//**DSAPatterns**//
---------------------------------------------------------------------------------------------------------------------------------------------

DATA STRUCTURES:

1.Linked List
2.Matrix/Grid
3.Stack
4.Arrays
5.Hash
6.Heap
7.Graph
8.String
9.Tree

ALOGORITHMS:

1.Pattern searching
2.Divide and Conquer
3.Searching
4.Sorting
5.Bitwise
6.Greedy
7.Recursion
8.Backtracking
9.Mathematical
10.Dynamic Programming

---------------------------------------------------------------------------------------------------------------------------------------------
//DATA STRUCTURES://
---------------------------------------------------------------------------------------------------------------------------------------------

1.LINKED LIST:

 1).Traversal
 2).Addition/Substraction
 3).Fast Slow Pointer
 4).Double Linked List
 5).Override Value
 6).Monotonic stack
 7).BFS/DFS
 8)Design
    1.Circular Queue
    2.LRU Cache
    3.FIFO Cache

>>>[1.1]
Traversal:-----------------------------------------------------------------------------------------------

```
 function traverseLinkedList(head):
    current = head
    while current is not null:
        // Process the data in the current node
        print(current.data)
        // Move to the next node
```

```
        current = current.next
```

>>>[1.2]
Addition/Substraction:------------------------------------------------------------------------------------------

pseudocode for performing addition and subtraction on linked lists, where each node represents a single digit of a number.

Addition of Two Linked Lists

```
Function addLinkedLists(list1, list2):
    Initialize carry to 0
    Initialize resultList to an empty list
    Initialize pointers p1 to head of list1 and p2 to head of list2

    While p1 is not null or p2 is not null:
        Set value1 to p1's value if p1 is not null, otherwise 0
        Set value2 to p2's value if p2 is not null, otherwise 0

        Set sum to value1 + value2 + carry
        Update carry to sum // 10
        Append sum % 10 to resultList

        Move p1 to p1's next node if p1 is not null
        Move p2 to p2's next node if p2 is not null

    If carry is not 0:
        Append carry to resultList

    Return resultList
```

------------------------------------------------------------------------------------------------------------------

Subtraction of Two Linked Lists

```
Function subtractLinkedLists(list1, list2):
    Initialize borrow to 0
    Initialize resultList to an empty list
    Initialize pointers p1 to head of list1 and p2 to head of list2

    While p1 is not null or p2 is not null:
        Set value1 to p1's value if p1 is not null, otherwise 0
        Set value2 to p2's value if p2 is not null, otherwise 0

        Set diff to value1 - value2 - borrow
        If diff is less than 0:
            Set diff to diff + 10
            Set borrow to 1
        Else:
            Set borrow to 0

        Append diff to resultList

        Move p1 to p1's next node if p1 is not null
        Move p2 to p2's next node if p2 is not null

    Remove leading zeros from resultList if any

    Return resultList
```

```
>>>[1.3] Fast Slow
Pointer:--------------------------------------------------------------------------------
---------------

function hasCycle(head):
    if head is null:
        return false

    slowPointer = head
    fastPointer = head

    while fastPointer is not null and fastPointer.next is not null:
        slowPointer = slowPointer.next
        fastPointer = fastPointer.next.next

        if slowPointer == fastPointer:
            return true

    return false

>>>[1.4] Double Linked
List:-----------------------------------------------------------------------------------
------------

class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node

    def prepend(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.head.prev = new_node
            new_node.next = self.head
            self.head = new_node

    def delete(self, data):
        current = self.head
        while current:
            if current.data == data:
```

```
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next
                if current.next:
                    current.next.prev = current.prev
                else:
                    self.tail = current.prev
                return
            current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print()

dll = DoublyLinkedList()
dll.append(1)
dll.append(2)
dll.prepend(0)
dll.display()
dll.delete(1)
dll.display()
```

>>>[1.5] Override
Value:-----------------------------------------------------------------------------------------------

```
BEGIN OverrideValue
  INPUT originalValue
  INPUT newValue
  INPUT overrideCondition

  IF overrideCondition IS TRUE THEN
    OUTPUT newValue
  ELSE
    OUTPUT originalValue
  END IF
END OverrideValue
```

>>>[1.6] Monotonic
Stack:----------------------------------------------------------------------------------------------

Pseudocode for implementing a monotonic stack pattern:

```
function monotonicStack(arr):
    stack = empty stack
    result = empty list

    for each element in arr:
        while stack is not empty and stack.top() > element:
            stack.pop()
        stack.push(element)
        result.append(stack.top())
```

```
    return result

>>>[1.7]
BFS/DFS:--------------------------------------------------------------------------------------------------------

BFS(graph, start_vertex):
    # Initialize a queue and enqueue the start vertex
    queue = empty queue
    enqueue(queue, start_vertex)

    # Initialize a set to keep track of visited vertices
    visited = empty set
    add start_vertex to visited

    # Loop until the queue is empty
    while queue is not empty:
        # Dequeue a vertex from the queue
        current_vertex = dequeue(queue)

        # Process the current vertex (e.g., print it or store it in a result list)
        process(current_vertex)

        # Get all adjacent vertices of the current vertex
        for each neighbor in neighbors(current_vertex, graph):
            # If the neighbor has not been visited
            if neighbor is not in visited:
                # Mark the neighbor as visited
                add neighbor to visited
                # Enqueue the neighbor
                enqueue(queue, neighbor)
----------------------------------------------------------------------------------------------------------
BFS(tree):
    if tree is empty:
        return

    create an empty queue Q
    enqueue the root node of the tree onto Q

    while Q is not empty:
        current_node = dequeue Q
        process(current_node)

        for each child in current_node's children:
            enqueue child onto Q
----------------------------------------------------------------------------------------------------------
DFS(G, v):
    initialize Stack S
    push v to S
    while S is not empty:
        u = pop S
        if u is not visited:
            mark u as visited
            for each neighbor w of u:
                if w is not visited:
                    push w to S
----------------------------------------------------------------------------------
```

--------------------
```
DFS(node):
    if node is null:
        return
    visit(node)
    for each child in node.children:
        DFS(child)
visit(node):
    # Define what you want to do when visiting the node
    print(node.value)  # Example: print the node's value
```

>>>[1.8]
Design:-------------------------------------------------------------------------------------------------------
 1).Circular Queue

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = self.rear = -1

    def enqueue(self, data):
        if ((self.rear + 1) % self.size == self.front):
            print("Queue is Full")
        elif (self.front == -1):  # If queue is initially empty
            self.front = self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1):
            print("Queue is Empty")
        elif (self.front == self.rear):  # If there's only one element in the queue
            temp = self.queue[self.front]
            self.front = self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp

    def display(self):
        if(self.front == -1):
            print("Queue is Empty")
        elif(self.rear >= self.front):
            print("Queue elements:", end = " ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end = " ")
            print()
        else:
            print("Queue elements:", end = " ")
            for i in range(self.front, self.size):
                print(self.queue[i], end = " ")
            for i in range(0, self.rear + 1):
                print(self.queue[i], end = " ")
            print()
```

------------------------------------------------------------------------------------------------------
 2).LRU Cache

```
// LRU Cache pseudocode
Class LRUCache:
    // Initialize the cache with a given capacity
    Function __init__(capacity):
        self.capacity = capacity
        self.cache = {}         // Dictionary to store key-value pairs
        self.order = []         // List to track the order of access

    // Get the value for a given key
    Function get(key):
        If key is not in cache:
            Return -1
        // Move the accessed key to the end to mark it as recently used
        self.order.remove(key)
        self.order.append(key)
        Return self.cache[key]

    // Put a key-value pair in the cache
    Function put(key, value):
        If key in cache:
            // Update the value and move the key to the end
            self.cache[key] = value
            self.order.remove(key)
        Else:
            // If cache is at capacity, remove the least recently used item
            If length(self.cache) == self.capacity:
                lru_key = self.order.pop(0)
                del self.cache[lru_key]
            // Add the new key-value pair
            self.cache[key] = value

        // Add the key to the end to mark it as recently used
        self.order.append(key)

// Example usage:
cache = LRUCache(2)
cache.put(1, 1)
cache.put(2, 2)
cache.get(1)    // returns 1
cache.put(3, 3) // evicts key 2
cache.get(2)    // returns -1 (not found)
cache.put(4, 4) // evicts key 1
cache.get(1)    // returns -1 (not found)
cache.get(3)    // returns 3
cache.get(4)    // returns 4
```

------------------------------------------------------------------------------------------------------
 3.FIFO Cache

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class FIFO_Cache:
```

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.head = None
        self.tail = None

    def get(self, key):
        if key in self.cache:
            return self.cache[key].value
        return -1

    def put(self, key, value):
        if key in self.cache:
            node = self.cache[key]
            node.value = value
            return

        if len(self.cache) == self.capacity:
            del self.cache[self.head.key]
            self.head = self.head.next
            if self.head:
                self.head.prev = None

        new_node = Node(key, value)
        self.cache[key] = new_node
        if not self.head:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
```
------------------------------------------------------------------------------------
-------------------------------------------------------------------------

2.Matrix/Grid:

1. Traversing a Matrix (4-Directions):

```
FUNCTION traverse_matrix(matrix, rows, cols):
    DIRECTIONS = [(0,1), (1,0), (0,-1), (-1,0)]  # Right, Down, Left, Up

    FOR i FROM 0 TO rows-1:
        FOR j FROM 0 TO cols-1:
            PRINT matrix[i][j]  # Process current cell

            FOR (dx, dy) IN DIRECTIONS:
                new_x = i + dx
                new_y = j + dy
                IF 0 ≤ new_x < rows AND 0 ≤ new_y < cols:  # Stay within bounds
                    PRINT matrix[new_x][new_y]  # Process neighbor
```

2. DFS in a Matrix (Connected Components)

```
FUNCTION dfs(matrix, x, y, rows, cols, visited):
    IF x < 0 OR y < 0 OR x ≥ rows OR y ≥ cols OR visited[x][y]:
        RETURN

    visited[x][y] = True  # Mark as visited

    DIRECTIONS = [(0,1), (1,0), (0,-1), (-1,0)]  # Right, Down, Left, Up
```

```
    FOR (dx, dy) IN DIRECTIONS:
        dfs(matrix, x + dx, y + dy, rows, cols, visited)
```

3. BFS in a Matrix (Shortest Path)

```
FUNCTION bfs(matrix, start_x, start_y, rows, cols):
    QUEUE = [(start_x, start_y, 0)]  # (x, y, steps)
    VISITED = Set()
    VISITED.add((start_x, start_y))

    DIRECTIONS = [(0,1), (1,0), (0,-1), (-1,0)]  # Right, Down, Left, Up

    WHILE QUEUE IS NOT EMPTY:
        (x, y, steps) = QUEUE.pop(0)  # Dequeue

        FOR (dx, dy) IN DIRECTIONS:
            new_x = x + dx
            new_y = y + dy

            IF 0 ≤ new_x < rows AND 0 ≤ new_y < cols AND (new_x, new_y) NOT IN
VISITED:
                VISITED.add((new_x, new_y))
                QUEUE.append((new_x, new_y, steps + 1))

    RETURN "Shortest path found"
```

4. Finding the Number of Islands (DFS/BFS):

```
FUNCTION count_islands(grid, rows, cols):
    CREATE visited = [[False] * cols FOR _ IN range(rows)]
    COUNT = 0

    FUNCTION dfs(x, y):
        IF x < 0 OR y < 0 OR x ≥ rows OR y ≥ cols OR grid[x][y] == 0 OR visited[x]
[y]:
            RETURN
        visited[x][y] = True

        FOR (dx, dy) IN [(0,1), (1,0), (0,-1), (-1,0)]:
            dfs(x + dx, y + dy)

    FOR i FROM 0 TO rows-1:
        FOR j FROM 0 TO cols-1:
            IF grid[i][j] == 1 AND NOT visited[i][j]:
                COUNT += 1
                dfs(i, j)

    RETURN COUNT
```

5. Matrix Rotation (90 Degrees Clockwise):

```
FUNCTION rotate_matrix(matrix, N):
    FOR i FROM 0 TO N-1:
        FOR j FROM i TO N-1:
            SWAP matrix[i][j] WITH matrix[j][i]  # Transpose

    FOR row IN matrix:
        REVERSE row  # Reverse each row
```

------------------------------------------------------------------------------------
----------------------------------------------------------------------------
3.STACK:

  1).Nearest Greater/Smaller on Left/Right
  2).Stock span problem
  3).Histogram Problems
  4).Stack/Queue Implementation
  5).Traversal
  6).Parenthesis Checker
  7).Infix/Prefix/Postfix Conversion
  8).Tower of Hanoi Problem

>>>[3.1]Nearest Greater/Smaller on
Left/Right:-----------------------------------------------------------------------
-------------------------------

case1:
```
function nearestGreaterOnLeft(arr):
    stack = empty stack
    result = []

    for element in arr:
        while stack is not empty and stack.peek() <= element:
            stack.pop()

        if stack is empty:
            result.append(-1)
        else:
            result.append(stack.peek())

        stack.push(element)

    return result
```
case2:
```
function nearestSmallerOnLeft(arr):
    stack = empty stack
    result = []

    for element in arr:
        while stack is not empty and stack.peek() >= element:
            stack.pop()

        if stack is empty:
            result.append(-1)
        else:
            result.append(stack.peek())

        stack.push(element)

    return result
```

case3:
```
function nearestGreaterOnRight(arr):
    stack = empty stack
    result = create list of size arr with -1

    for i from len(arr) - 1 to 0:
```

```
        while stack is not empty and stack.peek() <= arr[i]:
            stack.pop()

        if stack is not empty:
            result[i] = stack.peek()

        stack.push(arr[i])

    return result
```

case4:

```
function nearestSmallerOnRight(arr):
    stack = empty stack
    result = create list of size arr with -1

    for i from len(arr) - 1 to 0:
        while stack is not empty and stack.peek() >= arr[i]:
            stack.pop()

        if stack is not empty:
            result[i] = stack.peek()

        stack.push(arr[i])

    return result
```

>>>[3.2] Stock span
problem:-----------------------------------------------------------------------------------------------------

```
function calculateSpan(prices):
    # Initialize an empty stack and a list to store spans
    stack = empty stack
    span = array of size prices.length

    # Iterate through each price
    for i from 0 to prices.length - 1:
        # Pop elements from the stack while the stack is not empty and the top of
the stack is less than or equal to the current price
        while stack is not empty and prices[stack.peek()] <= prices[i]:
            stack.pop()

        # If the stack becomes empty, then the current price is greater than all
prices before it
        if stack is empty:
            span[i] = i + 1
        else:
            span[i] = i - stack.peek()

        # Push the current index onto the stack
        stack.push(i)

    return span
```

>>>[3.3] Histogram
Problems:-----------------------------------------------------------------------------------------------------

```
// Function to find the largest rectangular area in a histogram
```

```
function findLargestRectangle(histogram):
    // Initialize maxArea to 0
    maxArea = 0

    // Initialize an empty stack
    stack = empty stack

    // Initialize index to 0
    index = 0

    // Loop through each bar in the histogram
    while index < length(histogram):
        // If the current bar is higher than the bar at the stack's top
        if stack is empty OR histogram[index] >= histogram[top of stack]:
            // Push the current index to the stack
            push stack with index
            // Move to the next index
            index = index + 1
        else:
            // Pop the top of the stack
            top = pop from stack
            // Calculate the area with the popped bar as the smallest (or minimum
height) bar 'h'
            area = histogram[top] * (index - top of stack - 1 if stack is not empty
else index)
            // Update maxArea, if needed
            maxArea = max(maxArea, area)

    // Now, pop the remaining bars from stack and calculate area with every popped
bar
    while stack is not empty:
        top = pop from stack
        area = histogram[top] * (index - top of stack - 1 if stack is not empty
else index)
        maxArea = max(maxArea, area)

    return maxArea

case:1. Brute Force Method

function bruteForceLargestRectangle(histogram):
    maxArea = 0
    for i from 0 to length(histogram) - 1:
        minHeight = histogram[i]
        for j from i to length(histogram) - 1:
            minHeight = min(minHeight, histogram[j])
            area = minHeight * (j - i + 1)
            maxArea = max(maxArea, area)
    return maxArea

case:2. Divide and Conquer

function divideAndConquer(histogram, start, end):
    if start > end:
        return 0

    minIndex = findMinIndex(histogram, start, end)
    maxArea = histogram[minIndex] * (end - start + 1)
```

```
    leftArea = divideAndConquer(histogram, start, minIndex - 1)
    rightArea = divideAndConquer(histogram, minIndex + 1, end)

    return max(maxArea, leftArea, rightArea)

function findMinIndex(histogram, start, end):
    minIndex = start
    for i from start to end:
        if histogram[i] < histogram[minIndex]:
            minIndex = i
    return minIndex

case:3. Segment Tree

function buildSegmentTree(histogram, tree, start, end, node):
    if start == end:
        tree[node] = histogram[start]
    else:
        mid = (start + end) / 2
        buildSegmentTree(histogram, tree, start, mid, 2 * node + 1)
        buildSegmentTree(histogram, tree, mid + 1, end, 2 * node + 2)
        tree[node] = min(tree[2 * node + 1], tree[2 * node + 2])

function querySegmentTree(tree, start, end, l, r, node):
    if l > end OR r < start:
        return infinity
    if l <= start AND r >= end:
        return tree[node]
    mid = (start + end) / 2
    return min(querySegmentTree(tree, start, mid, l, r, 2 * node + 1),
               querySegmentTree(tree, mid + 1, end, l, r, 2 * node + 2))

function largestRectangleUsingSegmentTree(histogram):
    n = length(histogram)
    tree = array of size 4 * n
    buildSegmentTree(histogram, tree, 0, n - 1, 0)
    return largestRectangleUtil(histogram, tree, 0, n - 1)

function largestRectangleUtil(histogram, tree, start, end):
    if start > end:
        return 0
    minIndex = querySegmentTree(tree, 0, length(histogram) - 1, start, end, 0)
    maxArea = histogram[minIndex] * (end - start + 1)
    leftArea = largestRectangleUtil(histogram, tree, start, minIndex - 1)
    rightArea = largestRectangleUtil(histogram, tree, minIndex + 1, end)
    return max(maxArea, leftArea, rightArea)
```

>>>[3.4]Stack/Queue
Implementation:-----------------------------------------------------------------------------------------------------

Array-Based Queue Implementation
(a) Initialize Queue

