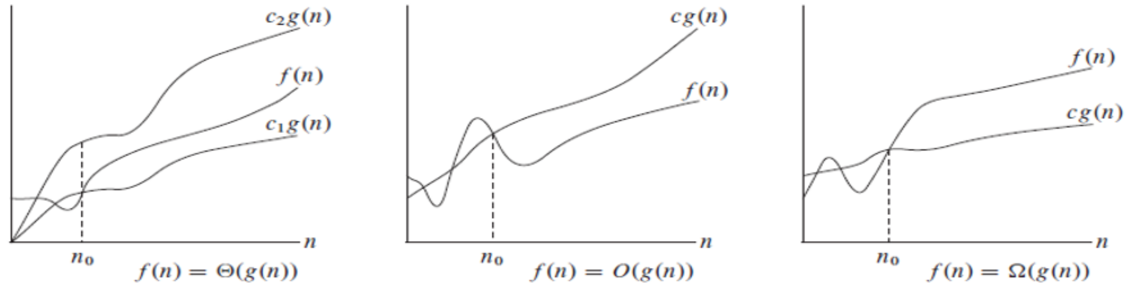
```
void fun(int n, int arr[])
{
        int i=0; j=0;
        for(;i<n;++i)
                while(j<n && arr[i]< arr[j])
                        j++;
}
```

- **Asymptotic Notations**
- It is the mathematical notations to represent frequency count. 5 types of asymptotic notations

1. **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

2. **Omega (Ω)**
   - The function $f(n) = \Omega(g(n))$ 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

3. **Theta (Θ)**
   - The function $f(n) = \Theta(g(n))$ 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).




4. **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 $0 \leq f(n) < c\, g(n)$ for all $n \geq n_0$
   - 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

5. **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 $f(n) > c\, g(n) \geq 0$ for all $n \geq n_0$
   - 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
      $3n + 2 \leq 4\, n$ for all $n \geq 2$
      Here f(n)= 3n + 2 g(n)=n c=4 $n_0$=2
      Therefore 3n + 2= **O(n)**
   b) f(n) = $4n^3 + 2n + 3$
      $4n^3 + 2n + 3 \leq 5\, n^3$ for all $n \geq 2$
      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) = $2^{n+1}$
      $2^{n+1} \leq 2\, 2^n$ for all $n \geq 1$
      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) = $2^n + 6n^2 + 3n$
      $2^n + 6n^2 + 3n \leq 7\, 2^n$ for all $n \geq 5$
      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 $2^{2n} = O(2^n)$?
   $2^{2n} \leq c\, 2^n$

$2^n \le c$

There is no value for c and $n_0$ that can make this true.

Therefore $2^{2n}$ != $\mathbf{O(2^n)}$

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. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is $2^n$ on the same machine?

5. 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) $f(n) = 10\ n^2 + 4n + 2$

6. 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) $f(n) = 27n^2 + 16$

7. Let f(n) and g(n) be asymptotically nonnegative functions. Using basic definition of Θ notation prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$

   $f(n) \geq 0 \qquad g(n) \geq 0$
   
   $f(n) + g(n) \geq f(n)$
   
   $f(n) + g(n) \geq g(n)$
   
   From the above two equations we can write
   
   $f(n) + g(n) \geq \max(f(n), g(n))$
   
   $\max(f(n), g(n)) \leq 1. (f(n) + g(n))$ where c=1
   
   $\mathbf{\max(f(n), g(n)) = O (f(n) + g(n))}$

   $\max(f(n), g(n)) \geq f(n)$
   
   $\max(f(n), g(n)) \geq g(n)$
   
   Add the above 2 equations
   
   $2. \max(f(n), g(n)) \geq f(n) + g(n)$
   
   $\max(f(n), g(n)) \geq (1/2)(f(n) + g(n))$ where c=(1/2)
   
   $\mathbf{\max(f(n), g(n)) = \Omega (f(n) + g(n))}$
   
   Now we can conclude that $\mathbf{(1/2)(f(n) + g(n)) \leq \max(f(n), g(n)) \leq 1. (f(n) + g(n))}$
   
   Here $c_1=(1/2)$, $c_2=1$
   
   Therefore $\mathbf{\max(f(n), g(n)) = \Theta (f(n) + g(n))}$

8. Show that for any real constants a and b where b>0, $(n+a)^b = \mathbf{\Theta} (n^b)$

   $n + a \leq 2 n$ for all $n \geq |a|$
   
   $n + a \geq (1/2) n$ for all $n \geq 2 |a|$
   
   Combine above two equations
   
   $(1/2) n \leq n + a \leq 2 n$ for all $n \geq 2 |a|$
   
   $(1/2)^b n^b \leq (n + a)^b \leq 2^b n^b$ for all $n \geq 2 |a|$
   
   Here $c_1=(1/2)^b$, $c_2=2^b$ and $n_0 \geq 2 |a|$
   
   Therefore $(n+a)^b = \mathbf{\Theta (n^b)}$

9. Let f(n) = 7n + 8 and g(n) = n. Is f(n)=o(g(n))?

   $f(n) < c. g(n)$ for all $n \geq n_0$
   
   $7n + 8 < c.n$ for all $n \geq n_0$
   
   If c = 8, then $n_0=9$
   
   If c = 1, then there is no $n_0$ value.
   
