Tab 1

# Loop Detection in Linked List

## Intuition:

The goal of loop detection in a linked list is to determine whether the linked list contains a cycle, i.e., if any node in the list points back to a previous node, causing an infinite loop. This problem ensures that operations on a linked list don't run indefinitely.

The intuition behind detecting a loop is based on the observation that in a looped linked list, the fast-moving pointer will eventually meet the slow-moving pointer, as they will "lap" each other due to the loop's cyclic nature.

## Key Points:

- **Floyd's Cycle-Finding Algorithm (Tortoise and Hare):**
  - This algorithm uses two pointers:
    - A **slow pointer** (the tortoise) moves one step at a time.
    - A **fast pointer** (the hare) moves two steps at a time.
  - If a loop exists, the fast pointer will eventually meet the slow pointer because it will enter the cycle after a certain number of steps.
  - If the fast pointer reaches the end of the list (i.e., the next pointer is null), then there is no loop.
- **Efficiency:**
  - Time Complexity: **O(n)** because each pointer only traverses the list once.
  - Space Complexity: **O(1)** as only two pointers are used, regardless of the list size.
- **Key Observation:**
  - If there is a cycle, the fast pointer, moving at twice the speed, will eventually "lap" the slow pointer within the cycle.
  - If there is no cycle, the fast pointer will reach the end of the list.

## Applications

- **Memory Management**: Detect and prevent infinite loops caused by circular references in data structures.
- **Network and Graph Theory**: Identify cycles in network paths or dependency graphs.
- **Compiler Design**: Check for circular dependencies in variable declarations.
- **Games and Simulation**: Detect infinite loops in game states or simulations.
- **Linked List Problems**: Prevent infinite traversals in data handling algorithms.

**Code: (Floyd's Cycle)**

```
import java.util.*;
class Node
{
    int val;
    Node next;
    Node(int val)
    {
        this.val = val;
        this.next = null;
    }
}

public class Main
{
    public static boolean hasCycle(Node head)
    {
        if (head == null || head.next == null) return false;
        Node slow = head;
        Node fast = head;
        while (fast != null && fast.next != null)
        {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast)
            {
                return true;
            }
        }
        return false;
    }
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int size = sc.nextInt();
       if (size <= 0)
        {
            System.out.println("0");
        }
        Node head = null, tail = null;
        for (int i = 0; i < size; i++)
        {
            int value = sc.nextInt();
            Node newNode = new Node(value);
            if (head == null)
            {
                head = newNode;
                tail = newNode;
            } else
            {
                tail.next = newNode;
                tail = newNode;
            }
        }
        int pos = sc.nextInt();
        if (pos > 0) {
            Node temp = head;
            Node cycleNode = null;
            int count = 1;
            while (temp != null)
            {
                if (count == pos)
                {
                    cycleNode = temp;
                }
                if (temp.next == null)
                {
                    temp.next = cycleNode;
                    break;
                }
                temp = temp.next;
                count++;
            }
        }
        System.out.println((hasCycle(head) ? "true" : "false"));
    }
}
```
-----------------------------------------------------------------------

# Segregate Even & Nodes in a Linked List

## Intuition:

The task is to rearrange the nodes of a linked list so that all nodes with even values come before the nodes with odd values while maintaining the relative order of nodes within the even and odd groups. The intuition behind the solution is to treat the linked list as two separate lists—one for even nodes and one for odd nodes—and then combine them. By traversing the original list once, we can partition the nodes into two sub-lists (even and odd) and finally merge them to achieve the desired result.

## Key Points: (Partitioning with Two Pointers)

- **Separation into Groups:**
  - Traverse the linked list while maintaining two separate lists:
    - **Even List:** Stores all nodes with even values.
    - **Odd List:** Stores all nodes with odd values.
  - Use pointers (evenTail and oddTail) to track the end of the respective lists for appending nodes efficiently.
- **Re-linking Nodes:**
  - After traversing the entire list, connect the evenTail to the head of the odd list, forming a single linked list where all even nodes precede the odd nodes.
- **Efficiency:**
  - Time Complexity: **O(n)** because the list is traversed only once.
  - Space Complexity: **O(1)** because the reordering is done in place without using extra space for new nodes.
- **Handling Edge Cases:**
  - If there are no even nodes or no odd nodes, the original list is already segregated.
  - Ensure the last node of the odd list points to null to avoid creating a loop.