```
Function InitializeQueue(size):
    Create an array of size 'size'
    Set front = -1
    Set rear = -1
```

(b) Enqueue Operation

```
Function Enqueue(queue, rear, size, data):
```

```
    If rear == size - 1:
        Print "Queue Overflow"
        Return
    If front == -1:
        front = 0  # Initialize front on the first enqueue
    rear = rear + 1
    queue[rear] = data
(c) Dequeue Operation
Function Dequeue(queue, front, rear):
    If front == -1 or front > rear:
        Print "Queue Underflow"
        Return NULL
    data = queue[front]
    front = front + 1
    If front > rear:  # Reset the queue when it's empty
        front = -1
        rear = -1
    Return data
(d) Peek Operation
Function Peek(queue, front):
    If front == -1:
        Print "Queue is empty"
        Return NULL
    Return queue[front]
(e) isEmpty Operation
Function isEmpty(front):
    Return front == -1

Circular Queue Implementation

(a) Enqueue Operation in Circular Queue
Function EnqueueCircular(queue, front, rear, size, data):
    nextRear = (rear + 1) % size
    If nextRear == front:
        Print "Queue Overflow"
        Return
    rear = nextRear
    queue[rear] = data
(b) Dequeue Operation in Circular Queue
Function DequeueCircular(queue, front, rear, size):
    If front == -1:
        Print "Queue Underflow"
        Return NULL
    data = queue[front]
    front = (front + 1) % size
    If front == (rear + 1) % size:  # Reset the queue when it's empty
        front = -1
        rear = -1
    Return data

Linked-List-Based Queue Implementation

Node Structure
Structure Node:
    data
    next
(a) Initialize Queue
Function InitializeQueue():
    Set front = NULL
```

```
    Set rear = NULL
(b) Enqueue Operation
Function Enqueue(front, rear, data):
    Create newNode with data
    If rear is NULL:
        front = newNode
        rear = newNode
    Else:
        rear.next = newNode
        rear = newNode
    Return front, rear
(c) Dequeue Operation
Function Dequeue(front, rear):
    If front is NULL:
        Print "Queue Underflow"
        Return NULL
    data = front.data
    front = front.next
    If front is NULL:
        rear = NULL  # Reset rear if the queue is empty
    Return data, front, rear
(d) Peek Operation
Function Peek(front):
    If front is NULL:
        Print "Queue is empty"
        Return NULL
    Return front.data
(e) isEmpty Operation
Function isEmpty(front):
    Return front == NULL
```

Array-Based Stack Implementation

```
(a) Initialize Stack
Function InitializeStack(size):
    Create an array of size 'size'
    Set top = -1  # Indicates the stack is empty
(b) Push Operation
Function Push(stack, top, size, data):
    If top == size - 1:
        Print "Stack Overflow"
        Return
    top = top + 1
    stack[top] = data
(c) Pop Operation
Function Pop(stack, top):
    If top == -1:
        Print "Stack Underflow"
        Return NULL
    data = stack[top]
    top = top - 1
    Return data
(d) Peek Operation
Function Peek(stack, top):
    If top == -1:
        Print "Stack is empty"
        Return NULL
    Return stack[top]
```

```
(e) isEmpty Operation
Function isEmpty(top):
    Return top == -1
```

Linked-List-Based Stack Implementation

```
Node Structure
Structure Node:
    data
    next
(a) Initialize Stack
Function InitializeStack():
    Set top = NULL  # Empty stack
(b) Push Operation
Function Push(top, data):
    Create newNode with data
    newNode.next = top
    top = newNode
    Return top
(c) Pop Operation
Function Pop(top):
    If top == NULL:
        Print "Stack Underflow"
        Return NULL
    data = top.data
    top = top.next
    Return data
(d) Peek Operation
Function Peek(top):
    If top == NULL:
        Print "Stack is empty"
        Return NULL
    Return top.data
(e) isEmpty Operation
Function isEmpty(top):
    Return top == NULL
```

Circular Stack Implementation

```
(a) Push Operation (Circular Stack)
Function PushCircular(stack, top, size, data):
    nextIndex = (top + 1) % size
    If nextIndex == 0 and top == size - 1:
        Print "Stack Overflow"
        Return
    top = nextIndex
    stack[top] = data
(b) Pop Operation (Circular Stack)
Function PopCircular(stack, top, size):
    If top == -1:
        Print "Stack Underflow"
        Return NULL
    data = stack[top]
    top = (top - 1 + size) % size
    Return data
```

>>>[3.5].Traversal:-----------------------------------------------------------------------------------------------------

```
case1:
Function DFS(node):
    Create an empty stack
    Push(node)
    While stack is not empty:
        current = Pop()
        Process(current)
        For each neighbor of current:
            If neighbor is not visited:
                Push(neighbor)
case2:
Function traverseStack():
    While stack is not empty:
        element = pop()
        Process element

Function traverseStackRecursive(stack):
    If stack is empty:
        Return
    element ← stack.pop()
    Print element
    traverseStackRecursive(stack)  # Recursive call
    stack.push(element)  # Restore element back

case3:
Function BFS(node):
    Create an empty queue
    Enqueue(node)
    While queue is not empty:
        current = Dequeue()
        Process(current)
        For each neighbor of current:
            If neighbor is not visited:
                Enqueue(neighbor)
case4:
Function traverseQueue():
    While queue is not empty:
        element = dequeue()
        Process element

Function traverseQueueRecursive(queue):
    If queue is empty:
        Return
    element ← queue.dequeue()
    Print element
    traverseQueueRecursive(queue)  # Recursive call
    queue.enqueue(element)  # Restore element back

>>>[3.6].Parenthesis
Checker:----------------------------------------------------------------------------
----------------------------

Function isBalanced(expression):
    # Initialize an empty stack
    stack ← empty list

    # Dictionary to map closing brackets to their corresponding opening brackets
    matching_parenthesis ← { ')': '(', ']': '[', '}': '{' }
```

```
    # Loop through each character in the expression
    For each char in expression:
        # If it's an opening bracket, push it onto the stack
        If char is in ('(', '[', '{'):
            stack.push(char)

        # If it's a closing bracket
        Else If char is in (')', ']', '}'):
            # If stack is empty, return False (no matching opening bracket)
            If stack is empty:
                Return False

            # Pop the last opened bracket
            top ← stack.pop()

            # Check if the popped bracket matches the corresponding opening bracket
            If matching_parenthesis[char] ≠ top:
                Return False

    # If stack is empty, expression is balanced; otherwise, it's not
    Return stack is empty

# Test cases
Print isBalanced("{[()]}")  # True (Balanced)
Print isBalanced("{[(])}")  # False (Not Balanced)
Print isBalanced("{[}")     # False (Not Balanced)
```

>>>[3.7].Infix/Prefix/Postfix
Conversion:-----------------------------------------------------------------------------------------------

Infix to Postfix Conversion (Shunting Yard Algorithm):

```
Function infixToPostfix(expression):
    Create an empty stack for operators
    Create an empty list for output

    For each token in expression:
        If token is an operand:
            Append it to output

        Else If token is '(':
            Push it to stack

        Else If token is ')':
            While stack is not empty AND top of stack is not '(':
                Pop from stack and append to output
            Pop '(' from stack

        Else If token is an operator (+, -, *, /, ^):
            While stack is not empty AND precedence(top of stack) >=
precedence(token):
                Pop from stack and append to output
            Push token to stack

    While stack is not empty:
        Pop from stack and append to output

    Return output as a string
```

Infix to Prefix Conversion

```
Function infixToPrefix(expression):
    Reverse the expression
    Replace '(' with ')' and vice versa
    Convert the reversed expression to Postfix using infixToPostfix()
    Reverse the resulting postfix expression
    Return the final Prefix expression
```

Postfix to Infix Conversion

```
Function postfixToInfix(expression):
    Create an empty stack

    For each token in expression:
        If token is an operand:
            Push token to stack

        Else If token is an operator:
            Operand2 ← stack.pop()
            Operand1 ← stack.pop()
            Subexpression ← "( Operand1 token Operand2 )"
            Push Subexpression to stack

    Return the only element in the stack as the final Infix expression
```

```
Infix → Postfix :Use operator stack, output operands directly
Infix → Prefix    :Reverse infix → Convert to postfix → Reverse result
Postfix → Infix   :Use a stack, pop operands, and construct expressions
```

>>>[3.8].Tower of Hanoi
Problem :-----------------------------------------------------------------------------------------------------------

```
function TowerOfHanoi(n, source, target, auxiliary):
    if n == 1:
        print "Move disk 1 from rod", source, "to rod", target
        return
    TowerOfHanoi(n-1, source, auxiliary, target)
    print "Move disk", n, "from rod", source, "to rod", target
    TowerOfHanoi(n-1, auxiliary, target, source)
```

-----------------------------------------------------------------------------------------------------------------------------------------------------------

4.ARRAYS:

1. Traversing an Array:

```
FUNCTION traverse_array(arr, N):
    FOR i FROM 0 TO N-1:
        PRINT arr[i]  # Process each element
```

2. Searching an Element in an Array:

(a) Linear Search (Unsorted Array)

```
FUNCTION linear_search(arr, N, target):
    FOR i FROM 0 TO N-1:
```

```
        IF arr[i] == target:
            RETURN i  # Found at index i
    RETURN -1  # Not found
```

(b) Binary Search (Sorted Array):

```
FUNCTION binary_search(arr, left, right, target):
    WHILE left ≤ right:
        mid = (left + right) // 2
        IF arr[mid] == target:
            RETURN mid
        ELSE IF arr[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1
    RETURN -1  # Not found
```

3. Sorting an Array:

(a) Bubble Sort:

```
FUNCTION bubble_sort(arr, N):
    FOR i FROM 0 TO N-1:
        FOR j FROM 0 TO N-i-1:
            IF arr[j] > arr[j+1]:
                SWAP arr[j] WITH arr[j+1]
```

(b) Quick Sort:

```
FUNCTION quick_sort(arr, low, high):
    IF low < high:
        pivot_index = partition(arr, low, high)
        quick_sort(arr, low, pivot_index - 1)
        quick_sort(arr, pivot_index + 1, high)

FUNCTION partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    FOR j FROM low TO high-1:
        IF arr[j] < pivot:
            i = i + 1
            SWAP arr[i] WITH arr[j]
    SWAP arr[i+1] WITH arr[high]
    RETURN i + 1
```

4. Finding Maximum & Minimum in an Array:

```
FUNCTION find_max_min(arr, N):
    max_val = arr[0]
    min_val = arr[0]

    FOR i FROM 1 TO N-1:
        IF arr[i] > max_val:
            max_val = arr[i]
        IF arr[i] < min_val:
            min_val = arr[i]

    RETURN (max_val, min_val)
```

5. Reverse an Array:

```
FUNCTION reverse_array(arr, N):
    left = 0
    right = N - 1
    WHILE left < right:
        SWAP arr[left] WITH arr[right]
        left = left + 1
        right = right - 1
```

6. Find the Missing Number (1 to N):

(a) Using Sum Formula:

```
FUNCTION find_missing_number(arr, N):
    total_sum = N * (N + 1) / 2
    arr_sum = SUM(arr)
    RETURN total_sum - arr_sum
```

(b) Using XOR:

```
FUNCTION find_missing_xor(arr, N):
    total_xor = 0
    arr_xor = 0

    FOR i FROM 1 TO N:
        total_xor ^= i

    FOR num IN arr:
        arr_xor ^= num

    RETURN total_xor ^ arr_xor
```

7. Kadane’s Algorithm (Maximum Subarray Sum)

```
FUNCTION max_subarray_sum(arr, N):
    max_sum = -∞
    current_sum = 0

    FOR i FROM 0 TO N-1:
        current_sum = MAX(arr[i], current_sum + arr[i])
        max_sum = MAX(max_sum, current_sum)

    RETURN max_sum
```

8. Merge Two Sorted Arrays:

```
FUNCTION merge_sorted_arrays(arr1, arr2, N, M):
    i = 0, j = 0
    merged = []

    WHILE i < N AND j < M:
        IF arr1[i] < arr2[j]:
            merged.append(arr1[i])
            i = i + 1
        ELSE:
            merged.append(arr2[j])
            j = j + 1
```

```
    WHILE i < N:
        merged.append(arr1[i])
        i = i + 1

    WHILE j < M:
        merged.append(arr2[j])
        j = j + 1

    RETURN merged
```

-------------------------------------------------------------------------------------
-----------------------------------------------------------------------
5.HASH:

 1).Static Hashing
 2).Dynamic Hashing
 3).Open Hashing(Separate chaining)
 4).Closed Hashing(Open Addressing)
    1.Linear,Double,Quadratic probing

>>>[5.1].Static
Hashing :-------------------------------------------------------------------------
----------------------------

```
Function initializeHashTable(size):
    Create an array of size 'size'
    For i from 0 to size-1:
        hashTable[i] ← empty list
    Return hashTable

Function hashFunction(key, size):
    Return key MOD size  # Simple division method

Function insert(hashTable, key, size):
    index ← hashFunction(key, size)
    Append key to hashTable[index]  # Add key to linked list

Function search(hashTable, key, size):
    index ← hashFunction(key, size)
    If key exists in hashTable[index]:
        Return True  # Key found
    Return False  # Key not found

Function delete(hashTable, key, size):
    index ← hashFunction(key, size)
    If key exists in hashTable[index]:
        Remove key from hashTable[index]
        Return True  # Key deleted
    Return False  # Key not found
```

>>>[5.2].Dynamic
Hashing :-------------------------------------------------------------------------
----------------------------

Pseudocode for Extendible Hashing

```
Function initializeHashTable():
    globalDepth ← 1
    directory ← array of size 2^globalDepth (each entry points to a bucket)
```

```
    Initialize each bucket with localDepth = globalDepth

Function hashFunction(key, depth):
    Return first 'depth' bits of hash(key)

Function insert(directory, key):
    index ← hashFunction(key, globalDepth)
    bucket ← directory[index]

    If bucket is not full:
        Append key to bucket
        Return

    # If full, split the bucket
    newBucket ← createNewBucket(bucket.localDepth + 1)
    oldBucket ← bucket
    oldBucket.localDepth += 1

    If oldBucket.localDepth > globalDepth:
        globalDepth += 1
        Double directory size

    Rehash all keys in oldBucket and newBucket based on updated depth
    Update directory pointers to reflect new bucket mapping

Function search(directory, key):
    index ← hashFunction(key, globalDepth)
    bucket ← directory[index]

    If key exists in bucket:
        Return True
    Return False

Function delete(directory, key):
    index ← hashFunction(key, globalDepth)
    bucket ← directory[index]

    If key exists in bucket:
        Remove key
        Return True
    Return False

Linear Hashing (Bucket Splitting):

Function initializeHashTable():
    level ← 0
    splitPointer ← 0
    numBuckets ← 2^level
    Create 'numBuckets' empty buckets

Function hashFunction(key, level):
    Return key MOD (2^level)

Function insert(hashTable, key):
    index ← hashFunction(key, level)
    bucket ← hashTable[index]

    If bucket is not full:
        Append key to bucket
```

```
        Return

    # If full, split bucket at splitPointer
    newBucket ← createNewBucket()
    oldBucket ← hashTable[splitPointer]

    Move keys from oldBucket to correct bucket (rehashed)
    splitPointer += 1

    If splitPointer == 2^level:
        splitPointer = 0
        level += 1  # Increase hash range
```

>>>[5.3].Open Hashing(Separate chaining) :-----------------------------------------------------------------------------------------------------

```
Initialize an array called table with M buckets.
Each bucket is initially an empty linked list.

function hash(key):
    return hash_value_of(key) % M

function insert(key, value):
    bucket_index = hash(key)
    bucket = table[bucket_index]

    if key is already in the bucket:
        update the existing value
    else:
        append (key, value) to the bucket

function search(key):
    bucket_index = hash(key)
    bucket = table[bucket_index]

    for (k, v) in bucket:
        if k == key:
            return v
    return null

function delete(key):
    bucket_index = hash(key)
    bucket = table[bucket_index]

    for (k, v) in bucket:
        if k == key:
            remove (k, v) from the bucket
            return true
    return false
```

>>>[5.4].Closed Hashing(Open Addressing)
    1.Linear,Double,Quadratic probing :--------------------------------------------------------------------------------------------------------

1).Linear Probing:

```
initialize_table()
function insert(key, value):
```

```
    index = hash(key)
    while table[index] is not empty:
        index = (index + 1) % table_size
    table[index] = (key, value)

function search(key):
    index = hash(key)
    while table[index] is not empty:
        if table[index].key == key:
            return table[index].value
        index = (index + 1) % table_size
    return null
```

2).Double Hashing:

```
initialize_table()
function primary_hash(key):
    return hash(key) % table_size

function secondary_hash(key):
    return 1 + (hash(key) % (table_size - 1))

function insert(key, value):
    index = primary_hash(key)
    step_size = secondary_hash(key)
    while table[index] is not empty:
        index = (index + step_size) % table_size
    table[index] = (key, value)

function search(key):
    index = primary_hash(key)
    step_size = secondary_hash(key)
    while table[index] is not empty:
        if table[index].key == key:
            return table[index].value
        index = (index + step_size) % table_size
    return null
```

3).Quadratic Probing:

```
initialize_table()
function insert(key, value):
    index = hash(key)
    i = 1
    while table[index] is not empty:
        index = (index + i^2) % table_size
        i += 1
    table[index] = (key, value)

function search(key):
    index = hash(key)
    i = 1
    while table[index] is not empty:
        if table[index].key == key:
            return table[index].value
        index = (index + i^2) % table_size
        i += 1
    return null
```

----------------------------------------------------------------------------------
-------------------------------------------------------------------------
6.HEAP:

  1).Top K Patterns
      1.Nearest/Farthest
      2.Greatest/Closest
      3.Largest/Smallest
      4.Maximium/Minimium
      5.Expensive/Cheapest
 2).Priority+Hashing
 3).Course Schedule
 4).Median and Math
 5).Merge Arrays
 6).Sliding Window

>>>[6.1].Top K
Patterns:-------------------------------------------------------------------------
----------------------------

1. Top K Nearest/Farthest Elements