   As per o definition, for every c there should be corresponding $n_0$
   
   Therefore **f(n)!=o(g(n))**

10. Let f(n) = 7n + 8 and $g(n^2) = n$. Is f(n)=o(g(n))?

    $f(n) < c. g(n)$ for all $n \geq n_0$
    
    $7n + 8 < c. n^2$ for all $n \geq n_0$
    
    If c = 1, then $n_0=9$
    
    If c = 8, then $n_0=2$
    
    If c = 100, then $n_0=3$
    
    If c = 1/100, then $n_0=800$
    
    Now we can assume that for every c there is an $n_0$
    
    Therefore **f(n)=o(g(n))**

11. Theorem: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots\ldots\ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = O(n^m)$
12. Theorem: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots\ldots\ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \mathbf{\Omega}(n^m)$
13. Theorem: If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots\ldots\ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \mathbf{\Theta}(n^m)$

- **Properties of Asymptotic Notations**
  - Reflexivity
    1. f(n) = **O**(f(n))
    2. f(n) = **Ω**(f(n))
    3. f(n) = **Θ**(f(n))
  - Symmetry
    1. f(n) = **Θ**(g(n)) if and only if g(n) = **Θ**(f(n)
  - Transpose Symmetry
    1. f(n) = **O**(g(n)) if and only if g(n) = **Ω**(f(n))
    2. f(n) = **o**(g(n)) if and only if g(n) = **ω**(f(n))
  - Transitivity
    1. f(n) = **O**(g(n)) and g(n) = **O**(h(n)) imply f(n) = **O**(h(n))
    2. f(n) = **Ω**(g(n)) and g(n) = **Ω**(h(n)) imply f(n) = **Ω**(h(n))
    3. f(n) = **Θ**(g(n)) and g(n) = **Θ**(h(n)) imply f(n) = **Θ**(h(n))
    4. f(n) = **o**(g(n)) and g(n) = **o**(h(n)) imply f(n) = **o**(h(n))
    5. f(n) = **ω**(g(n)) and g(n) = **ω**(h(n)) imply f(n) = **ω**(h(n))

- **Common Complexity Functions**
  - Constant Time
    1. 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.
    2. Computational time is constant
    3. Eg: O(1)
  - Logarithmic Time
    1. An algorithm is said to be logarithmic time if f(n) = O(log n)
  - Linear Time
    1. If f(n) = O(n), then the algorithm is said to be linear time .
  - Quadratic Time
    1. If f(n) = O($n^2$), then the algorithm is said to be quadratic time .
  - Polynomial Time
    1. If f(n) = O($n^k$), then the algorithm is said to be polynomial time .
  - Exponential Time
    1. If f(n) = O($2^n$), then the algorithm is said to be exponential time .
  - Factorial Time
    1. If f(n) = O(n!), then the algorithm is said to be factorial time

- **Running Time Comparison (Classifying functions by their asymptotic growth rate)**
  - 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 log n) < O($n^k$) < O($2^n$) < O(n!)**

- **Recurrence Relations**
  - A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.
  - There are several methods for solving recurrence relation
    1. **Iteration Method**
    2. **Recursion tree Method**
    3. **Substitution Method**
    4. **Master's Method**
  - **Recursion Tree Method**
    - It is the pictorial representation of iteration method, which is in the form of a tree.

  - **Master's Method**

    $T(n) = aT(\frac{n}{b}) + \theta(n^k log^p n)$

    $a \geq 1$ $b>1$ $k \geq 0$ and p is a real number

    1. If $a>b^k$ then $T(n) = \theta(n^{(\log_b a)})$
    2. If $a=b^k$
       a. If p>-1 then $T(n)= \theta(n^{(\log_b a)} \log^{p+1}(n))$
       b. If p=-1 then $T(n)= \theta(n^{(\log_b a)} \log(\log n))$
       c. If p<-1 then $T(n)= \theta(n^{(\log_b a)})$
    3. If $a<b^k$
       a. If $p \geq 0$, then $T(n)= \theta(n^k \log^p(n))$
       b. If p<0, then $T(n)= O(n^k)$

  - Solve the following recurrence relation using Iteration method.
    1. T(n) = 1 + T(n-1)