## Other Possible Approaches:

- **In-Place Rearrangement:**
  - Instead of using separate lists, rearrange the nodes by updating their next pointers during traversal.

  - ➢ While efficient, this approach can be tricky to implement and prone to errors.
- ❖ **Using Extra Memory (Naive Approach):**
  - ➢ Store all even nodes in one list and all odd nodes in another list (e.g., arrays or vectors).
  - ➢ Reconstruct the linked list from these two groups.
  - ➢ **Downside:** Higher space complexity (**O(n)**).

## Applications:

- ❖ **Preprocessing for Algorithms:** Simplify operations or analyses that treat even and odd values differently.
- ❖ **Load Balancing:** In distributed systems, segregating tasks based on even or odd IDs.
- ❖ **Partitioning for Simplicity:** Useful for problems where even and odd nodes need to be processed separately.
- ❖ **Sorting or Filtering Data Streams:** Quick filtering of data based on parity while maintaining order.
- ❖ **Linked List Problems:** This technique serves as a foundational step for more complex linked list transformations.

## Code: (Partitioning with Two Pointers)

```
import java.util.*;
class Node {
   int data;
   Node next;
   Node(int data)
   {
      this.data = data;
      this.next = null;
   }
}

public class Main
{
   public static Node segregateEvenOdd(Node head)
   {
      if (head == null || head.next == null)
         return head;
      Node evenStart = null, evenEnd = null;
```

```
    Node oddStart = null, oddEnd = null;
    Node current = head;
    while (current != null)
    {
        if (current.data % 2 == 0)
        {
            if (evenStart == null)
            {
                evenStart = current;
                evenEnd = evenStart;
            } else
            {
                evenEnd.next = current;
                evenEnd = evenEnd.next;
            }
        }
        else
        {
            if (oddStart == null)
            {
                oddStart = current;
                oddEnd = oddStart;
            }
            else
            {
                oddEnd.next = current;
                oddEnd = oddEnd.next;
            }
        }
        current = current.next;
    }
    if (evenStart == null)
        return oddStart;
    if (oddStart == null)
        return evenStart;
    evenEnd.next = oddStart;
    oddEnd.next = null;
    return evenStart;
}
```

```
    public static void printList(Node head)
    {
        Node current = head;
        while (current != null)
        {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int size = sc.nextInt();
        if (size <= 0)
        {
            System.out.println("0");
        }
        Node head = null, tail = null;
        for (int i = 0; i < size; i++)
        {
            int value = sc.nextInt();
            Node newNode = new Node(value);
            if (head == null)
            {
                head = newNode;
                tail = newNode;
            }
            else
            {
                tail.next = newNode;
                tail = newNode;
            }
        }
        head = segregateEvenOdd(head);
        printList(head);
    }
}
```
-------------------------------------------------------------------

# Sort Bitonic in DLL

## Intuition:

A **bitonic sequence** is a sequence that is first monotonically **increasing** and then monotonically **decreasing**. The idea for sorting such a sequence efficiently is to split the sequence into two parts. The **increasing** part (already sorted in ascending order).The **decreasing** part (sorted in descending order). Once split, the problem reduces to **merging two sorted sequences**: The increasing part stays as it is. The decreasing part is reversed to convert it into ascending order. After reversing, the two sorted parts can be **merged** efficiently using a two-pointer technique (similar to merging two sorted lists).