```
FUNCTION find_top_k_nearest_farthest(arr, target, K, mode):
    # mode: "nearest" -> closest K elements, "farthest" -> farthest K elements

    CREATE a max heap (for nearest) or min heap (for farthest)

    FOR each number in arr:
        CALCULATE distance = ABS(number - target)

        IF mode is "nearest":
            PUSH (distance, number) into max heap
            IF heap size > K:
                POP largest element (to maintain K smallest distances)

        ELSE IF mode is "farthest":
            PUSH (distance, number) into min heap
            IF heap size > K:
                POP smallest element (to maintain K largest distances)

    RETURN elements from heap
```

2. Top K Greatest/Closest Elements

#Approach: Sorting or Heap:

```
FUNCTION find_top_k_greatest_closest(arr, K, mode):
    # mode: "greatest" -> top K largest elements, "closest" -> smallest K elements

    IF mode is "greatest":
        SORT arr in descending order
        RETURN first K elements

    ELSE IF mode is "closest":
        SORT arr in ascending order
        RETURN first K elements
```

#Optimized Heap Approach:

```
    IF mode is "greatest":
        CREATE a min heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP smallest element

    ELSE IF mode is "closest":
        CREATE a max heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP largest element

    RETURN heap elements
```

3. Top K Largest/Smallest Elements

```
#Approach: Sorting or Heap:

FUNCTION find_top_k_largest_smallest(arr, K, mode):
    # mode: "largest" -> top K largest elements, "smallest" -> top K smallest
elements

    IF mode is "largest":
        SORT arr in descending order
        RETURN first K elements

    ELSE IF mode is "smallest":
        SORT arr in ascending order
        RETURN first K elements

#Heap Approach: (Efficient for large datasets)

    IF mode is "largest":
        CREATE a min heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP smallest element

    ELSE IF mode is "smallest":
        CREATE a max heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP largest element

    RETURN heap elements
```

4. Top K Maximum/Minimum Values

```
#Approach: Sorting or Heap:

FUNCTION find_top_k_max_min(arr, K, mode):
    # mode: "max" -> highest values, "min" -> lowest values

    IF mode is "max":
        SORT arr in descending order
```

```
        RETURN first K elements

    ELSE IF mode is "min":
        SORT arr in ascending order
        RETURN first K elements
```

```
#Heap Approach:

    IF mode is "max":
        CREATE a min heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP smallest element

    ELSE IF mode is "min":
        CREATE a max heap
        FOR each number in arr:
            PUSH number into heap
            IF heap size > K:
                POP largest element

    RETURN heap elements
```

5. Top K Most Expensive/Cheapest Items

#Approach: Sorting or Heap:

```
FUNCTION find_top_k_expensive_cheapest(prices, K, mode):
    # mode: "expensive" -> highest prices, "cheapest" -> lowest prices

    IF mode is "expensive":
        SORT prices in descending order
        RETURN first K elements

    ELSE IF mode is "cheapest":
        SORT prices in ascending order
        RETURN first K elements
```

```
#Heap Approach:

    IF mode is "expensive":
        CREATE a min heap
        FOR each price in prices:
            PUSH price into heap
            IF heap size > K:
                POP smallest element

    ELSE IF mode is "cheapest":
        CREATE a max heap
        FOR each price in prices:
            PUSH price into heap
            IF heap size > K:
                POP largest element

    RETURN heap elements
```

>>>[6.2].Priority+Hashing:-------------------------------------------------------------------------------------------------------------

1. Find Top-K Frequent Elements (Priority Queue + Hash Map)

```
FUNCTION find_top_k_frequent(arr, K):
    CREATE hash_map to store element frequencies

    # Step 1: Count frequencies
    FOR each num in arr:
        hash_map[num] += 1

    # Step 2: Use a min-heap (size K)
    CREATE min_heap

    FOR each (num, freq) in hash_map:
        PUSH (freq, num) into min_heap
        IF size of min_heap > K:
            POP smallest frequency element

    RETURN elements in min_heap
```

2. Task Scheduler (Using Priority Queue + Hashing)

```
FUNCTION min_time_to_execute_tasks(tasks, cooldown):
    CREATE hash_map to store task frequencies

    # Step 1: Count frequencies
    FOR each task in tasks:
        hash_map[task] += 1

    # Step 2: Max-Heap to process most frequent tasks first
    CREATE max_heap
    FOR each (task, freq) in hash_map:
        PUSH (-freq, task) into max_heap  # Negative frequency for max heap

    CREATE queue (for cooldown tracking)
    SET time = 0

    # Step 3: Process tasks
    WHILE max_heap is NOT empty OR queue is NOT empty:
        time += 1

        IF max_heap is NOT empty:
            POP (freq, task) from max_heap
            IF freq + 1 != 0:  # Still has occurrences left
                PUSH (task, freq + 1, current_time + cooldown) into queue

        # Check if any task in queue can be re-added to max_heap
        IF queue is NOT empty AND queue.front's cooldown is reached:
            POP from queue and PUSH into max_heap

    RETURN time
```

3. Merge K Sorted Lists (Priority Queue + Hash Map)

```
FUNCTION merge_k_sorted_lists(lists):
    CREATE min_heap

    # Step 1: Push first elements of each list into min_heap
    FOR each list in lists:
```

```
        IF list is NOT empty:
            PUSH (list[0], list_index, element_index) into min_heap

    CREATE merged_list

    # Step 2: Extract elements in sorted order
    WHILE min_heap is NOT empty:
        (val, list_idx, elem_idx) = POP from min_heap
        APPEND val to merged_list

        # If more elements exist in the same list, add next element
        IF elem_idx + 1 < length of lists[list_idx]:
            PUSH (lists[list_idx][elem_idx + 1], list_idx, elem_idx + 1) into
min_heap

    RETURN merged_list
```

>>>[6.3].Course
Schedule:--------------------------------------------------------------------------------------------------------

Pseudocode for Course Schedule (Topological Sorting using BFS & DFS):

```
function canFinishCourses(numCourses, prerequisites):
    # Create an adjacency list to represent the graph
    graph = [[] for _ in range(numCourses)]
    # Create an array to store the in-degrees of each course
    inDegree = [0] * numCourses

    # Build the graph and update in-degrees
    for (course, prereq) in prerequisites:
        graph[prereq].append(course)
        inDegree[course] += 1

    # Initialize a queue and add all courses with in-degree 0
    queue = []
    for course in range(numCourses):
        if inDegree[course] == 0:
            queue.append(course)

    # Process the queue
    while queue:
        course = queue.pop(0)
        for neighbor in graph[course]:
            inDegree[neighbor] -= 1
            if inDegree[neighbor] == 0:
                queue.append(neighbor)

    # Check if all courses have been taken
    for course in range(numCourses):
        if inDegree[course] != 0:
            return False

    return True
```

1. BFS (Kahn’s Algorithm) Approach

```
FUNCTION can_finish_courses(n, prerequisites):
    CREATE adjacency_list as a graph
    CREATE in_degree array initialized to 0
```

```
    # Step 1: Build graph & count in-degree
    FOR each (course, prereq) in prerequisites:
        ADD course to adjacency_list[prereq]
        INCREMENT in_degree[course]

    # Step 2: Push all courses with no prerequisites into the queue
    CREATE queue
    FOR course in range(n):
        IF in_degree[course] == 0:
            ENQUEUE course into queue

    # Step 3: Process courses
    SET count = 0  # Count of processed courses
    WHILE queue is NOT empty:
        current_course = DEQUEUE queue
        INCREMENT count

        FOR next_course in adjacency_list[current_course]:
            DECREMENT in_degree[next_course]
            IF in_degree[next_course] == 0:
                ENQUEUE next_course

    # If all courses are processed, return True; else, return False
    RETURN count == n
```

2. DFS Approach (Cycle Detection using Recursion)

```
FUNCTION can_finish_courses(n, prerequisites):
    CREATE adjacency_list as a graph
    CREATE visited array of size n initialized to 0

    # Step 1: Build the graph
    FOR each (course, prereq) in prerequisites:
        ADD course to adjacency_list[prereq]

    # Step 2: DFS function for cycle detection
    FUNCTION dfs(course):
        IF visited[course] == 1:  # Cycle detected
            RETURN False
        IF visited[course] == 2:  # Already processed
            RETURN True

        MARK visited[course] as 1 (Visiting)

        FOR next_course in adjacency_list[course]:
            IF NOT dfs(next_course):
                RETURN False  # Cycle found

        MARK visited[course] as 2 (Processed)
        RETURN True

    # Step 3: Run DFS for all unvisited nodes
    FOR course in range(n):
        IF visited[course] == 0:
            IF NOT dfs(course):
                RETURN False  # Cycle detected

    RETURN True  # All courses can be completed
```

>>>[6.4].Median and Math:-----------------------------------------------------------------------------------------------------

#1. Finding Median (Using Sorting & Heap)

Approach 1: Sorting (Simple, O(N log N))

```
FUNCTION find_median(arr):
    SORT arr

    n = LENGTH(arr)
    IF n is ODD:
        RETURN arr[n // 2]  # Middle element
    ELSE:
        RETURN (arr[n // 2] + arr[n // 2 - 1]) / 2  # Average of middle elements
```

Approach 2: Using Two Heaps (Efficient, O(log N) per insert)

```
FUNCTION median_heap():
    CREATE max_heap (for left half)  # Stores smaller numbers
    CREATE min_heap (for right half) # Stores larger numbers

    FUNCTION insert_num(num):
        IF max_heap is empty OR num <= TOP(max_heap):
            PUSH num into max_heap
        ELSE:
            PUSH num into min_heap

        # Balance heaps
        IF SIZE(max_heap) > SIZE(min_heap) + 1:
            MOVE TOP(max_heap) to min_heap
        ELSE IF SIZE(min_heap) > SIZE(max_heap):
            MOVE TOP(min_heap) to max_heap

    FUNCTION get_median():
        IF SIZE(max_heap) == SIZE(min_heap):
            RETURN (TOP(max_heap) + TOP(min_heap)) / 2
        ELSE:
            RETURN TOP(max_heap)
```

#2. Math Problems:

1. Sum of First N Numbers

```
FUNCTION sum_n(n):
    RETURN (n * (n + 1)) / 2  # Formula: Sum of first N numbers
```

2. Check if a Number is Prime

```
FUNCTION is_prime(n):
    IF n <= 1:
        RETURN False
    IF n <= 3:
        RETURN True
    IF n is EVEN OR divisible by 3:
        RETURN False

    FOR i FROM 5 TO sqrt(n) STEP 2:
```

```
        IF n is divisible by i:
            RETURN False
    RETURN True
```

3. Greatest Common Divisor (GCD) Using Euclidean Algorithm

```
FUNCTION gcd(a, b):
    WHILE b != 0:
        a, b = b, a % b  # Keep replacing a with b, and b with remainder
    RETURN a
```

4. Least Common Multiple (LCM)

```
FUNCTION lcm(a, b):
    RETURN (a * b) / gcd(a, b)
```

>>>[6.5].Merge
Arrays:---------------------------------------------------------------------------------------------------------

1. Merge Two Unsorted Arrays

```
FUNCTION merge_unsorted(arr1, arr2):
    CREATE result = arr1 + arr2  # Concatenate both arrays
    SORT result  # Sort the merged array
    RETURN result
```

2. Merge Two Sorted Arrays (Efficient)

```
FUNCTION merge_sorted(arr1, arr2):
    CREATE result = []
    SET i = 0, j = 0

    # Step 1: Merge elements in sorted order
    WHILE i < LENGTH(arr1) AND j < LENGTH(arr2):
        IF arr1[i] < arr2[j]:
            APPEND arr1[i] to result
            INCREMENT i
        ELSE:
            APPEND arr2[j] to result
            INCREMENT j

    # Step 2: Append remaining elements
    WHILE i < LENGTH(arr1):
        APPEND arr1[i] to result
        INCREMENT i

    WHILE j < LENGTH(arr2):
        APPEND arr2[j] to result
        INCREMENT j

    RETURN result
```

3. Merge Two Sorted Arrays In-Place (Without Extra Space)

```
FUNCTION merge_sorted_in_place(arr1, arr2, m, n):
    SET i = m - 1  # Last element in arr1 (non-zero part)
    SET j = n - 1  # Last element in arr2
    SET k = m + n - 1  # Last position in arr1 (full size)
```

```
    # Merge from the back
    WHILE i >= 0 AND j >= 0:
        IF arr1[i] > arr2[j]:
            arr1[k] = arr1[i]
            DECREMENT i
        ELSE:
            arr1[k] = arr2[j]
            DECREMENT j
        DECREMENT k

    # If arr2 has remaining elements, copy them
    WHILE j >= 0:
        arr1[k] = arr2[j]
        DECREMENT j
        DECREMENT k
```

4. Merge K Sorted Arrays (Heap Approach)

```
FUNCTION merge_k_sorted(arrays):
    CREATE min_heap
    CREATE result = []

    # Step 1: Push the first element of each array into the heap
    FOR i FROM 0 TO LENGTH(arrays):
        PUSH (arrays[i][0], i, 0) into min_heap  # (value, array_index,
element_index)

    # Step 2: Extract elements in sorted order
    WHILE min_heap is NOT empty:
        (val, arr_idx, elem_idx) = POP from min_heap
        APPEND val to result

        # If next element exists in the same array, add it to heap
        IF elem_idx + 1 < LENGTH(arrays[arr_idx]):
            PUSH (arrays[arr_idx][elem_idx + 1], arr_idx, elem_idx + 1) into
min_heap

    RETURN result
```

>>>[6.6].Sliding
Window:---------------------------------------------------------------------------------------------------

1. Fixed-Size Sliding Window:

```
FUNCTION max_sum_subarray(arr, k):
    SET window_sum = SUM of first k elements
    SET max_sum = window_sum

    FOR i FROM k TO LENGTH(arr) - 1:
        window_sum = window_sum - arr[i - k] + arr[i]  # Slide the window
        max_sum = MAX(max_sum, window_sum)

    RETURN max_sum
```

2. Variable-Size Sliding Window:

```
FUNCTION min_subarray_length(arr, target):
```

```
    SET left = 0, window_sum = 0, min_length = ∞

    FOR right FROM 0 TO LENGTH(arr) - 1:
        window_sum += arr[right]  # Expand window

        WHILE window_sum ≥ target:
            min_length = MIN(min_length, right - left + 1)  # Update min size
            window_sum -= arr[left]  # Shrink window
            INCREMENT left

    RETURN min_length IF min_length ≠ ∞ ELSE 0
```

3. Longest Substring with Unique Characters:

```
FUNCTION longest_unique_substring(s):
    CREATE set seen_chars
    SET left = 0, max_length = 0

    FOR right FROM 0 TO LENGTH(s) - 1:
        WHILE s[right] IS IN seen_chars:
            REMOVE s[left] FROM seen_chars
            INCREMENT left

        ADD s[right] TO seen_chars
        max_length = MAX(max_length, right - left + 1)

    RETURN max_length
```

4. Longest Substring with At Most K Distinct Characters:

```
FUNCTION longest_substring_k_distinct(s, k):
    CREATE hashmap char_count
    SET left = 0, max_length = 0

    FOR right FROM 0 TO LENGTH(s) - 1:
        INCREMENT char_count[s[right]]

        WHILE SIZE of char_count > k:
            DECREMENT char_count[s[left]]
            IF char_count[s[left]] == 0:
                REMOVE s[left] FROM char_count
            INCREMENT left  # Shrink window

        max_length = MAX(max_length, right - left + 1)

    RETURN max_length
```

-----------------------------------------------------------------------------------------------------------------------------------------------

7.GRAPH:

1).Union Find
2).DFS
  1.Island Problems
  2.Cycle Find
  3.Reach all nodes
3).BFS
4).Graph Coloring/Bipartition
5).Topological sort

6)Shortest Path(Dijkstra's/Bellman Ford)

>>>[7.1].Union
Find:------------------------------------------------------------------------------------------------------------

1. Basic Union-Find (Without Optimization):

```
FUNCTION find(parent, x):
    WHILE parent[x] ≠ x:
        x = parent[x]  # Move up the tree
    RETURN x  # Root of x

FUNCTION union(parent, x, y):
    rootX = find(parent, x)
    rootY = find(parent, y)

    IF rootX ≠ rootY:
        parent[rootY] = rootX  # Merge y into x
```

2. Optimized Union-Find (Path Compression & Union by Rank):

```
FUNCTION find(parent, x):
    IF parent[x] ≠ x:
        parent[x] = find(parent, parent[x])  # Path compression
    RETURN parent[x]

FUNCTION union(parent, rank, x, y):
    rootX = find(parent, x)
    rootY = find(parent, y)

    IF rootX ≠ rootY:
        IF rank[rootX] > rank[rootY]:
            parent[rootY] = rootX
        ELSE IF rank[rootX] < rank[rootY]:
            parent[rootX] = rootY
        ELSE:
            parent[rootY] = rootX
            rank[rootX] += 1
```

3. Detect Cycle in an Undirected Graph:

```
FUNCTION detect_cycle(edges, N):
    CREATE parent array of size N
    CREATE rank array of size N, initialized to 0

    FOR i FROM 0 TO N-1:
        parent[i] = i  # Each node is its own leader

    FOR each (u, v) in edges:
        rootU = find(parent, u)
        rootV = find(parent, v)

        IF rootU == rootV:
            RETURN True  # Cycle detected
        ELSE:
            union(parent, rank, u, v)

    RETURN False  # No cycle found
```

4. Count Connected Components in a Graph:

```
FUNCTION count_components(N, edges):
    CREATE parent array of size N
    SET count = N  # Initially, all nodes are separate

    FOR i FROM 0 TO N-1:
        parent[i] = i

    FOR each (u, v) in edges:
        rootU = find(parent, u)
        rootV = find(parent, v)

        IF rootU ≠ rootV:
            union(parent, rank, u, v)
            DECREMENT count