       $T(n) = 1 + T(n-1)$
       $= 1 + 1 + T(n-2) = 2 + T(n-2)$
       $= 2 + 1 + T(n-3) = 3 + T(n-3)$
       . . . .
       $= k + T(n-k)$ $k^{th}$ term
       Assume n-k = 1 ➔ k = n-1
       $T(n) = n-1 + T(n-(n-1))$
       $= n-1 + T(1) = O(n) + O(1)$ = **O(n)**
    2. $T(2^k) = 3\,T(2^{k-1}) + 1$
       T(1) = 1
    3. T(n) = T(n/3) + n
    4. T(n) = 3 T(n/4) + n
    5. T(n) = T(n/2) + 1
    6. $T(n) = 4\,T(n/2) + n^2$
    7. T(n) = 4 T(n/3) + n
    8. T(n) =2T(n/2)+2 if n>2
       =1 if n=2
    9. T(n) = 2 T(n/2) + n T(1)=1
    10. T(n) = 2 if n=1
        T(n) = 2T(n/2) + 2n + 3 Otherwise
    11. $T(2^k) = 3T(2^{k-1}) + 1$ T(1) = 1

  - Solve the following recurrence relation using Recursion Tree method.
    1. $T(n) = 2\,T(n/2) + n^2$
    2. T(1) = 1
       $T(n) = 3\,T(n/4) + cn^2$

3. $T(n) = 3T(n/4) + n^2$
4. $T(n) = 2\,T(n/10) + T(9n/10) + n$
   Assume constant time for small value of n
5. $T(n) = 3\,T(n/3) + c\,n$
6. $T(n) = 4\,T(n/2) + n$
7. $T(n) = T(n/3) + T(2n/3) + n$
8. $T(n) = 2\,T(n-1) + c$
9. $T(n) = 8\,T(n/2) + n^2$
10. $T(n) = 3T(n/2) + n$
11. $T(n) = T(n/2) + n^2$

- Solve the following recurrence relation using Master's method.

1. $T(n) = 3\,T(n/2) + n^2$
2. $T(n) = 4\,T(n/2) + n^2$
3. $T(n) = T(n/2) + n^2$
4. $T(n) = 2^n\,T(n/2) + n^n$
5. $T(n) = 16\,T(n/4) + n$
6. $T(n) = 2\,T(n/2) + n \log n$
7. $T(n) = 2\,T(n/2) + n/\log n$
8. $T(n) = 2\,T(n/4) + n^{0.51}$
9. $T(n) = 0.5\,T(n/2) + 1/n$
10. $T(n) = 6\,T(n/3) + n^2 \log n$
11. $T(n) = 64\,T(n/8) - n^2 \log n$
12. $T(n) = 7\,T(n/3) - n^2$
13. $T(n) = 4\,T(n/2) + \log n$
14. $T(n) = \sqrt{2}\,T(n/2) + \log n$
15. $T(n) = 2\,T(n/2) + \sqrt{n}$
16. $T(n) = 3\,T(n/2) + n$
17. $T(n) = 3\,T(n/3) + \sqrt{n}$
18. $T(n) = 4\,T(n/2) + c\,n$
19. $T(n) = 9\,T(n/3) + n$
20. $T(n) = T(2n/3) + 1$
21. $T(n) = 3\,T(n/4) + n \log n$
22. $T(n) = 2\,T(n/2) + n \log n$
23. $T(n) = 8\,T(n/2) + \Theta(n^2)$
24. $T(n) = 7\,T(n/2) + \Theta(n^2)$
25. $T(n) = 2\,T(n/4) + 1$
26. $T(n) = 2\,T(n/4) + \sqrt{n}$
27. $T(n) = 2\,T(n/4) + n$
28. $T(n) = 2\,T(n/4) + n^2$
29. $T(n) = T(n/2) + \Theta(1)$
30. $T(n) = 4\,T(n/2) + n^2 \log n$

- **Arithmetic Progression**
  - Series: a , a+d , a+2d , a+3d . . . . . . . . . . a+(n-1)d
  - $n^{th}$ term: a+(n-1)d
  - sum of first n terms:

$$S_n = \frac{n}{2}[2a + (n-1)d] \quad \text{or} \quad S_n = \frac{n}{2}[T_1 + T_n] \quad \text{or} \quad S_n = n \times (\text{middle term}).$$

- **Geometric progression**
  - Series: a , ar , $ar^2$ , $ar^3$ . . . . . . . . . . $ar^{n-1}$
  - $n^{th}$ term: $ar^{n-1}$
  - sum of first n terms:

$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

  - sum of infinite terms:

$$\sum_{k=0}^{\infty}(ar^k) = a\left(\frac{1}{1-r}\right)$$