## Key Points: (Merge-Based Approach)

- **Bitonic Property:**
  - A bitonic sequence has two parts: an increasing part followed by a decreasing part.
  - Identifying these parts helps split the problem into manageable sub-problems.
- **Steps to Solve:**
  - **Split the List:** Traverse the doubly linked list to identify the point where the sequence switches from increasing to decreasing.
  - **Reverse the Decreasing Part:** Reverse the second part of the list (the decreasing segment) to sort it in ascending order.
  - **Merge the Two Sorted Lists:** Use a merge operation (like in merge sort) to combine the two sorted parts into one sorted doubly linked list.
- **Efficiency:**
  - **Time Complexity: O(n)**, where n is the number of nodes in the list. Each operation (splitting, reversing, merging) takes linear time.
  - **Space Complexity: O(1)**, as operations are done in place without requiring additional memory.
- **Why Efficient?**
  - We utilize the properties of the bitonic sequence, avoiding redundant comparisons.
  - The operations (split, reverse, merge) are linear and work efficiently on a doubly linked list.

## Other Possible Approaches:

- **Naive Approach:**

    - ➢ Copy the nodes into an auxiliary array, sort the array using any comparison-based sorting algorithm (like quicksort or mergesort), and rebuild the doubly linked list.
    - ➢ **Drawback:** This approach requires **O(n)** extra space for the array.
- ❖ **Recursive Divide and Conquer:**
    - ➢ Similar to merge sort, recursively identify the bitonic structure, split the list, sort each part, and merge.
    - ➢ This method is effective, but it requires recursive function calls and additional stack space.
- ❖ **Using a Heap (Priority Queue):**
    - ➢ Insert all nodes into a min-heap and reconstruct the sorted list.
    - ➢ **Drawback:** The heap-based approach takes **O(n log n)** time and extra space for the heap.

## Applications:

- ❖ **Sorting Specialized Data:** Bitonic sequences often arise in real-world problems like signal processing and time-series data.
- ❖ **Parallel Computing:** Bitonic sorting is widely used in parallel computing because it can be efficiently implemented using divide-and-conquer techniques.
- ❖ **Data Analysis and Compression:** Bitonic sequences can appear in datasets where values first increase and then decrease (e.g., sales trends, stock market data).
- ❖ **Networking (Bitonic Sort in Sorting Networks):** Bitonic sort is an essential component of sorting networks, which are used in hardware-based sorting algorithms.
- ❖ **Optimization Problems:** Useful in problems where splitting and merging data provides a more optimal solution compared to general-purpose sorting.

## Code:  (Merge-Based Approach)

```
import java.util.*;
class Node
{
   int data;
   Node next, prev;
   Node(int data)
   {
      this.data = data;
      this.next = null;
```

```
        this.prev = null;
    }
}

public class Main
{
    public static Node reverse(Node head)
    {
        Node temp = null;
        Node current = head;
        while (current != null)
        {
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev;
        }
        return temp != null ? temp.prev : head;
    }

    public static Node mergeSortedLists(Node first, Node second)
    {
        if (first == null)
            return second;
        if (second == null)
            return first;
        Node dummy = new Node(0);
        Node current = dummy;
        while (first != null && second != null)
        {
            if (first.data <= second.data)
            {
                current.next = first;
                first.prev = current;
                first = first.next;
            }
            else
            {
                current.next = second;
                second.prev = current;
                second = second.next;
            }
            current = current.next;
        }
        if (first != null)
        {
            current.next = first;
            first.prev = current;
        }
        else if (second != null)
        {
            current.next = second;
            second.prev = current;
        }
        return dummy.next;
    }

    public static Node sortBitonicDoublyLinkedList(Node head)
    {
        if (head == null || head.next == null)
            return head;
        Node increasing = head;
        Node decreasing = head.next;
        while (decreasing != null && decreasing.next != null && decreasing.data <=
decreasing.next.data)
        {
            decreasing = decreasing.next;
        }
        if (decreasing != null)
        {
            decreasing.prev.next = null;
            decreasing.prev = null;
        }
        decreasing = reverse(decreasing);
        return mergeSortedLists(increasing, decreasing);
    }
    public static void printList(Node head)
    {
        Node temp = head;
        while (temp != null)
        {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Node head = null, tail = null;
        for (int i = 0; i < n; i++)
        {
            int value = sc.nextInt();
            Node newNode = new Node(value);
            if (head == null)
            {
                head = newNode;
                tail = newNode;
            }
            else
            {
                tail.next = newNode;
                newNode.prev = tail;
                tail = newNode;
            }
        }
        head = sortBitonicDoublyLinkedList(head);
        printList(head);
    }
}
```