    RETURN count
```

>>>[7.2].DFS:--------------------------------------------------------------------------------------------------

   1.Island Problems
   2.Cycle Find
   3.Reach all nodes

1. Island Problems (Number of Islands):

case1:

```
FUNCTION num_islands(grid):
    SET count = 0
    CREATE directions = [(0,1), (1,0), (0,-1), (-1,0)]  # Right, Down, Left, Up

    FUNCTION dfs(grid, i, j):
        IF i < 0 OR j < 0 OR i >= ROWS OR j >= COLS OR grid[i][j] == '0':
            RETURN
        grid[i][j] = '0'  # Mark as visited
        FOR each (dx, dy) in directions:
            dfs(grid, i + dx, j + dy)

    FOR i FROM 0 TO ROWS-1:
        FOR j FROM 0 TO COLS-1:
            IF grid[i][j] == '1':
                INCREMENT count
                dfs(grid, i, j)  # Explore full island

    RETURN count
```

case2:

```
function countIslands(grid):
    if grid is empty:
        return 0

    numIslands = 0
    rows = length of grid
    cols = length of grid[0]

    function dfs(r, c):
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == '0':
            return
```

```
        grid[r][c] = '0'  // Mark as visited
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    for r from 0 to rows - 1:
        for c from 0 to cols - 1:
            if grid[r][c] == '1':
                numIslands += 1
                dfs(r, c)

    return numIslands
```

2. Cycle Detection in Graph (Directed & Undirected):

case1:
Approach for Directed Graph:

```
FUNCTION has_cycle_directed(graph, N):
    CREATE visited = [0] * N  # 0 = Not visited

    FUNCTION dfs(node):
        IF visited[node] == 1:
            RETURN True  # Cycle detected

        IF visited[node] == 2:
            RETURN False  # Already processed, no cycle

        visited[node] = 1  # Mark node as being visited
        FOR neighbor in graph[node]:
            IF dfs(neighbor):
                RETURN True

        visited[node] = 2  # Fully explored
        RETURN False

    FOR i FROM 0 TO N-1:
        IF visited[i] == 0 AND dfs(i):
            RETURN True

    RETURN False  # No cycle found
```

Approach for Undirected Graph:

```
FUNCTION has_cycle_undirected(graph, N):
    CREATE visited = [False] * N

    FUNCTION dfs(node, parent):
        visited[node] = True

        FOR neighbor in graph[node]:
            IF NOT visited[neighbor]:
                IF dfs(neighbor, node):
                    RETURN True
            ELSE IF neighbor ≠ parent:
                RETURN True  # Found a back edge (cycle)

        RETURN False
```

```
    FOR i FROM 0 TO N-1:
        IF NOT visited[i]:
            IF dfs(i, -1):
                RETURN True

    RETURN False  # No cycle found
```

case2:

```
function hasCycle(graph):
    visited = set()

    function dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node):
                    return True
            elif neighbor != parent:
                return True
        return False

    for node in graph:
        if node not in visited:
            if dfs(node, None):
                return True

    return False
```

3. Reach All Nodes (Check Graph Connectivity):

case1:

```
FUNCTION can_reach_all_nodes(graph, N):
    CREATE visited = [False] * N

    FUNCTION dfs(node):
        visited[node] = True
        FOR neighbor in graph[node]:
            IF NOT visited[neighbor]:
                dfs(neighbor)

    dfs(0)  # Start from node 0

    RETURN ALL(visited)  # True if all nodes are visited
```

case2:

```
function canReachAllNodes(graph, startNode):
    visited = set()

    function dfs(node):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor)

    dfs(startNode)

    return len(visited) == len(graph)
```

>>>[7.3].BFS:---------------------------------------------------------------------------------------------------------

1. Standard BFS Traversal:

```
FUNCTION bfs(graph, start, N):
    CREATE visited[N] = [False]  # Track visited nodes
    CREATE queue = [start]  # Initialize queue
    visited[start] = True

    WHILE queue IS NOT EMPTY:
        node = queue.pop(0)  # Dequeue first node
        PRINT(node)  # Process node

        FOR neighbor IN graph[node]:
            IF NOT visited[neighbor]:
                queue.append(neighbor)  # Enqueue unvisited neighbors
                visited[neighbor] = True
```

2. Shortest Path in an Unweighted Graph:

```
FUNCTION shortest_path(graph, start, end, N):
    CREATE visited[N] = [False]
    CREATE queue = [(start, 0)]  # (node, distance)
    visited[start] = True

    WHILE queue IS NOT EMPTY:
        node, dist = queue.pop(0)

        IF node == end:
            RETURN dist  # Return shortest distance

        FOR neighbor IN graph[node]:
            IF NOT visited[neighbor]:
                queue.append((neighbor, dist + 1))  # Increase distance
                visited[neighbor] = True

    RETURN -1  # No path found
```

3. Cycle Detection in an Undirected Graph:

```
FUNCTION has_cycle_undirected(graph, N):
    CREATE visited[N] = [False]

    FOR i FROM 0 TO N-1:
        IF NOT visited[i]:
            CREATE queue = [(i, -1)]  # (node, parent)
            visited[i] = True

            WHILE queue IS NOT EMPTY:
                node, parent = queue.pop(0)

                FOR neighbor IN graph[node]:
                    IF NOT visited[neighbor]:
                        queue.append((neighbor, node))
                        visited[neighbor] = True
                    ELSE IF neighbor ≠ parent:
                        RETURN True  # Found a cycle
```

```
    RETURN False  # No cycle found
```

4. Check Graph Connectivity:

```
FUNCTION is_connected(graph, N):
    CREATE visited[N] = [False]
    CREATE queue = [0]  # Start BFS from node 0
    visited[0] = True

    WHILE queue IS NOT EMPTY:
        node = queue.pop(0)

        FOR neighbor IN graph[node]:
            IF NOT visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True

    RETURN ALL(visited)  # True if all nodes were visited
```

5. Number of Connected Components:

```
FUNCTION count_components(graph, N):
    CREATE visited[N] = [False]
    SET count = 0

    FOR i FROM 0 TO N-1:
        IF NOT visited[i]:
            bfs(graph, i, visited)  # Run BFS from this component
            count += 1  # Increment component count

    RETURN count

FUNCTION bfs(graph, start, visited):
    CREATE queue = [start]
    visited[start] = True

    WHILE queue IS NOT EMPTY:
        node = queue.pop(0)

        FOR neighbor IN graph[node]:
            IF NOT visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True
```

>>>[7.4].Graph
Coloring/Bipartition:-----------------------------------------------------------------------------------------------------

1. Graph Bipartition (Check if Graph is Bipartite):

```
FUNCTION is_bipartite(graph, N):
    CREATE color[N] = [-1] * N  # -1 means uncolored

    FUNCTION bfs(start):
        CREATE queue = [start]
        color[start] = 0  # Assign first color

        WHILE queue IS NOT EMPTY:
```

```
            node = queue.pop(0)

            FOR neighbor IN graph[node]:
                IF color[neighbor] == -1:  # Not colored yet
                    color[neighbor] = 1 - color[node]  # Assign opposite color
                    queue.append(neighbor)
                ELSE IF color[neighbor] == color[node]:
                    RETURN False  # Conflict detected, not bipartite

        RETURN True

    FOR i FROM 0 TO N-1:
        IF color[i] == -1:
            IF NOT bfs(i):  # Check all components
                RETURN False

    RETURN True  # Graph is bipartite
```

2. Graph Coloring (k-Colorability):

```
FUNCTION can_color_graph(graph, N, k):
    CREATE colors[N] = [-1] * N  # -1 means uncolored

    FUNCTION is_safe(node, c):
        FOR neighbor IN graph[node]:
            IF colors[neighbor] == c:
                RETURN False  # Conflict detected
        RETURN True

    FUNCTION backtrack(node):
        IF node == N:
            RETURN True  # All nodes colored

        FOR c FROM 0 TO k-1:  # Try all k colors
            IF is_safe(node, c):
                colors[node] = c  # Assign color
                IF backtrack(node + 1):
                    RETURN True  # Solution found
                colors[node] = -1  # Undo assignment (backtrack)

        RETURN False  # No valid color found

    RETURN backtrack(0)  # Start coloring from node 0
```

3. Find Chromatic Number (Minimum Colors Required):

```
FUNCTION find_chromatic_number(graph, N):
    FOR k FROM 1 TO N:
        IF can_color_graph(graph, N, k):
            RETURN k  # Smallest valid number of colors
```

>>>[7.5].Topological sort:-------------------------------------------------------------------------------------------------

```
TopologicalSort(Graph G):
    1. Initialize an empty stack S
    2. Initialize an empty set of visited nodes V
```

```
    3. For each node u in G:
        a. If u is not in V:
            i. Call TopologicalSortUtil(u, V, S)

    4. While S is not empty:
        a. Pop a node from S and print it

TopologicalSortUtil(Node u, Set V, Stack S):
    1. Mark u as visited by adding it to V

    2. For each neighbor v of u:
        a. If v is not in V:
            i. Call TopologicalSortUtil(v, V, S)

    3. Push u onto S
```

#Pseudocode for Topological Sorting in Graphs:
There are two main approaches to performing a topological sort:
Kahn's Algorithm (BFS-based, using in-degree)
DFS-based (Using a Stack)

1. Kahn’s Algorithm (BFS-based):

```
FUNCTION topological_sort_kahn(graph, N):
    CREATE in_degree[N] = [0] * N  # Track incoming edges
    FOR each node IN graph:
        FOR each neighbor IN graph[node]:
            in_degree[neighbor] += 1  # Count incoming edges

    CREATE queue = []  # Store nodes with in-degree 0
    FOR i FROM 0 TO N-1:
        IF in_degree[i] == 0:
            queue.append(i)

    CREATE result = []  # Store topological order
    WHILE queue IS NOT EMPTY:
        node = queue.pop(0)  # Process first node in queue
        result.append(node)

        FOR neighbor IN graph[node]:
            in_degree[neighbor] -= 1  # Remove edge
            IF in_degree[neighbor] == 0:
                queue.append(neighbor)  # Add new zero in-degree nodes

    IF LENGTH(result) == N:
        RETURN result  # Valid topological order
    ELSE:
        RETURN "Cycle detected"  # No topological order (Graph has a cycle)
```

2. DFS-Based Approach:

```
FUNCTION topological_sort_dfs(graph, N):
    CREATE visited[N] = [False]  # Track visited nodes
    CREATE stack = []  # Store topological order

    FUNCTION dfs(node):
        visited[node] = True

        FOR neighbor IN graph[node]:
```

```
            IF NOT visited[neighbor]:
                dfs(neighbor)

        stack.append(node)  # Push to stack after visiting all neighbors

    FOR i FROM 0 TO N-1:
        IF NOT visited[i]:
            dfs(i)

    RETURN REVERSE(stack)  # Stack gives order in reverse
```

>>>[7.6].Shortest Path(Dijkstra's/Bellman Ford):-------------------------------------------------------------------------------------------------------

1. Dijkstra’s Algorithm (Greedy Approach):

case1:

```
function Dijkstra(graph, source):
    dist[source] <- 0
    for each vertex v in graph:
        if v != source:
            dist[v] <- infinity
        add v to the priority queue Q

    while Q is not empty:
        u <- vertex in Q with the smallest dist[u]
        remove u from Q

        for each neighbor v of u:
            alt <- dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] <- alt
                previous[v] <- u
                decrease priority of v in Q

    return dist, previous
```

case2:

```
FUNCTION dijkstra(graph, N, start):
    CREATE dist[N] = [∞]  # Distance array initialized to infinity
    CREATE min_heap = [(0, start)]  # (distance, node)
    dist[start] = 0

    WHILE min_heap IS NOT EMPTY:
        (d, node) = min_heap.pop()  # Extract the node with the smallest distance

        IF d > dist[node]:  # Skip if already processed
            CONTINUE

        FOR (neighbor, weight) IN graph[node]:  # Iterate over neighbors
            new_dist = dist[node] + weight
            IF new_dist < dist[neighbor]:  # Found a shorter path
                dist[neighbor] = new_dist
                min_heap.push((new_dist, neighbor))  # Push updated distance

    RETURN dist  # Shortest distances from start node
```

2. Bellman-Ford Algorithm (Dynamic Programming):

```
case1:
function BellmanFord(graph, source):
    dist[source] <- 0
    for each vertex v in graph:
        if v != source:
            dist[v] <- infinity

    for i from 1 to |V| - 1:
        for each edge (u, v) in graph:
            if dist[u] + length(u, v) < dist[v]:
                dist[v] <- dist[u] + length(u, v)
                previous[v] <- u

    for each edge (u, v) in graph:
        if dist[u] + length(u, v) < dist[v]:
            error "Graph contains a negative-weight cycle"

    return dist, previous

case2:
FUNCTION bellman_ford(graph, N, start):
    CREATE dist[N] = [∞]  # Distance array initialized to infinity
    dist[start] = 0

    FOR i FROM 1 TO N-1:  # Relax all edges (N-1) times
        FOR (u, v, weight) IN edges:  # Iterate over all edges
            IF dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Check for negative-weight cycles
    FOR (u, v, weight) IN edges:
        IF dist[u] + weight < dist[v]:
            RETURN "Negative cycle detected"

    RETURN dist  # Shortest distances from start node

3. Detect Negative Cycles using Bellman-Ford:

FUNCTION detect_negative_cycle(graph, N):
    CREATE dist[N] = [∞]
    dist[0] = 0  # Start from any node

    FOR i FROM 1 TO N-1:
        FOR (u, v, weight) IN edges:
            IF dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Run one more iteration to check for cycles
    FOR (u, v, weight) IN edges:
        IF dist[u] + weight < dist[v]:
            RETURN True  # Negative cycle exists

    RETURN False
```

------------------------------------------------------------------------------------------------------------------------------------------------------

8.STRING:

1).Palindromic String
2).Types of String
3).Parenthesis Problem
4).Count substring
5).Sorting on String
6).Longest and Shortest
7).Sliding window substring
8).Permutation Problems
9).Pattern Print
10)Lexicographic Problems

>>>[8.1].Palindromic String:----------------------------------------------------------------------------------------------------

```
FUNCTION isPalindrome(s):
    LEFT ← 0
    RIGHT ← length(s) - 1

    WHILE LEFT < RIGHT:
        IF s[LEFT] ≠ s[RIGHT]:
            RETURN False
        LEFT ← LEFT + 1
        RIGHT ← RIGHT - 1

    RETURN True
```

>>>[8.2].Types of String:----------------------------------------------------------------------------------------------------

1. Check if a String is Palindromic:

```
FUNCTION isPalindrome(s):
    LEFT ← 0
    RIGHT ← length(s) - 1

    WHILE LEFT < RIGHT:
        IF s[LEFT] ≠ s[RIGHT]:
            RETURN False
        LEFT ← LEFT + 1
        RIGHT ← RIGHT - 1

    RETURN True
```

2. Check if a String is a Pangram:

```
FUNCTION isPangram(s):
    ALPHABET_SET ← set of all letters 'a' to 'z'
    s ← convert s to lowercase

    FOR CHAR in s:
        IF CHAR is a letter:
            REMOVE CHAR from ALPHABET_SET

    IF ALPHABET_SET is empty:
        RETURN True
    ELSE:
```

```
        RETURN False
```

3. Check if a String is an Anagram:

```
FUNCTION isAnagram(s1, s2):
    IF length(s1) ≠ length(s2):
        RETURN False

    SORT s1 and s2 alphabetically

    IF sorted(s1) = sorted(s2):
        RETURN True
    ELSE:
        RETURN False
```

4. Check if a String is Numeric:

```
FUNCTION isNumeric(s):
    FOR CHAR in s:
        IF CHAR is NOT a digit:
            RETURN False
    RETURN True
```

5. Check if a String is Alphabetic:

```
FUNCTION isAlphabetic(s):
    FOR CHAR in s:
        IF CHAR is NOT a letter:
            RETURN False
    RETURN True
```

6. Check if a String is Alphanumeric:

```
FUNCTION isAlphanumeric(s):
    FOR CHAR in s:
        IF CHAR is NOT a letter AND NOT a digit:
            RETURN False
    RETURN True
```

7. Convert a String to Title Case:

```
FUNCTION toTitleCase(s):
    WORDS ← split s into words
    FOR EACH WORD in WORDS:
        Capitalize the first letter of WORD
    RETURN join WORDS into a single string
```

>>>[8.3].Parenthesis
Problem:----------------------------------------------------------------------------------------------------

```
FUNCTION isValidParentheses(s):
    STACK ← empty list
    PAREN_MAP ← { ')' → '(', ']' → '[', '}' → '{' }

    FOR CHAR in s:
        IF CHAR is an opening bracket ('(', '[', '{'):
            PUSH CHAR onto STACK
```

```
        ELSE IF CHAR is a closing bracket (')', ']', '}'):
            IF STACK is empty OR STACK.TOP ≠ PAREN_MAP[CHAR]:
                RETURN False
            POP from STACK

    RETURN STACK is empty  // True if balanced, False otherwise
```

>>>[8.4].Count
substring:-------------------------------------------------------------------------------------------------------------

```
FUNCTION countSubstring(mainString, subString):
    COUNT ← 0
    LEN_MAIN ← length(mainString)
    LEN_SUB ← length(subString)

    FOR i FROM 0 TO (LEN_MAIN - LEN_SUB):
        IF mainString[i : i + LEN_SUB] == subString:
            COUNT ← COUNT + 1

    RETURN COUNT
```

>>>[8.5].Sorting on
String:-------------------------------------------------------------------------------------------------------------

Pseudocode (Sorting a String in Lexicographical Order):

```
FUNCTION sortString(s):
    CONVERT s to an array of characters
    SORT the array in ascending order
    RETURN the sorted array joined back into a string
```

Detailed Pseudocode (Step-by-Step):

```
FUNCTION sortString(s):
    CHAR_ARRAY ← convert s into an array of characters
    N ← length(CHAR_ARRAY)

    FOR i FROM 0 TO N-1:
        FOR j FROM 0 TO N-i-1:
            IF CHAR_ARRAY[j] > CHAR_ARRAY[j+1]:  // Compare adjacent characters
                SWAP CHAR_ARRAY[j] and CHAR_ARRAY[j+1]  // Swap if out of order