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

# Merge Sort in DLL

## Intuition:

The **Merge Sort** algorithm divides the linked list into two halves, recursively sorts each half, and then merges the two sorted halves to produce a fully sorted list. For a doubly linked list (DLL), this approach leverages the bidirectional nature of the list to efficiently split and merge nodes without requiring extra memory for arrays or additional data structures.

## Key Points:

- **Splitting the List:**
  - Find the middle of the DLL to divide it into two halves.
  - Use the **slow and fast pointer technique** to locate the midpoint efficiently in **O(n)** time.
  - Break the list into two sublists by updating the prev and next pointers at the midpoint.
- **Recursively Sort Both Halves:**
  - Apply Merge Sort to the two halves independently.
  - Each recursion divides the list further until each sublist contains a single node (base case).
- **Merging Two Sorted Lists:**
  - Use the **merge step of Merge Sort** to combine the two sorted halves into one sorted list.
  - Compare the values of nodes from each half, append the smaller node to the result, and move pointers accordingly.
  - Ensure proper updates to the prev and next pointers during the merge to maintain the doubly linked structure.
- **Efficiency:**
  - **Time Complexity: O(n log n)** for the entire algorithm.
  - **Space Complexity: O(log n)** for the recursion stack.

## Other Possible Approaches:

- **QuickSort for DLL:**
  - Use the last node as a pivot and partition the list around it.
  - Recursively sort the partitions.

- While QuickSort has an average time complexity of **O(n log n)**, it can degrade to **O(n²)** in the worst case (e.g., when the list is already sorted).

- **Insertion Sort for DLL:**
  - Traverse the list and insert each node into its correct position in a sorted portion of the list.
  - Time complexity is **O(n²)**, making it less efficient for large lists.

## Applications:

- **Sorting Large Linked Data Structures:** Efficiently sort doubly linked lists in scenarios where in-place sorting is required.
- **Database Management Systems:** Used for indexing and sorting rows in doubly linked structures.
- **Optimal Sorting for Sparse Memory:** No additional memory allocation makes it suitable for low memory environments.
- **Backend Operations:** Efficiently sort logs, tasks, or records stored in doubly linked lists.
- **Foundational Algorithm:** Merge Sort for DLLs is a foundational step for understanding advanced linked list manipulations and sorting algorithms.

## CODE: (Merge Sort)

```
import java.util.*;
class Node {
   int data;
   Node next, prev;
   Node(int data) {
      this.data = data;
      this.next = null;
      this.prev = null;
   }
}

public class Main
{
   private static Node split(Node head)
   {
      Node slow = head, fast = head;
      while (fast.next != null && fast.next.next != null)
      {
          slow = slow.next;
          fast = fast.next.next;
      }
      Node secondHalf = slow.next;
      slow.next = null;
      if (secondHalf != null)
      {
          secondHalf.prev = null;
      }
      return secondHalf;
   }

   private static Node merge(Node first, Node second)
   {
      if (first == null)
           return second;
      if (second == null) return first;
      Node dummy = new Node(0);
      Node current = dummy;
      while (first != null && second != null)
      {
          if (first.data <= second.data)
          {
              current.next = first;
              first.prev = current;
              first = first.next;
          }
          else
          {
              current.next = second;
              second.prev = current;
              second = second.next;
          }
          current = current.next;
      }
      if (first != null)
      {
          current.next = first;
          first.prev = current;
      }
```

```
        else if (second != null)
        {
            current.next = second;
            second.prev = current;
        }
        return dummy.next;
    }

    public static Node mergeSort(Node head)
    {
        if (head == null || head.next == null)
            return head;
        Node secondHalf = split(head);
        head = mergeSort(head);
        secondHalf = mergeSort(secondHalf);
        return merge(head, secondHalf);
    }

    private static void printList(Node head)
    {
        Node temp = head;
        while (temp != null)
        {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

public static void main(String[] args)
{
        Scanner sc = new Scanner(System.in);
        int size = sc.nextInt();
        if (size <= 0)
        {
            System.out.println("0");
        }
        Node head = null, tail = null;
        for (int i = 0; i < size; i++)
        {
            int value = sc.nextInt();
            Node newNode = new Node(value);
            if (head == null)
            {
                head = newNode;
                tail = newNode;
            }
            else
            {
                tail.next = newNode;
                newNode.prev = tail;
                tail = newNode;
            }
        }
        head = mergeSort(head);
        printList(head);
    }
}
```