    RETURN join CHAR_ARRAY into a string
```

>>>[8.6].Longest and
Shortest:-------------------------------------------------------------------------------------------------------------

```
FUNCTION findLongestAndShortest(s):
    WORDS ← split s by spaces
    LONGEST ← WORDS[0]
    SHORTEST ← WORDS[0]

    FOR EACH WORD in WORDS:
        IF length(WORD) > length(LONGEST):
```

```
            LONGEST ← WORD
        IF length(WORD) < length(SHORTEST):
            SHORTEST ← WORD

    RETURN LONGEST, SHORTEST
```

>>>[8.7].Sliding window substring:------------------------------------------------------------------------------------------------------------

```
FUNCTION longestUniqueSubstring(s):
    LEFT ← 0
    MAX_LENGTH ← 0
    CHAR_MAP ← empty dictionary (to track character positions)

    FOR RIGHT FROM 0 TO length(s) - 1:
        IF s[RIGHT] is in CHAR_MAP AND CHAR_MAP[s[RIGHT]] ≥ LEFT:
            LEFT ← CHAR_MAP[s[RIGHT]] + 1  // Move LEFT to avoid repetition

        CHAR_MAP[s[RIGHT]] ← RIGHT  // Update last seen position
        MAX_LENGTH ← max(MAX_LENGTH, RIGHT - LEFT + 1)

    RETURN MAX_LENGTH
```

>>>[8.8].Permutation Problems:------------------------------------------------------------------------------------------------------------

1. Generate All Permutations of a String:

```
FUNCTION generatePermutations(s, LEFT, RIGHT):
    IF LEFT == RIGHT:
        PRINT s  // Base case: a permutation is found
    ELSE:
        FOR i FROM LEFT TO RIGHT:
            SWAP s[LEFT] and s[i]   // Swap to create a new permutation
            generatePermutations(s, LEFT + 1, RIGHT)
            SWAP s[LEFT] and s[i]   // Backtrack to restore original state
```

2. Check If Two Strings Are Permutations of Each Other:

```
FUNCTION arePermutations(s1, s2):
    IF length(s1) ≠ length(s2):
        RETURN False

    SORT s1
    SORT s2

    RETURN s1 == s2  // Check if sorted versions match
```

3. Find the Next Lexicographical Permutation:

```
FUNCTION nextPermutation(arr):
    FIND largest index i where arr[i] < arr[i+1]
    IF no such i exists:
        REVERSE arr and RETURN

    FIND largest index j where arr[j] > arr[i]
```

```
    SWAP arr[i] and arr[j]
    REVERSE arr[i+1:]  // Reverse suffix

    RETURN arr
```

>>>[8.9].Pattern
Print:------------------------------------------------------------------------------------------------------------

1. Triangle Pattern (Right-Aligned):

```
FUNCTION printTrianglePattern(s):
    N ← length(s)

    FOR i FROM 1 TO N:
        PRINT s[0:i]  // Print the first i characters
```

2. Reverse Triangle Pattern:

```
FUNCTION printReverseTrianglePattern(s):
    N ← length(s)

    FOR i FROM N DOWN TO 1:
        PRINT s[0:i]  // Print the first i characters
```

3. Diamond Pattern:

```
FUNCTION printDiamondPattern(s):
    N ← length(s)

    // Upper Part
    FOR i FROM 1 TO N:
        PRINT (N-i) spaces + s[0:i]

    // Lower Part
    FOR i FROM N-1 DOWN TO 1:
        PRINT (N-i) spaces + s[0:i]
```

4. Zig-Zag Pattern:

```
FUNCTION printZigZagPattern(s, rows):
    IF rows == 1:
        PRINT s
        RETURN

    CREATE 2D array zigzag[rows][length(s)]
    DIRECTION ← down  // Flag to track movement
    ROW ← 0

    FOR EACH CHARACTER c IN s:
        zigzag[ROW] ← zigzag[ROW] + c
        IF ROW == 0:
            DIRECTION ← down
        ELSE IF ROW == rows-1:
            DIRECTION ← up

        IF DIRECTION == down:
            ROW ← ROW + 1
        ELSE:
```

```
        ROW ← ROW - 1

    FOR EACH ROW IN zigzag:
        PRINT ROW
```

>>>[8.10].Lexicographic
Problems:--------------------------------------------------------------------------------------------------------------

1. Find the Lexicographically Smallest and Largest Substring:

```
FUNCTION findLexicographicExtremes(s, k):
    SUBSTRINGS ← empty list

    FOR i FROM 0 TO length(s) - k:
        ADD substring(s, i, i+k) TO SUBSTRINGS  // Extract substring of length k

    SORT SUBSTRINGS lexicographically

    SMALLEST ← SUBSTRINGS[0]
    LARGEST ← SUBSTRINGS[length(SUBSTRINGS) - 1]

    RETURN SMALLEST, LARGEST
```

2. Generate All Lexicographic Permutations of a String:

```
FUNCTION lexicographicPermutations(s):
    SORT s  // Ensure we start from the smallest permutation
    WHILE s is not NULL:
        PRINT s
        s ← next lexicographic permutation(s)
```

3. Check If Two Strings Are Lexicographically Ordered:

```
FUNCTION isLexicographicallySmaller(s1, s2):
    RETURN s1 < s2
```

4. Find the Lexicographically Smallest Rotation of a String:

```
FUNCTION lexicographicallySmallestRotation(s):
    N ← length(s)
    ROTATIONS ← empty list

    FOR i FROM 0 TO N-1:
        ROTATIONS.ADD( s[i:] + s[:i] )  // Rotate string

    RETURN MIN(ROTATIONS)
```

5. Find the Smallest Lexicographic Concatenation of Strings:

```
FUNCTION smallestLexicographicConcatenation(strings):
    SORT strings USING custom comparator (x + y < y + x)
    RETURN concatenate(strings)
```

-----------------------------------------------------------------------------------------------------------------------------

9.TREE:

1).Kth Smallest/Largest
2).Ancestor Problems
3).Range Sum
4).Traversal
5).Distance Between Nodes
6).Tree Construction
7).Serialize and Deserialize
8).Searching
9).Root to Leaf Path
10).Depth Problems
11).Check/Compare Binary Trees

>>>[9.1].Kth Smallest/Largest:--------------------------------------------------------------------------------------------------------

1. Find Kth Smallest Element in BST:

```
FUNCTION kthSmallest(root, k):
    COUNT ← 0
    RESULT ← NULL

    FUNCTION inorder(node):
        IF node is NULL OR RESULT is NOT NULL:
            RETURN

        inorder(node.left)  // Traverse left subtree

        COUNT ← COUNT + 1
        IF COUNT == k:
            RESULT ← node.value
            RETURN

        inorder(node.right)  // Traverse right subtree

    inorder(root)
    RETURN RESULT
```

2. Find Kth Largest Element in BST:

```
FUNCTION kthLargest(root, k):
    COUNT ← 0
    RESULT ← NULL

    FUNCTION reverseInorder(node):
        IF node is NULL OR RESULT is NOT NULL:
            RETURN

        reverseInorder(node.right)  // Traverse right subtree

        COUNT ← COUNT + 1
        IF COUNT == k:
            RESULT ← node.value
            RETURN

        reverseInorder(node.left)  // Traverse left subtree

    reverseInorder(root)
```

```
    RETURN RESULT
```

3. Find Kth Smallest/Largest Using Iterative Approach:

```
FUNCTION kthSmallestIterative(root, k):
    STACK ← empty stack
    CURRENT ← root
    COUNT ← 0

    WHILE STACK is NOT empty OR CURRENT is NOT NULL:
        WHILE CURRENT is NOT NULL:
            PUSH CURRENT to STACK
            CURRENT ← CURRENT.left  // Move left

        CURRENT ← POP STACK
        COUNT ← COUNT + 1

        IF COUNT == k:
            RETURN CURRENT.value

        CURRENT ← CURRENT.right  // Move right

    RETURN NULL  // If k is out of range
```

>>>[9.2].Ancestor
Problems:---------------------------------------------------------------------------------------------------------

1. Find All Ancestors of a Given Node in a Binary Tree:

```
FUNCTION findAncestors(root, target):
    IF root is NULL:
        RETURN False

    IF root.value == target:
        RETURN True

    IF findAncestors(root.left, target) OR findAncestors(root.right, target):
        PRINT root.value  // Print ancestor
        RETURN True

    RETURN False
```

2. Find the Lowest Common Ancestor (LCA) of Two Nodes:

```
FUNCTION lowestCommonAncestor(root, p, q):
    IF root is NULL OR root.value == p OR root.value == q:
        RETURN root

    LEFT ← lowestCommonAncestor(root.left, p, q)
    RIGHT ← lowestCommonAncestor(root.right, p, q)

    IF LEFT is NOT NULL AND RIGHT is NOT NULL:
        RETURN root  // Both nodes found, this is LCA

    RETURN LEFT IF LEFT is NOT NULL ELSE RIGHT
```

3. Find the Kth Ancestor of a Node:

```
FUNCTION kthAncestor(root, target, k):
    MAP parent  // Store parent pointers

    FUNCTION storeParents(node, parentNode):
        IF node is NULL:
            RETURN

        parent[node.value] ← parentNode
        storeParents(node.left, node)
        storeParents(node.right, node)

    storeParents(root, NULL)

    CURRENT ← target
    FOR i FROM 1 TO k:
        IF CURRENT is NULL:
            RETURN -1  // Kth ancestor doesn't exist
        CURRENT ← parent[CURRENT]

    RETURN CURRENT
```

>>>[9.3].Range Sum:-----------------------------------------------------------------------------------------------------

1. Range Sum in a Binary Search Tree (BST):

```
FUNCTION rangeSumBST(root, low, high):
    IF root is NULL:
        RETURN 0

    SUM ← 0

    IF low ≤ root.value ≤ high:
        SUM ← SUM + root.value  // Include this node

    IF root.value > low:
        SUM ← SUM + rangeSumBST(root.left, low, high)  // Search left subtree

    IF root.value < high:
        SUM ← SUM + rangeSumBST(root.right, low, high)  // Search right subtree

    RETURN SUM
```

Pseudocode (Iterative Approach - Using Stack):

```
FUNCTION rangeSumBST_iterative(root, low, high):
    SUM ← 0
    STACK ← [root]

    WHILE STACK is NOT empty:
        NODE ← POP(STACK)

        IF NODE is NOT NULL:
            IF low ≤ NODE.value ≤ high:
                SUM ← SUM + NODE.value  // Include this node
```

```
            IF NODE.value > low:
                PUSH NODE.left TO STACK  // Search left subtree

            IF NODE.value < high:
                PUSH NODE.right TO STACK  // Search right subtree

    RETURN SUM
```

2. Range Sum in a Binary Tree (Brute Force)

```
FUNCTION rangeSumBinaryTree(root, low, high):
    IF root is NULL:
        RETURN 0

    SUM ← 0
    IF low ≤ root.value ≤ high:
        SUM ← SUM + root.value

    SUM ← SUM + rangeSumBinaryTree(root.left, low, high)
    SUM ← SUM + rangeSumBinaryTree(root.right, low, high)

    RETURN SUM
```

>>>[9.4].Traversal:----------------------------------------------------------------------------------------------------------

###1. Depth-First Traversals (DFS):

1.1 Inorder Traversal (Left → Root → Right):
#Pseudocode (Recursive)::

```
FUNCTION inorderTraversal(root):
    IF root is NULL:
        RETURN
    inorderTraversal(root.left)   // Visit left subtree
    PRINT root.value              // Visit root
    inorderTraversal(root.right)  // Visit right subtree
```

#Pseudocode (Iterative - Using Stack)::

```
FUNCTION inorderTraversalIterative(root):
    STACK ← empty
    CURRENT ← root

    WHILE STACK is NOT empty OR CURRENT is NOT NULL:
        WHILE CURRENT is NOT NULL:
            PUSH CURRENT TO STACK
            CURRENT ← CURRENT.left   // Move left

        CURRENT ← POP STACK
        PRINT CURRENT.value   // Visit root
        CURRENT ← CURRENT.right   // Move right
```

1.2 Preorder Traversal (Root → Left → Right):
#Pseudocode (Recursive):

```
FUNCTION preorderTraversal(root):
    IF root is NULL:
```

```
        RETURN
    PRINT root.value              // Visit root
    preorderTraversal(root.left)  // Visit left subtree
    preorderTraversal(root.right) // Visit right subtree
```

#Pseudocode (Iterative - Using Stack):

```
FUNCTION preorderTraversalIterative(root):
    STACK ← empty
    PUSH root TO STACK

    WHILE STACK is NOT empty:
        NODE ← POP STACK
        PRINT NODE.value  // Visit root

        IF NODE.right is NOT NULL:
            PUSH NODE.right TO STACK  // Right child pushed first

        IF NODE.left is NOT NULL:
            PUSH NODE.left TO STACK  // Left child pushed second (processed first)
```

1.3 Postorder Traversal (Left → Right → Root):
#Pseudocode (Recursive):

```
FUNCTION postorderTraversal(root):
    IF root is NULL:
        RETURN
    postorderTraversal(root.left)   // Visit left subtree
    postorderTraversal(root.right)  // Visit right subtree
    PRINT root.value                // Visit root
```

#Pseudocode (Iterative - Using Two Stacks):

```
FUNCTION postorderTraversalIterative(root):
    IF root is NULL:
        RETURN

    STACK1 ← empty
    STACK2 ← empty
    PUSH root TO STACK1

    WHILE STACK1 is NOT empty:
        NODE ← POP STACK1
        PUSH NODE TO STACK2

        IF NODE.left is NOT NULL:
            PUSH NODE.left TO STACK1
        IF NODE.right is NOT NULL:
            PUSH NODE.right TO STACK1

    WHILE STACK2 is NOT empty:
        PRINT POP STACK2  // Reverse order of ROOT-LEFT-RIGHT (Preorder)
```

2. Breadth-First Traversal (BFS) / Level Order Traversal:

```
FUNCTION levelOrderTraversal(root):
    IF root is NULL:
        RETURN
```

```
    QUEUE ← empty
    ENQUEUE root TO QUEUE

    WHILE QUEUE is NOT empty:
        NODE ← DEQUEUE QUEUE
        PRINT NODE.value  // Visit node

        IF NODE.left is NOT NULL:
            ENQUEUE NODE.left TO QUEUE
        IF NODE.right is NOT NULL:
            ENQUEUE NODE.right TO QUEUE
```

>>>[9.5].Distance Between Nodes:-----------------------------------------------------------------------------------------------------

```
FUNCTION findDistance(root, node1, node2):
    FUNCTION findLCA(root, p, q):
        IF root is NULL OR root.value == p OR root.value == q:
            RETURN root

        LEFT ← findLCA(root.left, p, q)
        RIGHT ← findLCA(root.right, p, q)

        IF LEFT is NOT NULL AND RIGHT is NOT NULL:
            RETURN root  // This is the LCA

        RETURN LEFT IF LEFT is NOT NULL ELSE RIGHT

    FUNCTION findDepth(root, target, depth):
        IF root is NULL:
            RETURN -1  // Not found

        IF root.value == target:
            RETURN depth  // Found target node

        LEFT ← findDepth(root.left, target, depth + 1)
        IF LEFT ≠ -1:
            RETURN LEFT  // If found in left subtree, return depth

        RETURN findDepth(root.right, target, depth + 1)  // Check right subtree

    LCA ← findLCA(root, node1, node2)

    DIST1 ← findDepth(root, node1, 0)
    DIST2 ← findDepth(root, node2, 0)
    LCA_DEPTH ← findDepth(root, LCA.value, 0)

    RETURN DIST1 + DIST2 - 2 × LCA_DEPTH
```

>>>[9.6].Tree Construction:------------------------------------------------------------------------------------------------------

1. Construct a Binary Tree from Level Order Traversal:

```
FUNCTION buildTreeFromLevelOrder(arr):
```

```
    IF arr is empty:
        RETURN NULL

    ROOT ← CREATE new TreeNode(arr[0])
    QUEUE ← empty
    ENQUEUE ROOT TO QUEUE
    INDEX ← 1

    WHILE INDEX < LENGTH(arr):
        NODE ← DEQUEUE QUEUE

        IF arr[INDEX] is NOT NULL:
            NODE.left ← CREATE new TreeNode(arr[INDEX])
            ENQUEUE NODE.left TO QUEUE

        INDEX ← INDEX + 1

        IF INDEX < LENGTH(arr) AND arr[INDEX] is NOT NULL:
            NODE.right ← CREATE new TreeNode(arr[INDEX])
            ENQUEUE NODE.right TO QUEUE

        INDEX ← INDEX + 1

    RETURN ROOT
```

2. Construct a Binary Tree from Inorder and Preorder Traversal:

```
FUNCTION buildTreeFromPreIn(preorder, inorder):
    IF preorder is empty OR inorder is empty:
        RETURN NULL

    ROOT_VAL ← preorder[0]
    ROOT ← CREATE new TreeNode(ROOT_VAL)

    ROOT_INDEX_IN_INORDER ← FIND INDEX of ROOT_VAL in inorder

    LEFT_INORDER ← inorder[0 : ROOT_INDEX_IN_INORDER]
    RIGHT_INORDER ← inorder[ROOT_INDEX_IN_INORDER + 1 : END]

    LEFT_PREORDER ← preorder[1 : LENGTH(LEFT_INORDER) + 1]
    RIGHT_PREORDER ← preorder[LENGTH(LEFT_INORDER) + 1 : END]

    ROOT.left ← buildTreeFromPreIn(LEFT_PREORDER, LEFT_INORDER)
    ROOT.right ← buildTreeFromPreIn(RIGHT_PREORDER, RIGHT_INORDER)

    RETURN ROOT
```

3. Construct a Binary Search Tree (BST) from Sorted Array:

```
FUNCTION buildBSTFromSortedArray(arr, start, end):
    IF start > end:
        RETURN NULL

    MID ← (start + end) / 2
    ROOT ← CREATE new TreeNode(arr[MID])

    ROOT.left ← buildBSTFromSortedArray(arr, start, MID - 1)
    ROOT.right ← buildBSTFromSortedArray(arr, MID + 1, end)
```

```
    RETURN ROOT
```

>>>[9.7].Serialize and Deserialize:-------------------------------------------------------------------------------------------------------

1. Serialization (Convert Tree → String/List):

```
FUNCTION serialize(root):
    IF root is NULL:
        RETURN "NULL"

    QUEUE ← empty
    ENQUEUE root TO QUEUE
    RESULT ← empty list

    WHILE QUEUE is NOT empty:
        NODE ← DEQUEUE QUEUE

        IF NODE is NOT NULL:
            APPEND NODE.value TO RESULT
            ENQUEUE NODE.left TO QUEUE
            ENQUEUE NODE.right TO QUEUE
        ELSE:
            APPEND "NULL" TO RESULT  // Placeholder for missing nodes

    RETURN JOIN(RESULT, ",")  // Convert list to a comma-separated string
```

2. Deserialization (Convert String/List → Tree):

```
FUNCTION deserialize(data):
    IF data is "NULL":
        RETURN NULL

    NODES ← SPLIT(data, ",")
    ROOT ← CREATE new TreeNode(NODES[0])
    QUEUE ← empty
    ENQUEUE ROOT TO QUEUE
    INDEX ← 1

    WHILE QUEUE is NOT empty:
        NODE ← DEQUEUE QUEUE

        IF NODES[INDEX] ≠ "NULL":
            NODE.left ← CREATE new TreeNode(NODES[INDEX])
            ENQUEUE NODE.left TO QUEUE
        INDEX ← INDEX + 1

        IF INDEX < LENGTH(NODES) AND NODES[INDEX] ≠ "NULL":
            NODE.right ← CREATE new TreeNode(NODES[INDEX])
            ENQUEUE NODE.right TO QUEUE
        INDEX ← INDEX + 1

    RETURN ROOT
```

>>>[9.8].Searching:-------------------------------------------------------------------------------------------------

1. Searching in a Binary Tree (DFS):
#Approach (Recursive):

```
FUNCTION searchInBinaryTree(root, target):
    IF root is NULL:
        RETURN False  // Target not found

    IF root.value == target:
        RETURN True  // Target found

    RETURN searchInBinaryTree(root.left, target) OR searchInBinaryTree(root.right, target)
```

#Approach (Iterative - Using Stack):

```
FUNCTION searchInBinaryTreeIterative(root, target):
    IF root is NULL:
        RETURN False

    STACK ← empty
    PUSH root TO STACK

    WHILE STACK is NOT empty:
        NODE ← POP STACK

        IF NODE.value == target:
            RETURN True

        IF NODE.right is NOT NULL:
            PUSH NODE.right TO STACK

        IF NODE.left is NOT NULL:
            PUSH NODE.left TO STACK

    RETURN False  // Target not found
```

2. Searching in a Binary Tree (BFS - Level Order):

```
FUNCTION searchInBinaryTreeBFS(root, target):
    IF root is NULL:
        RETURN False

    QUEUE ← empty
    ENQUEUE root TO QUEUE

    WHILE QUEUE is NOT empty:
        NODE ← DEQUEUE QUEUE

        IF NODE.value == target:
            RETURN True

        IF NODE.left is NOT NULL:
            ENQUEUE NODE.left TO QUEUE

        IF NODE.right is NOT NULL:
            ENQUEUE NODE.right TO QUEUE

    RETURN False  // Target not found
```

3. Searching in a Binary Search Tree (BST):
#Approach (Recursive):

```
FUNCTION searchInBST(root, target):
    IF root is NULL:
        RETURN False

    IF root.value == target:
        RETURN True

    IF target < root.value:
        RETURN searchInBST(root.left, target)

    RETURN searchInBST(root.right, target)
```

#Approach (Iterative):

```
FUNCTION searchInBSTIterative(root, target):
    WHILE root is NOT NULL:
        IF root.value == target:
            RETURN True

        IF target < root.value:
            root ← root.left
        ELSE:
            root ← root.right

    RETURN False  // Target not found
```

>>>[9.9].Root to Leaf Path:-------------------------------------------------------------------------------------------------

1. Recursive Approach (DFS - Preorder Traversal):

Pseudocode:

```
FUNCTION findRootToLeafPaths(root, path, result):
    IF root is NULL:
        RETURN

    APPEND root.value TO path

    IF root.left is NULL AND root.right is NULL:  // Leaf node
        APPEND path TO result

    ELSE:
        findRootToLeafPaths(root.left, path, result)
        findRootToLeafPaths(root.right, path, result)

    REMOVE last element from path  // Backtrack
```

Driver Function:

```
FUNCTION getAllRootToLeafPaths(root):
    RESULT ← empty list
    PATH ← empty list
    findRootToLeafPaths(root, PATH, RESULT)
    RETURN RESULT
```

2. Iterative Approach (Using Stack - DFS):

```
FUNCTION findRootToLeafPathsIterative(root):
    IF root is NULL:
        RETURN []

    STACK ← [(root, [root.value])]
    RESULT ← empty list

    WHILE STACK is NOT empty:
        NODE, PATH ← POP STACK

        IF NODE.left is NULL AND NODE.right is NULL:
            APPEND PATH TO RESULT

        IF NODE.right is NOT NULL:
            PUSH (NODE.right, PATH + [NODE.right.value]) TO STACK

        IF NODE.left is NOT NULL:
            PUSH (NODE.left, PATH + [NODE.left.value]) TO STACK

    RETURN RESULT
```

>>>[9.10].Depth
Problems:-----------------------------------------------------------------------------------------------------

1. Maximum Depth (Height) of a Binary Tree:
#Recursive Approach (DFS):

```
FUNCTION maxDepth(root):
    IF root is NULL:
        RETURN 0

    LEFT_DEPTH ← maxDepth(root.left)
    RIGHT_DEPTH ← maxDepth(root.right)

    RETURN MAX(LEFT_DEPTH, RIGHT_DEPTH) + 1
```

#Iterative Approach (BFS - Level Order):

```
FUNCTION maxDepthIterative(root):
    IF root is NULL:
        RETURN 0

    QUEUE ← [(root, 1)]  // Store node with depth
    MAX_DEPTH ← 0

    WHILE QUEUE is NOT empty:
        NODE, DEPTH ← DEQUEUE QUEUE
        MAX_DEPTH ← MAX(MAX_DEPTH, DEPTH)

        IF NODE.left is NOT NULL:
            ENQUEUE (NODE.left, DEPTH + 1) TO QUEUE

        IF NODE.right is NOT NULL:
            ENQUEUE (NODE.right, DEPTH + 1) TO QUEUE

    RETURN MAX_DEPTH
```

2. Minimum Depth of a Binary Tree:
#Recursive Approach:

```
FUNCTION minDepth(root):
    IF root is NULL:
        RETURN 0

    IF root.left is NULL:
        RETURN minDepth(root.right) + 1

    IF root.right is NULL:
        RETURN minDepth(root.left) + 1

    RETURN MIN(minDepth(root.left), minDepth(root.right)) + 1
```

#Iterative Approach (BFS - Level Order):

```
FUNCTION minDepthIterative(root):
    IF root is NULL:
        RETURN 0

    QUEUE ← [(root, 1)]

    WHILE QUEUE is NOT empty:
        NODE, DEPTH ← DEQUEUE QUEUE

        IF NODE.left is NULL AND NODE.right is NULL:
            RETURN DEPTH  // First leaf node found

        IF NODE.left is NOT NULL:
            ENQUEUE (NODE.left, DEPTH + 1) TO QUEUE

        IF NODE.right is NOT NULL:
            ENQUEUE (NODE.right, DEPTH + 1) TO QUEUE
```

3. Depth of a Specific Node in a Binary Tree:

```
FUNCTION findDepth(root, target, depth):
    IF root is NULL:
        RETURN -1  // Target not found

    IF root.value == target:
        RETURN depth

    LEFT_SEARCH ← findDepth(root.left, target, depth + 1)
    IF LEFT_SEARCH ≠ -1:
        RETURN LEFT_SEARCH

    RETURN findDepth(root.right, target, depth + 1)
```

>>>[9.11].Check/Compare Binary Trees:------------------------------------------------------------------------------------------------------

1. Check if Two Binary Trees are Identical:

```
FUNCTION isIdentical(tree1, tree2):
    IF tree1 is NULL AND tree2 is NULL:
```

```
        RETURN True  // Both trees are empty, so they are identical

    IF tree1 is NULL OR tree2 is NULL:
        RETURN False  // One tree is empty, the other is not

    IF tree1.value ≠ tree2.value:
        RETURN False  // Values do not match

    RETURN isIdentical(tree1.left, tree2.left) AND isIdentical(tree1.right,
tree2.right)
```

2. Check if One Tree is a Subtree of Another:

```
FUNCTION isSubtree(root, subRoot):
    IF subRoot is NULL:
        RETURN True  // An empty tree is always a subtree

    IF root is NULL:
        RETURN False  // subRoot exists, but root is empty, so it's not a subtree

    IF isIdentical(root, subRoot):
        RETURN True

    RETURN isSubtree(root.left, subRoot) OR isSubtree(root.right, subRoot)
```

3. Check if Two Trees are Mirror Images:

```
FUNCTION isMirror(tree1, tree2):
    IF tree1 is NULL AND tree2 is NULL:
        RETURN True  // Both trees are empty

    IF tree1 is NULL OR tree2 is NULL:
        RETURN False  // One tree is empty, the other is not

    IF tree1.value ≠ tree2.value:
        RETURN False  // Values do not match

    RETURN isMirror(tree1.left, tree2.right) AND isMirror(tree1.right, tree2.left)
```

4. Check if a Tree is Symmetric:

```
FUNCTION isSymmetric(root):
    IF root is NULL:
        RETURN True

    RETURN isMirror(root.left, root.right)
```

-------------------------------------------------------------------------------------
---------------------------------------------------------------------------
//ALOGORITHMS://
-------------------------------------------------------------------------------------
---------------------------------------------------------------------------
1.Pattern searching
2.Divide and Conquer
3.Searching
4.Sorting
5.Bitwise
6.Greedy
7.Recursion

8.Backtracking
9.Mathematical
10.Dynamic Programming

------------------------------------------------------------------------------------------------------------------------------------------------------------

1.PATTERN SEARCHING:

1. Naive Pattern Searching Algorithm:

```
FUNCTION naivePatternSearch(text, pattern):
    N ← LENGTH(text)
    M ← LENGTH(pattern)

    FOR i FROM 0 TO (N - M):
        MATCH ← True

        FOR j FROM 0 TO (M - 1):
            IF text[i + j] ≠ pattern[j]:
                MATCH ← False
                BREAK

        IF MATCH:
            PRINT "Pattern found at index", i
```

2. Knuth-Morris-Pratt (KMP) Algorithm:

```
FUNCTION computeLPS(pattern):
    M ← LENGTH(pattern)
    LPS[M] ← [0]  // Array to store prefix function
    LEN ← 0
    i ← 1

    WHILE i < M:
        IF pattern[i] == pattern[LEN]:
            LEN ← LEN + 1
            LPS[i] ← LEN
            i ← i + 1
        ELSE:
            IF LEN ≠ 0:
                LEN ← LPS[LEN - 1]
            ELSE:
                LPS[i] ← 0
                i ← i + 1

    RETURN LPS


FUNCTION KMPSearch(text, pattern):
    N ← LENGTH(text)
    M ← LENGTH(pattern)
    LPS ← computeLPS(pattern)

    i ← 0 // Index for text
    j ← 0 // Index for pattern

    WHILE i < N:
        IF text[i] == pattern[j]:
            i ← i + 1
```

```
            j ← j + 1

        IF j == M:
            PRINT "Pattern found at index", i - j
            j ← LPS[j - 1]

        ELSE IF i < N AND text[i] ≠ pattern[j]:
            IF j ≠ 0:
                j ← LPS[j - 1]
            ELSE:
                i ← i + 1
```

3. Rabin-Karp Algorithm (Rolling Hash):

```
FUNCTION rabinKarpSearch(text, pattern, prime):
    N ← LENGTH(text)
    M ← LENGTH(pattern)
    BASE ← 256  // Number of characters in the input alphabet
    HASH_TEXT ← 0
    HASH_PATTERN ← 0
    H ← 1

    // Precompute H = BASE^(M-1) % prime
    FOR i FROM 0 TO M - 2:
        H ← (H * BASE) % prime

    // Compute initial hash values
    FOR i FROM 0 TO M - 1:
        HASH_PATTERN ← (BASE * HASH_PATTERN + ASCII(pattern[i])) % prime
        HASH_TEXT ← (BASE * HASH_TEXT + ASCII(text[i])) % prime

    // Sliding window over text
    FOR i FROM 0 TO N - M:
        IF HASH_PATTERN == HASH_TEXT:
            MATCH ← True
            FOR j FROM 0 TO M - 1:
                IF text[i + j] ≠ pattern[j]:
                    MATCH ← False
                    BREAK
            IF MATCH:
                PRINT "Pattern found at index", i

        // Compute next hash using rolling hash formula
        IF i < N - M:
            HASH_TEXT ← (BASE * (HASH_TEXT - ASCII(text[i]) * H) + ASCII(text[i +
M])) % prime
            IF HASH_TEXT < 0:
                HASH_TEXT ← HASH_TEXT + prime
```

4. Boyer-Moore Algorithm (Efficient for Large Texts):

```
FUNCTION computeBadCharTable(pattern):
    M ← LENGTH(pattern)
    BAD_CHAR ← ARRAY of size 256 initialized to -1

    FOR i FROM 0 TO M - 1:
        BAD_CHAR[ASCII(pattern[i])] ← i

    RETURN BAD_CHAR
```

```
FUNCTION boyerMooreSearch(text, pattern):
    N ← LENGTH(text)
    M ← LENGTH(pattern)
    BAD_CHAR ← computeBadCharTable(pattern)

    SHIFT ← 0

    WHILE SHIFT <= N - M:
        j ← M - 1

        WHILE j >= 0 AND pattern[j] == text[SHIFT + j]:
            j ← j - 1

        IF j < 0:
            PRINT "Pattern found at index", SHIFT
            SHIFT ← SHIFT + (M - BAD_CHAR[ASCII(text[SHIFT + M])] IF SHIFT + M < N
ELSE 1)
        ELSE:
            SHIFT ← SHIFT + MAX(1, j - BAD_CHAR[ASCII(text[SHIFT + j])])
```

-------------------------------------------------------------------------------------
-----------------------------------------------------------------------------

2.DIVIDE AND CONQUER

1. Merge Sort (Divide and Conquer Sorting Algorithm):

```
FUNCTION mergeSort(arr, left, right):
    IF left >= right:
        RETURN

    MID ← (left + right) / 2

    mergeSort(arr, left, MID)
    mergeSort(arr, MID + 1, right)

    merge(arr, left, MID, right)

FUNCTION merge(arr, left, mid, right):
    LEFT_PART ← arr[left to mid]
    RIGHT_PART ← arr[mid+1 to right]

    i ← 0, j ← 0, k ← left

    WHILE i < LENGTH(LEFT_PART) AND j < LENGTH(RIGHT_PART):
        IF LEFT_PART[i] ≤ RIGHT_PART[j]:
            arr[k] ← LEFT_PART[i]
            i ← i + 1
        ELSE:
            arr[k] ← RIGHT_PART[j]
            j ← j + 1
        k ← k + 1

    WHILE i < LENGTH(LEFT_PART):
        arr[k] ← LEFT_PART[i]
        i ← i + 1
        k ← k + 1
```

```
    WHILE j < LENGTH(RIGHT_PART):
        arr[k] ← RIGHT_PART[j]
        j ← j + 1
        k ← k + 1
```

2. Quick Sort (Divide and Conquer Sorting Algorithm):

```
FUNCTION quickSort(arr, left, right):
    IF left >= right:
        RETURN

    PIVOT_INDEX ← partition(arr, left, right)

    quickSort(arr, left, PIVOT_INDEX - 1)
    quickSort(arr, PIVOT_INDEX + 1, right)

FUNCTION partition(arr, left, right):
    PIVOT ← arr[right]
    i ← left - 1

    FOR j FROM left TO right - 1:
        IF arr[j] ≤ PIVOT:
            i ← i + 1
            SWAP arr[i] WITH arr[j]

    SWAP arr[i + 1] WITH arr[right]
    RETURN i + 1
```

3. Binary Search (Divide and Conquer Searching Algorithm):

```
FUNCTION binarySearch(arr, left, right, target):
    IF left > right:
        RETURN -1  // Target not found

    MID ← (left + right) / 2

    IF arr[MID] == target:
        RETURN MID
    ELSE IF arr[MID] > target:
        RETURN binarySearch(arr, left, MID - 1, target)
    ELSE:
        RETURN binarySearch(arr, MID + 1, right, target)
```

4. Matrix Multiplication (Strassen’s Algorithm):

```
FUNCTION strassenMultiplication(A, B):
    IF SIZE(A) == 1:
        RETURN A * B

    Divide A and B into submatrices:
        A11, A12, A21, A22
        B11, B12, B21, B22

    Compute 7 matrix multiplications:
        P1 = strassenMultiplication(A11 + A22, B11 + B22)
        P2 = strassenMultiplication(A21 + A22, B11)
        P3 = strassenMultiplication(A11, B12 - B22)
        P4 = strassenMultiplication(A22, B21 - B11)
        P5 = strassenMultiplication(A11 + A12, B22)
```

```
        P6 = strassenMultiplication(A21 - A11, B11 + B12)
        P7 = strassenMultiplication(A12 - A22, B21 + B22)

    Compute final submatrices:
        C11 = P1 + P4 - P5 + P7
        C12 = P3 + P5
        C21 = P2 + P4
        C22 = P1 - P2 + P3 + P6

    RETURN Matrix C (formed by C11, C12, C21, C22)
```

5. Closest Pair of Points (Divide and Conquer):

```
FUNCTION closestPair(points):
    SORT points by x-coordinates
    RETURN closestPairRecursive(points)

FUNCTION closestPairRecursive(points):
    IF LENGTH(points) ≤ 3:
        RETURN bruteForceClosestPair(points)