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

# Minimum Stack

## Intuition:

The goal of a **Minimum Stack (Min Stack)** is to design a stack that, in addition to the standard stack operations (push, pop, top), can retrieve the minimum element in constant time **O(1)**.

The intuition behind the efficient implementation is to use an **auxiliary stack** to track the minimum values. This auxiliary stack stores the minimum value observed so far whenever a new value is pushed onto the main stack. Thus, at any point, the top of the auxiliary stack represents the current minimum value.

## Key Points: (Stack with Auxiliary Stack)

- ❖ **Two Stacks:**
  - ➢ Use two stacks:
    - ■ **Main Stack:** Stores all the elements as in a regular stack.
    - ■ **Auxiliary Stack:** Tracks the minimum value at each stage of the main stack.
- ❖ **Push Operation:**
  - ➢ Push the value onto the main stack.
  - ➢ For the auxiliary stack:
    - ■ If the auxiliary stack is empty, push the same value.
    - ■ Otherwise, push the smaller of the current value and the top of the auxiliary stack (to maintain the minimum).
- ❖ **Pop Operation:**
  - ➢ Pop the top element from both the main stack and the auxiliary stack to ensure consistency.
- ❖ **Get Minimum:**
  - ➢ Return the top of the auxiliary stack, which represents the minimum value in the main stack.
- ❖ **Efficiency:**
  - ➢ Time Complexity: O(1)
  - ➢ Space Complexity: O(n), as the auxiliary stack stores an extra value for each element in the main stack.

## Other Possible Approaches:

- ❖ **Using a Single Stack with Pairs:**
  - ➢ Store a pair (value, current_min) in the stack instead of maintaining a separate auxiliary stack.
  - ➢ The first element in the pair represents the stack value, and the second element keeps track of the minimum value at that point.
  - ➢ **Downside:** Slightly more memory overhead due to pairs.
- ❖ **Using a Single Stack with Adjusted Values:**
  - ➢ Store adjusted values in the stack to track the minimum implicitly.
  - ➢ Example: Instead of directly storing the value, store the difference between the value and the current minimum.
  - ➢ Update the minimum dynamically during push and pop.
  - ➢ **Complexity:** O(1) for all operations but harder to implement and debug.

## Applications:

- ❖ **Efficient Data Retrieval:** Retrieve minimum values quickly in stack-based algorithms.
- ❖ **Histogram Problems:** Useful for solving problems involving rectangles in histograms or areas bounded by stacks.
- ❖ **Dynamic Minimum Tracking:** Real-time systems that require constant-time retrieval of minimum values (e.g., price monitoring).
- ❖ **Algorithm Optimization:** Used in dynamic programming or sliding window problems to optimize minimum value calculations.
- ❖ **Game Development:** Track minimum values during gameplay logic involving stack-like undo/redo features.