    MID ← LENGTH(points) / 2
    LEFT_HALF ← points[0 to MID]
    RIGHT_HALF ← points[MID to END]

    LEFT_DISTANCE ← closestPairRecursive(LEFT_HALF)
    RIGHT_DISTANCE ← closestPairRecursive(RIGHT_HALF)

    MIN_DISTANCE ← MIN(LEFT_DISTANCE, RIGHT_DISTANCE)

    RETURN mergeClosest(points, MIN_DISTANCE)
```
-----------------------------------------------------------------------------------------------------------------------------------------------

3.SEARCHING:

1. Linear Search (Sequential Search):

```
FUNCTION linearSearch(arr, target):
    FOR i FROM 0 TO LENGTH(arr) - 1:
        IF arr[i] == target:
            RETURN i  // Found at index i
    RETURN -1  // Not found
```

2. Binary Search (For Sorted Arrays):

```
FUNCTION binarySearch(arr, left, right, target):
    WHILE left ≤ right:
        MID ← (left + right) / 2

        IF arr[MID] == target:
            RETURN MID  // Found
        ELSE IF arr[MID] > target:
            right ← MID - 1  // Search left half
        ELSE:
            left ← MID + 1  // Search right half

    RETURN -1  // Not found
```

#Recursive Binary Search:

```
FUNCTION binarySearchRecursive(arr, left, right, target):
    IF left > right:
        RETURN -1  // Not found

    MID ← (left + right) / 2

    IF arr[MID] == target:
        RETURN MID
    ELSE IF arr[MID] > target:
        RETURN binarySearchRecursive(arr, left, MID - 1, target)
    ELSE:
        RETURN binarySearchRecursive(arr, MID + 1, right, target)
```

3. Interpolation Search (For Uniformly Distributed Data):

```
FUNCTION interpolationSearch(arr, left, right, target):
    WHILE left ≤ right AND target ≥ arr[left] AND target ≤ arr[right]:
        POS ← left + ((target - arr[left]) * (right - left)) / (arr[right] -
arr[left])

        IF arr[POS] == target:
            RETURN POS  // Found
        ELSE IF arr[POS] > target:
            right ← POS - 1  // Search left half
        ELSE:
            left ← POS + 1  // Search right half

    RETURN -1  // Not found
```

4. Jump Search (For Sorted Arrays):

```
FUNCTION jumpSearch(arr, target):
    N ← LENGTH(arr)
    STEP ← FLOOR(SQRT(N))
    PREV ← 0

    WHILE arr[MIN(STEP, N) - 1] < target:
        PREV ← STEP
        STEP ← STEP + FLOOR(SQRT(N))
        IF PREV ≥ N:
            RETURN -1  // Not found

    FOR i FROM PREV TO MIN(STEP, N) - 1:
        IF arr[i] == target:
            RETURN i  // Found

    RETURN -1  // Not found
```

5. Exponential Search (For Unbounded or Infinite Arrays):

```
FUNCTION exponentialSearch(arr, target):
    IF arr[0] == target:
        RETURN 0  // Found

    i ← 1
    WHILE i < LENGTH(arr) AND arr[i] ≤ target:
        i ← i * 2  // Exponentially increase index

    RETURN binarySearch(arr, i/2, MIN(i, LENGTH(arr)-1), target)
```

6. Ternary Search (Alternative to Binary Search):

```
FUNCTION ternarySearch(arr, left, right, target):
    WHILE left ≤ right:
        MID1 ← left + (right - left) / 3
        MID2 ← right - (right - left) / 3

        IF arr[MID1] == target:
            RETURN MID1
        ELSE IF arr[MID2] == target:
            RETURN MID2
        ELSE IF target < arr[MID1]:
            right ← MID1 - 1
        ELSE IF target > arr[MID2]:
            left ← MID2 + 1
        ELSE:
            left ← MID1 + 1
            right ← MID2 - 1

    RETURN -1  // Not found
```

-----------------------------------------------------------------------------------
-------------------------------------------------------------------------
4.SORTING:

1. Bubble Sort (Simple but Inefficient):

```
FUNCTION bubbleSort(arr):
    N ← LENGTH(arr)

    FOR i FROM 0 TO N-1:
        SWAPPED ← FALSE
        FOR j FROM 0 TO N-i-2:
            IF arr[j] > arr[j+1]:
                SWAP arr[j] WITH arr[j+1]
                SWAPPED ← TRUE

        IF SWAPPED == FALSE:
            BREAK  // Optimization: Stop if already sorted
```

2. Selection Sort (Find Minimum and Swap):

```
FUNCTION selectionSort(arr):
    N ← LENGTH(arr)

    FOR i FROM 0 TO N-1:
        MIN_INDEX ← i
        FOR j FROM i+1 TO N-1:
            IF arr[j] < arr[MIN_INDEX]:
                MIN_INDEX ← j
        SWAP arr[i] WITH arr[MIN_INDEX]
```

3. Insertion Sort (Efficient for Nearly Sorted Data):

```
FUNCTION insertionSort(arr):
    N ← LENGTH(arr)

    FOR i FROM 1 TO N-1:
```

```
        KEY ← arr[i]
        j ← i - 1

        WHILE j >= 0 AND arr[j] > KEY:
            arr[j+1] ← arr[j]
            j ← j - 1

        arr[j+1] ← KEY
```

4. Merge Sort (Divide and Conquer):

```
FUNCTION mergeSort(arr, left, right):
    IF left >= right:
        RETURN

    MID ← (left + right) / 2

    mergeSort(arr, left, MID)
    mergeSort(arr, MID + 1, right)

    merge(arr, left, MID, right)

FUNCTION merge(arr, left, mid, right):
    LEFT_PART ← arr[left to mid]
    RIGHT_PART ← arr[mid+1 to right]

    i ← 0, j ← 0, k ← left

    WHILE i < LENGTH(LEFT_PART) AND j < LENGTH(RIGHT_PART):
        IF LEFT_PART[i] ≤ RIGHT_PART[j]:
            arr[k] ← LEFT_PART[i]
            i ← i + 1
        ELSE:
            arr[k] ← RIGHT_PART[j]
            j ← j + 1
        k ← k + 1

    WHILE i < LENGTH(LEFT_PART):
        arr[k] ← LEFT_PART[i]
        i ← i + 1
        k ← k + 1

    WHILE j < LENGTH(RIGHT_PART):
        arr[k] ← RIGHT_PART[j]
        j ← j + 1
        k ← k + 1
```

5. Quick Sort (Divide and Conquer):

```
FUNCTION quickSort(arr, left, right):
    IF left >= right:
        RETURN

    PIVOT_INDEX ← partition(arr, left, right)

    quickSort(arr, left, PIVOT_INDEX - 1)
    quickSort(arr, PIVOT_INDEX + 1, right)

FUNCTION partition(arr, left, right):
```

```
    PIVOT ← arr[right]
    i ← left - 1

    FOR j FROM left TO right - 1:
        IF arr[j] ≤ PIVOT:
            i ← i + 1
            SWAP arr[i] WITH arr[j]

    SWAP arr[i + 1] WITH arr[right]
    RETURN i + 1
```

6. Heap Sort (Uses a Binary Heap):

```
FUNCTION heapSort(arr):
    N ← LENGTH(arr)

    // Build max heap
    FOR i FROM N/2 DOWN TO 0:
        heapify(arr, N, i)

    // Extract elements one by one
    FOR i FROM N-1 DOWN TO 1:
        SWAP arr[0] WITH arr[i]
        heapify(arr, i, 0)

FUNCTION heapify(arr, N, i):
    LARGEST ← i
    LEFT ← 2*i + 1
    RIGHT ← 2*i + 2

    IF LEFT < N AND arr[LEFT] > arr[LARGEST]:
        LARGEST ← LEFT
    IF RIGHT < N AND arr[RIGHT] > arr[LARGEST]:
        LARGEST ← RIGHT
    IF LARGEST ≠ i:
        SWAP arr[i] WITH arr[LARGEST]
        heapify(arr, N, LARGEST)
```

7. Counting Sort (For Small Integer Ranges):

```
FUNCTION countingSort(arr, maxValue):
    COUNT ← ARRAY of size (maxValue + 1) initialized to 0

    FOR each element IN arr:
        COUNT[element] ← COUNT[element] + 1

    INDEX ← 0
    FOR i FROM 0 TO maxValue:
        WHILE COUNT[i] > 0:
            arr[INDEX] ← i
            INDEX ← INDEX + 1
            COUNT[i] ← COUNT[i] - 1
```

8. Radix Sort (Sorts Numbers Digit by Digit):

```
FUNCTION radixSort(arr):
    MAX_VALUE ← FIND_MAX(arr)
    EXP ← 1
```

```
    WHILE MAX_VALUE / EXP > 0:
        countingSortByDigit(arr, EXP)
        EXP ← EXP * 10

FUNCTION countingSortByDigit(arr, EXP):
    OUTPUT ← ARRAY of size LENGTH(arr)
    COUNT ← ARRAY[10] initialized to 0

    FOR each element IN arr:
        DIGIT ← (element / EXP) % 10
        COUNT[DIGIT] ← COUNT[DIGIT] + 1

    FOR i FROM 1 TO 9:
        COUNT[i] ← COUNT[i] + COUNT[i-1]

    FOR i FROM LENGTH(arr)-1 DOWN TO 0:
        DIGIT ← (arr[i] / EXP) % 10
        OUTPUT[COUNT[DIGIT] - 1] ← arr[i]
        COUNT[DIGIT] ← COUNT[DIGIT] - 1

    FOR i FROM 0 TO LENGTH(arr)-1:
        arr[i] ← OUTPUT[i]
```

-------------------------------------------------------------------------------------------------------------------------------------------------

5.BITWISE:

1. Check if a Number is Even or Odd (Using AND):

```
FUNCTION isEven(num):
    RETURN (num AND 1) == 0
```

2. Swap Two Numbers Without Using a Temporary Variable:

```
FUNCTION swap(a, b):
    a ← a XOR b
    b ← a XOR b
    a ← a XOR b
    RETURN (a, b)
```

3. Check if a Number is a Power of Two:

```
FUNCTION isPowerOfTwo(n):
    RETURN (n > 0) AND (n AND (n - 1)) == 0
```

4. Count the Number of Set Bits (Hamming Weight):

```
FUNCTION countSetBits(n):
    COUNT ← 0
    WHILE n > 0:
        n ← n AND (n - 1)
        COUNT ← COUNT + 1
    RETURN COUNT
```

5. Find the Single Non-Repeating Number in an Array (XOR):

```
FUNCTION findUnique(arr):
    UNIQUE ← 0
    FOR num IN arr:
```

```
        UNIQUE ← UNIQUE XOR num
    RETURN UNIQUE
```

6. Reverse Bits of a Number:

```
FUNCTION reverseBits(num):
    REVERSED ← 0
    FOR i FROM 0 TO 31:
        REVERSED ← (REVERSED << 1) OR (num AND 1)
        num ← num >> 1
    RETURN REVERSED
```

7. Find the Missing Number in an Array of 1 to N:

```
FUNCTION findMissing(arr, N):
    XOR_ALL ← 0
    FOR i FROM 1 TO N:
        XOR_ALL ← XOR_ALL XOR i

    XOR_ARR ← 0
    FOR num IN arr:
        XOR_ARR ← XOR_ARR XOR num

    RETURN XOR_ALL XOR XOR_ARR
```

8. Compute XOR from 1 to N

```
FUNCTION xorFrom1ToN(N):
    IF N MOD 4 == 0:
        RETURN N
    ELSE IF N MOD 4 == 1:
        RETURN 1
    ELSE IF N MOD 4 == 2:
        RETURN N + 1
    ELSE:
        RETURN 0
```

9. Check if Two Numbers Differ by One Bit (Hamming Distance = 1):

```
FUNCTION differsByOneBit(x, y):
    DIFF ← x XOR y
    RETURN (DIFF AND (DIFF - 1)) == 0 AND DIFF > 0
```

10. Flip the K-th Bit of a Number:

```
FUNCTION flipKthBit(num, K):
    RETURN num XOR (1 << K)
```

------------------------------------------------------------------------------------------------------------------------------------------------------------

6.GREEDY:

1. Activity Selection Problem (Maximum Non-Overlapping Intervals)

```
FUNCTION maxActivities(activities):
    SORT activities BY end_time
    SELECTED ← 0
    LAST_END ← -∞
```

```
    FOR EACH activity IN activities:
        IF activity.start ≥ LAST_END:
            SELECT activity
            LAST_END ← activity.end
            SELECTED ← SELECTED + 1

    RETURN SELECTED
```

2. Huffman Coding (Optimal Prefix Code):

```
FUNCTION huffmanCoding(frequencies):
    MIN_HEAP ← create priority queue from frequencies

    WHILE MIN_HEAP.size > 1:
        LEFT ← MIN_HEAP.extract_min()
        RIGHT ← MIN_HEAP.extract_min()
        NEW_NODE ← merge(LEFT, RIGHT)
        MIN_HEAP.insert(NEW_NODE)

    RETURN generateHuffmanCodes(MIN_HEAP.extract_min())
```

3. Fractional Knapsack Problem:

```
FUNCTION fractionalKnapsack(items, capacity):
    SORT items BY (value / weight) DESCENDING
    TOTAL_VALUE ← 0
    REMAINING_CAPACITY ← capacity

    FOR EACH item IN items:
        IF REMAINING_CAPACITY ≥ item.weight:
            TOTAL_VALUE ← TOTAL_VALUE + item.value
            REMAINING_CAPACITY ← REMAINING_CAPACITY - item.weight
        ELSE:
            TOTAL_VALUE ← TOTAL_VALUE + (item.value / item.weight) *
REMAINING_CAPACITY
            BREAK

    RETURN TOTAL_VALUE
```

4. Minimum Number of Coins (Change Making Problem):

```
FUNCTION minCoins(coins, V):
    SORT coins DESCENDING
    COUNT ← 0

    FOR EACH coin IN coins:
        WHILE V ≥ coin:
            V ← V - coin
            COUNT ← COUNT + 1
        IF V == 0:
            BREAK

    RETURN COUNT
```

5. Job Sequencing with Deadlines:

```
FUNCTION jobSequencing(jobs):
    SORT jobs BY profit DESCENDING
    MAX_DEADLINE ← findMaxDeadline(jobs)
```

```
    SLOTS ← array of size MAX_DEADLINE initialized to -1
    TOTAL_PROFIT ← 0

    FOR EACH job IN jobs:
        FOR t FROM job.deadline TO 1:
            IF SLOTS[t] == -1:
                SLOTS[t] ← job
                TOTAL_PROFIT ← TOTAL_PROFIT + job.profit
                BREAK

    RETURN TOTAL_PROFIT
```

6. Connecting N Ropes (Minimum Cost)

```
FUNCTION minRopeCost(ropes):
    MIN_HEAP ← create priority queue from ropes
    TOTAL_COST ← 0

    WHILE MIN_HEAP.size > 1:
        FIRST ← MIN_HEAP.extract_min()
        SECOND ← MIN_HEAP.extract_min()
        NEW_ROPE ← FIRST + SECOND
        TOTAL_COST ← TOTAL_COST + NEW_ROPE
        MIN_HEAP.insert(NEW_ROPE)

    RETURN TOTAL_COST
```

--------------------------------------------------------------------------------------------------------------------------------------------------

7.RECURSION:

1. Factorial of a Number (n!):

```
FUNCTION factorial(n):
    IF n == 0:
        RETURN 1
    RETURN n * factorial(n-1)
```

2. Fibonacci Sequence (Nth Term):

```
FUNCTION fibonacci(n):
    IF n == 0:
        RETURN 0
    IF n == 1:
        RETURN 1
    RETURN fibonacci(n-1) + fibonacci(n-2)
```

3. Power Function (Exponentiation):

```
FUNCTION power(x, n):
    IF n == 0:
        RETURN 1
    RETURN x * power(x, n-1)
```

#Optimized approach (Divide & Conquer):

```
FUNCTION power(x, n):
    IF n == 0:
        RETURN 1
```

```
    HALF ← power(x, n // 2)
    IF n is EVEN:
        RETURN HALF * HALF
    ELSE:
        RETURN HALF * HALF * x
```

4. Reverse a String:

```
FUNCTION reverseString(s):
    IF length(s) ≤ 1:
        RETURN s
    RETURN reverseString(s[1:]) + s[0]
```

5. Sum of Digits:

```
FUNCTION sumOfDigits(n):
    IF n < 10:
        RETURN n
    RETURN (n % 10) + sumOfDigits(n // 10)
```

6. Tower of Hanoi (Move Disks):

```
FUNCTION towerOfHanoi(N, Source, Auxiliary, Destination):
    IF N == 1:
        PRINT "Move disk 1 from", Source, "to", Destination
        RETURN
    towerOfHanoi(N-1, Source, Destination, Auxiliary)
    PRINT "Move disk", N, "from", Source, "to", Destination
    towerOfHanoi(N-1, Auxiliary, Source, Destination)
```

7. Generate All Subsets of a Set (Power Set):

```
FUNCTION generateSubsets(arr, index, subset):
    IF index == length(arr):
        PRINT subset
        RETURN

    // Exclude the current element
    generateSubsets(arr, index+1, subset)

    // Include the current element
    generateSubsets(arr, index+1, subset + [arr[index]])
```

8. Print All Permutations of a String:

```
FUNCTION permute(s, left, right):
    IF left == right:
        PRINT s
        RETURN

    FOR i FROM left TO right:
        SWAP(s[left], s[i])
        permute(s, left+1, right)
        SWAP(s[left], s[i])  // Backtrack
```

------------------------------------------------------------------------------------
----------------------------------------------------------------------------

8.BACKTRACKING

 1).Simple Backtracking
 2).Constraint-based Backtracking
 3).Optimized Backtracking (Branch and Bound)
 4).Forward Checking
 5).Backtracking with Heuristics

>>>[8.1] Recursive BackTracking:

Pseudocode for implementing a Recursive Backtracking pattern:

```
def solve_backtrack():
    if is_solved(): // base case: solution has been found
        return True

    for candidate in candidates:
        if not is_feasible(candidate): // if candidate is not a valid choice
            continue

        accept(candidate) // use candidate for the current instance
        if solve_backtrack(): // recursive call
            return True // if problem solved, return True
        reject(candidate) // problem not solved, remove candidate used and continue
in loop

    return False // solution does not exist
```

1).Simple Backtracking

```
FUNCTION generateSubsets(arr, index, subset):
    IF index == length(arr):
        PRINT subset
        RETURN

    // Include the current element
    generateSubsets(arr, index+1, subset + [arr[index]])

    // Exclude the current element (Backtrack)
    generateSubsets(arr, index+1, subset)
```

2).Constraint-based Backtracking

```
FUNCTION solveNQueens(board, row):
    IF row == N:
        PRINT board
        RETURN True

    FOR col FROM 0 TO N-1:
        IF isSafe(board, row, col):  // Check constraints
            board[row][col] = 'Q'
            IF solveNQueens(board, row + 1):
                RETURN True
            board[row][col] = '.'  // Backtrack

    RETURN False
```

3).Optimized Backtracking (Branch and Bound)

```
FUNCTION knapsack(index, weight, value, capacity):
    IF index == N OR weight > capacity:
```

```
        RETURN value  // Stop exploring further

    // Bound condition to prune branches
    IF bound(index, weight, value, capacity) < maxValue:
        RETURN  // Do not explore further

    // Include the current item
    include = knapsack(index+1, weight+arr[index].weight, value+arr[index].value,
capacity)

    // Exclude the current item (Backtrack)
    exclude = knapsack(index+1, weight, value, capacity)

    RETURN max(include, exclude)
```

4).Forward Checking

```
FUNCTION solveSudoku(board):
    FOR row FROM 0 TO 8:
        FOR col FROM 0 TO 8:
            IF board[row][col] == '.':
                FOR num FROM '1' TO '9':
                    IF isValid(board, row, col, num):
                        board[row][col] = num
                        forwardCheck(board)  // Reduce future choices
                        IF solveSudoku(board):
                            RETURN True
                        board[row][col] = '.'  // Backtrack
                RETURN False
    RETURN True
```

5).Backtracking with Heuristics

```
FUNCTION backtrack(assignment):
    IF assignment is complete:
        RETURN assignment

    var ← selectVariable(MRV)  // Choose the most constrained variable
    FOR value IN orderValues(var, LCV):  // Try least constraining value first
        IF isConsistent(var, value, assignment):
            assignment[var] = value
            IF backtrack(assignment):
                RETURN True
            assignment[var] = None  // Backtrack

    RETURN False
```

1. N-Queens Problem:

```
FUNCTION solveNQueens(board, row):
    IF row == N:
        PRINT board
        RETURN

    FOR col FROM 0 TO N-1:
        IF isSafe(board, row, col):
            board[row][col] = 'Q'
            solveNQueens(board, row + 1)
```

```
            board[row][col] = '.'  // Backtrack
```

2. Sudoku Solver:

```
FUNCTION solveSudoku(board):
    FOR row FROM 0 TO 8:
        FOR col FROM 0 TO 8:
            IF board[row][col] == '.':
                FOR num FROM '1' TO '9':
                    IF isValid(board, row, col, num):
                        board[row][col] = num
                        IF solveSudoku(board):
                            RETURN True
                        board[row][col] = '.'  // Backtrack
                RETURN False
    RETURN True
```

3. Generate All Subsets (Power Set):

```
FUNCTION generateSubsets(arr, index, subset):
    IF index == length(arr):
        PRINT subset
        RETURN

    // Include the current element
    generateSubsets(arr, index+1, subset + [arr[index]])

    // Exclude the current element (Backtrack)
    generateSubsets(arr, index+1, subset)
```

4. Word Search (Find a word in a grid):

```
FUNCTION exist(board, word):
    FOR i FROM 0 TO M-1:
        FOR j FROM 0 TO N-1:
            IF search(board, word, i, j, 0):
                RETURN True
    RETURN False

FUNCTION search(board, word, i, j, index):
    IF index == length(word):
        RETURN True

    IF i < 0 OR j < 0 OR i >= M OR j >= N OR board[i][j] != word[index]:
        RETURN False

    TEMP ← board[i][j]
    board[i][j] ← '#'  // Mark as visited

    IF search(board, word, i+1, j, index+1) OR
       search(board, word, i-1, j, index+1) OR
       search(board, word, i, j+1, index+1) OR
       search(board, word, i, j-1, index+1):
        RETURN True

    board[i][j] ← TEMP  // Backtrack
    RETURN False
```

5. Permutations of a String:

```
FUNCTION permute(s, left, right):
    IF left == right:
        PRINT s
        RETURN

    FOR i FROM left TO right:
        SWAP(s[left], s[i])
        permute(s, left+1, right)
        SWAP(s[left], s[i])  // Backtrack
```

6. Combination Sum (Find subsets that sum to target):

```
FUNCTION combinationSum(arr, index, target, subset):
    IF target == 0:
        PRINT subset
        RETURN

    FOR i FROM index TO length(arr)-1:
        IF arr[i] ≤ target:
            subset.append(arr[i])
            combinationSum(arr, i, target - arr[i], subset)
            subset.pop()  // Backtrack
```

7. Rat in a Maze:

```
FUNCTION solveMaze(maze, x, y, path):
    IF x == N-1 AND y == N-1:
        PRINT path
        RETURN True

    IF isValidMove(maze, x, y):
        maze[x][y] = 0  // Mark as visited

        IF solveMaze(maze, x+1, y, path + "D") OR
           solveMaze(maze, x, y+1, path + "R") OR
           solveMaze(maze, x-1, y, path + "U") OR
           solveMaze(maze, x, y-1, path + "L"):
            RETURN True

        maze[x][y] = 1  // Backtrack
    RETURN False
```

9.Mathematical
10.Dynamic Programming

-----------------------------------------------------------------------------------
--------------------------------------------------------------------------

9.MATHEMATICAL:

1) Greatest Common Divisor (GCD) - Euclidean Algorithm:

```
FUNCTION GCD(a, b):
    IF b == 0:
        RETURN a
    RETURN GCD(b, a MOD b)
```

2) Least Common Multiple (LCM):

```
FUNCTION LCM(a, b):
    RETURN (a * b) / GCD(a, b)
```

3) Prime Number Check:

```
FUNCTION isPrime(N):
    IF N <= 1:
        RETURN False
    FOR i FROM 2 TO sqrt(N):
        IF N MOD i == 0:
            RETURN False
    RETURN True
```

4) Sieve of Eratosthenes (Find all primes ≤ N):

```
FUNCTION sieve(N):
    isPrime = ARRAY of size N+1 initialized to True
    isPrime[0] = isPrime[1] = False

    FOR i FROM 2 TO sqrt(N):
        IF isPrime[i] == True:
            FOR j FROM i*i TO N STEP i:
                isPrime[j] = False

    PRINT all numbers where isPrime[i] is True
```

5) Fast Exponentiation (Modular Exponentiation):

```
FUNCTION power(A, B, M):
    result = 1
    WHILE B > 0:
        IF B MOD 2 == 1:  // If B is odd
            result = (result * A) MOD M
        A = (A * A) MOD M
        B = B / 2
    RETURN result
```

6) Factorial Computation:

```
FUNCTION factorial(N):
    IF N == 0:
        RETURN 1
    RETURN N * factorial(N - 1)
```

7) Fibonacci Numbers (Optimized using Matrix Exponentiation):

```
FUNCTION fib(N):
    MATRIX F = [[1, 1], [1, 0]]
    RETURN matrixPower(F, N-1)[0][0]

FUNCTION matrixPower(F, P):
    RESULT = identityMatrix()
    WHILE P > 0:
        IF P MOD 2 == 1:
            RESULT = matrixMultiply(RESULT, F)
        F = matrixMultiply(F, F)
        P = P / 2
    RETURN RESULT
```

8) Counting Number of Divisors:

```
FUNCTION countDivisors(N):
    count = 0
    FOR i FROM 1 TO sqrt(N):
        IF N MOD i == 0:
            count = count + 1
            IF i != N / i:
                count = count + 1
    RETURN count
```

9) Sum of Divisors:

```
FUNCTION sumOfDivisors(N):
    sum = 0
    FOR i FROM 1 TO sqrt(N):
        IF N MOD i == 0:
            sum = sum + i
            IF i != N / i:
                sum = sum + (N / i)
    RETURN sum
```

10) Greatest Integer Function (Floor Division):

```
FUNCTION floorDivide(A, B):
    RETURN A // B  // Integer division
```

----------------------------------------------------------------------------------------------------------------------------------------------------------------

10.DYNAMIC PROGRAMMING:

1).Longest Increasing Subsequence
2).Longest common subsequence
3).Palindrome
4).Fibonacci
5).0/1 Knapsack (Bounded/Unbounded)
6).Coin Change
7).Matrix Multiplication
8).DP on Grid
9).DP on Trees
10).DP on Graphs
11).DP + Hashmap
12).DP + Bitmasking
13).Digit DP

1) Longest Increasing Subsequence (LIS) – $O(N^2)$:

```
FUNCTION LIS(arr, N):
    dp = ARRAY of size N initialized to 1
    FOR i FROM 1 TO N:
        FOR j FROM 0 TO i-1:
            IF arr[j] < arr[i]:
                dp[i] = MAX(dp[i], dp[j] + 1)
    RETURN MAX(dp)
```

2) Longest Common Subsequence (LCS) – O(N*M)

```
FUNCTION LCS(s1, s2, N, M):
    dp = ARRAY of size (N+1) x (M+1) initialized to 0
```

```
    FOR i FROM 1 TO N:
        FOR j FROM 1 TO M:
            IF s1[i-1] == s2[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1]
            ELSE:
                dp[i][j] = MAX(dp[i-1][j], dp[i][j-1])
    RETURN dp[N][M]
```

3) Longest Palindromic Subsequence – O(N²):

```
FUNCTION LongestPalindromicSubseq(s, N):
    reverse_s = REVERSE(s)
    RETURN LCS(s, reverse_s, N, N)
```

4) Fibonacci (DP Approach) – O(N):

```
FUNCTION Fibonacci(N):
    dp = ARRAY of size N+1
    dp[0] = 0, dp[1] = 1
    FOR i FROM 2 TO N:
        dp[i] = dp[i-1] + dp[i-2]
    RETURN dp[N]
```

5) 0/1 Knapsack (Bounded) – O(N * W):

```
FUNCTION Knapsack(weights, values, N, W):
    dp = ARRAY of size (N+1) x (W+1) initialized to 0
    FOR i FROM 1 TO N:
        FOR w FROM 0 TO W:
            IF weights[i-1] <= w:
                dp[i][w] = MAX(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w])
            ELSE:
                dp[i][w] = dp[i-1][w]
    RETURN dp[N][W]
```

6) Coin Change – O(N * Amount):

```
FUNCTION CoinChange(coins, N, Amount):
    dp = ARRAY of size Amount+1 initialized to INF
    dp[0] = 0
    FOR i FROM 1 TO Amount:
        FOR coin IN coins:
            IF i >= coin:
                dp[i] = MIN(dp[i], 1 + dp[i - coin])
    RETURN dp[Amount]
```

7) Matrix Chain Multiplication – O(N³):

```
FUNCTION MatrixChainMultiplication(dimensions, N):
    dp = ARRAY of size N x N initialized to INF
    FOR i FROM 1 TO N:
        dp[i][i] = 0
    FOR L FROM 2 TO N:
        FOR i FROM 1 TO N - L + 1:
            j = i + L - 1
            FOR k FROM i TO j-1:
                cost = dp[i][k] + dp[k+1][j] + (dimensions[i-1] * dimensions[k] *
dimensions[j])
                dp[i][j] = MIN(dp[i][j], cost)
```

```
    RETURN dp[1][N-1]
```

8) DP on Grid (Minimum Path Sum):

```
FUNCTION MinPathSum(grid, N, M):
    dp = ARRAY of size N x M
    dp[0][0] = grid[0][0]
    FOR i FROM 1 TO N:
        dp[i][0] = dp[i-1][0] + grid[i][0]
    FOR j FROM 1 TO M:
        dp[0][j] = dp[0][j-1] + grid[0][j]
    FOR i FROM 1 TO N:
        FOR j FROM 1 TO M:
            dp[i][j] = MIN(dp[i-1][j], dp[i][j-1]) + grid[i][j]
    RETURN dp[N-1][M-1]
```

9) DP on Trees (Diameter of Tree):

```
FUNCTION TreeDiameter(root):
    FUNCTION DFS(node):
        IF node == NULL:
            RETURN 0
        left = DFS(node.left)
        right = DFS(node.right)
        diameter = MAX(diameter, left + right)
        RETURN MAX(left, right) + 1
    diameter = 0
    DFS(root)
    RETURN diameter
```

10) DP on Graphs (Shortest Path – Bellman-Ford):

```
FUNCTION BellmanFord(graph, N, source):
    dist = ARRAY of size N initialized to INF
    dist[source] = 0
    FOR i FROM 1 TO N-1:
        FOR EACH edge (u, v, w) in graph:
            IF dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    RETURN dist
```

11) DP + HashMap (Subset Sum):

```
FUNCTION SubsetSum(arr, sum):
    dp = HASHMAP
    dp[0] = True
    FOR num IN arr:
        FOR s IN REVERSED(dp.keys()):
            dp[s + num] = True
    RETURN dp[sum]
```

12) DP + Bitmasking (Traveling Salesman Problem):

```
FUNCTION TSP(mask, pos):
    IF mask == (1 << N) - 1:
        RETURN cost[pos][0]
    IF dp[mask][pos] != -1:
        RETURN dp[mask][pos]
    ans = INF
```

```
    FOR i FROM 0 TO N:
        IF (mask & (1 << i)) == 0:
            ans = MIN(ans, cost[pos][i] + TSP(mask | (1 << i), i))
    dp[mask][pos] = ans
    RETURN ans
```

13) Digit DP (Count Numbers with Given Sum of Digits):

```
FUNCTION DigitDP(pos, sum, tight):
    IF pos == length:
        RETURN (sum == target)
    IF dp[pos][sum][tight] != -1:
        RETURN dp[pos][sum][tight]
    ans = 0
    limit = digits[pos] IF tight ELSE 9
    FOR digit FROM 0 TO limit:
        ans += DigitDP(pos+1, sum+digit, tight AND (digit == limit))
    dp[pos][sum][tight] = ans
    RETURN ans
```

-------------------------------------------------------------------------------------
--------------------------------------------------------------------------

BINARY SEARCH:

1.Search
2.Maths
3.Rotated Array
4.Tricky Invariant
5.LIS variation
6.Kth Closest/Missing
7.2D matrix
8.Binary Search on Answer

1) Standard Binary Search (Find Target in Sorted Array):

```
FUNCTION BinarySearch(arr, target):
    left = 0, right = LENGTH(arr) - 1
    WHILE left ≤ right:
        mid = left + (right - left) / 2
        IF arr[mid] == target:
            RETURN mid
        ELSE IF arr[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1
    RETURN -1  // Not found
```

2) Binary Search in Mathematical Problems (Square Root):

```
FUNCTION SquareRoot(x):
    left = 1, right = x
    WHILE left ≤ right:
        mid = left + (right - left) / 2
        IF mid * mid == x:
            RETURN mid
        ELSE IF mid * mid < x:
            left = mid + 1
        ELSE:
            right = mid - 1
```

```
    RETURN right  // Largest integer ≤ sqrt(x)
```

3) Search in Rotated Sorted Array:

```
FUNCTION SearchRotatedArray(arr, target):
    left = 0, right = LENGTH(arr) - 1
    WHILE left ≤ right:
        mid = left + (right - left) / 2
        IF arr[mid] == target:
            RETURN mid
        IF arr[left] ≤ arr[mid]:  // Left half is sorted
            IF arr[left] ≤ target < arr[mid]:
                right = mid - 1
            ELSE:
                left = mid + 1
        ELSE:  // Right half is sorted
            IF arr[mid] < target ≤ arr[right]:
                left = mid + 1
            ELSE:
                right = mid - 1
    RETURN -1  // Not found
```

4) Tricky Invariant (Lower Bound - First Occurrence of Target):

```
FUNCTION LowerBound(arr, target):
    left = 0, right = LENGTH(arr)
    WHILE left < right:
        mid = left + (right - left) / 2
        IF arr[mid] >= target:
            right = mid
        ELSE:
            left = mid + 1
    RETURN left  // First occurrence index
```

5) LIS Variation (Using Binary Search - O(N log N)):

```
FUNCTION LIS(arr):
    dp = EMPTY LIST
    FOR num IN arr:
        pos = LOWER_BOUND(dp, num)  // Binary search
        IF pos == LENGTH(dp):
            APPEND num TO dp
        ELSE:
            dp[pos] = num
    RETURN LENGTH(dp)
```

6) Kth Closest/Missing Element:

```
FUNCTION KthMissing(arr, k):
    left = 0, right = LENGTH(arr)
    WHILE left < right:
        mid = left + (right - left) / 2
        missing = arr[mid] - (mid + 1)  // Count missing numbers so far
        IF missing < k:
            left = mid + 1
        ELSE:
            right = mid
    RETURN left + k  // K-th missing number
```

7) Binary Search in 2D Matrix:

```
FUNCTION Search2DMatrix(matrix, target):
    rows = LENGTH(matrix)
    cols = LENGTH(matrix[0])
    left = 0, right = rows * cols - 1
    WHILE left ≤ right:
        mid = left + (right - left) / 2
        mid_value = matrix[mid / cols][mid % cols]
        IF mid_value == target:
            RETURN True
        ELSE IF mid_value < target:
            left = mid + 1
        ELSE:
            right = mid - 1
    RETURN False
```

8) Binary Search on Answer (Find Smallest Divisor):

```
FUNCTION SmallestDivisor(arr, threshold):
    left = 1, right = MAX(arr)
    WHILE left < right:
        mid = left + (right - left) / 2
        sum_div = 0
        FOR num IN arr:
            sum_div += CEIL(num / mid)
        IF sum_div > threshold:
            left = mid + 1
        ELSE:
            right = mid
    RETURN left  // Smallest divisor
```