## CODE: (Stack with Auxiliary Stack)

```
import java.util.*;
class MinStack
{
   Stack<Integer> stack = new Stack<>();
   Stack<Integer> minStack = new Stack<>();

   public void push(int x)
   {
      stack.push(x);
      if (minStack.isEmpty() || x <= minStack.peek())
      {
         minStack.push(x);
```

```
        }
    }

    public void pop()
    {
      if (!stack.isEmpty())
        {
          if (stack.peek().equals(minStack.peek()))
          {
            minStack.pop();
          }
           stack.pop();
        }
    }

    public int top()
    {
        if (stack.isEmpty())
            return -1;
        return stack.peek();
    }

    public int getMin()
    {
        if (minStack.isEmpty())
            return -1;
        return minStack.peek();
    }
}

public class Main
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        MinStack minStack = new MinStack();
        int operations = sc.nextInt();
        for (int i = 0; i < operations; i++)
        {
            String operation = sc.next();
            switch (operation)
            {
                case "push":
                    int value = sc.nextInt();
                    minStack.push(value);
                    break;

                case "pop":
                    minStack.pop();
                    break;

                case "top":
                    System.out.println(minStack.top());
                    break;

                case "getMin":
                    System.out.println(minStack.getMin());
                    break;

                default:
                    System.out.println("Invalid operation");
                    break;
            }
        }
    }
}
```

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

# Celebrity Problem

## Intuition:

The core idea is to use a **stack** to eliminate non-celebrities efficiently. By leveraging the properties of a celebrity:

- ❖ A celebrity does not know anyone.
- ❖ Everyone knows the celebrity.

We can systematically eliminate potential candidates by comparing pairs of people and only keeping those who may satisfy the celebrity criteria.

## Key Points: (Using Stack)

- ❖ **Push All People onto the Stack:**
  - ➢ Push all n individuals (0 to n−1) onto the stack.
- ❖ **Eliminate Non-Celebrities:**
  - ➢ Pop two people, A and B, from the stack.
  - ➢ If A knows B, A cannot be a celebrity (eliminate A).
  - ➢ If A does not know B, B cannot be a celebrity (eliminate B).
  - ➢ Push the potential candidate back onto the stack.
- ❖ **Check Final Candidate:**
  - ➢ After eliminating n−1 people, one person remains on the stack.
  - ➢ Verify whether this person is a celebrity by checking:
    - ■ They do not know anyone.
    - ■ Everyone knows them.
- ❖ **Efficiency:**
  - ➢ **Time Complexity: O(n)**.
  - ➢ **Space Complexity: O(n)** for the stack.

## Other Possible Approaches:

- ❖ **Naive Approach (Brute Force):**
  - ➢ Check each person individually to see if they satisfy the celebrity criteria.
  - ➢ Time Complexity: O(n^2), as we check n−1 relationships for each person.
  - ➢ Space Complexity: O(1).
- ❖ **Matrix-Based Approach:**
  - ➢ Use a matrix representation of the "knows" relationship.

    - ➢ Process rows and columns to eliminate non-celebrities based on properties of a celebrity.
    - ➢ Time Complexity: O(n^2).
- ❖ **Two-Pointer Approach:**
    - ➢ Use two pointers to iterate over the array and eliminate non-celebrities in a single pass.
    - ➢ Similar elimination logic as the stack approach but uses constant space (**O(1)**).

## Applications:

- ❖ **Social Network Analysis:** Identify influencers or key nodes in a network.
- ❖ **Game Theory:** Solve problems where hierarchy or isolation criteria are involved.
- ❖ **Database Systems:** Resolve relationships in datasets with directed graphs.
- ❖ **Event Planning:** Identify VIPs at an event based on who is recognized by others.
- ❖ **Graph Theory Problems:** Applicable in problems involving dominance relationships in directed graphs.

## CODE: (Using Stack)

```
import java.util.*;
public class Main
{
  public static int findCelebrity(int[][] matrix, int n)
  {
    Stack<Integer> stack = new Stack<>();
    for (int i = 0; i < n; i++)
    {
      stack.push(i);
    }
    while (stack.size() > 1)
    {
      int a = stack.pop();
      int b = stack.pop();
      if (matrix[a][b] == 1)
      {
        stack.push(b);
      } else
      {
        stack.push(a);
```

```
            }
        }
        int candidate = stack.pop();
        for (int i = 0; i < n; i++)
        {
            if (i != candidate && (matrix[candidate][i] == 1 || matrix[i][candidate] == 0))
            {
                return -1;
            }
        }
        return candidate;
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                matrix[i][j] = sc.nextInt();
            }
        }
        int celebrity = findCelebrity(matrix, n);
        if (celebrity == -1)
        {
            System.out.println("No celebrity found.");
        } else
        {
            System.out.println(celebrity);
        }
    }
}
```

## CODE: (Two Pointer Approach)

```
import java.util.*;
public class Main {
    public static int findCelebrity(int[][] matrix, int n)
    {
        int candidate = 0;
        for (int i = 1; i < n; i++)
        {
            if (matrix[candidate][i] == 1)
            {
                candidate = i;
            }
        }
        for (int i = 0; i < n; i++) {
            if (i != candidate && (matrix[candidate][i] == 1 || matrix[i][candidate] == 0))
            {
                return -1;
            }
        }
        return candidate;
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                matrix[i][j] = sc.nextInt();
            }
        }
        int celebrity = findCelebrity(matrix, n);
        if (celebrity == -1) {
            System.out.println("No celebrity found.");
        } else {
            System.out.println(celebrity);
        }
    }
}
```
---------------------------------------------------------------------

# Tower of Hanoi

## Intuition:

The **Tower of Hanoi** is a classic problem where the goal is to move n disks from a source rod to a target rod, following these rules:

- ❖ Only one disk can be moved at a time.
- ❖ A larger disk cannot be placed on top of a smaller disk.
- ❖ There is a third rod (auxiliary) that can be used as a temporary holding area.

The key intuition is to **divide the problem into smaller subproblems**:

- ❖ To move n disks, first move the top n-1 disks to the auxiliary rod.
- ❖ Move the largest disk (nth disk) directly to the target rod.
- ❖ Finally, move the n-1 disks from the auxiliary rod to the target rod.

This recursive strategy systematically reduces the problem size until it reaches the base case of moving a single disk.

## Key Points:

- ❖ **Recursive Breakdown:**
    - ➢ Break the problem of moving n disks into smaller problems of size n-1.
    - ➢ Use the following steps recursively:
        - ■ Move n-1 disks from the source to the auxiliary rod using the target rod.
        - ■ Move the largest disk from the source to the target rod.
        - ■ Move n-1 disks from the auxiliary rod to the target rod using the source rod.
- ❖ **Base Case:**
    - ➢ When n = 1, simply move the disk from the source rod to the target rod.
- ❖ **Time Complexity: O(2^n)**.
- ❖ **Space Complexity: O(n)** due to the recursive function call stack space.

## Other Possible Approaches:

- ❖ **Iterative Solution:**
    - ➢ Use an iterative method with a stack to simulate the recursive process.

➢ This approach is less intuitive but avoids the recursion stack overhead.

## Applications:

❖ **Algorithmic Thinking:** Teaches recursive problem-solving and divide-and-conquer techniques.
❖ **Data Structure Manipulation:** Useful in scenarios requiring stepwise movement or transfer of data while maintaining order.
❖ **Real-Life Scheduling:** Models tasks that require intermediate steps, such as assembly lines or multitiered workflows.
❖ **Disk Balancing in Distributed Systems:** Simulates transferring data between servers while ensuring order is maintained.
❖ **Educational Tools:** Demonstrates recursion, exponential growth, and problem decomposition in computer science courses.

## CODE:

```
import java.util.*;
public class Main
{
   public static void solveHanoi(int n, char source, char target, char auxiliary)
   {
      if (n == 1)
      {
         System.out.println("Move disk 1 from " + source + " to " + target);
         return;
      }
      solveHanoi(n - 1, source, auxiliary, target);
      System.out.println("Move disk " + n + " from " + source + " to " + target);
      solveHanoi(n - 1, auxiliary, target, source);
   }

   public static void main(String[] args)
   {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      solveHanoi(n, 'A', 'C', 'B');
   }
}
```

---

# Stock Span

## Intuition:

The **Stock Span Problem** involves finding the span of stock prices for each day. The span for a given day is defined as the maximum number of consecutive days (including the current day) the stock price has been less than or equal to the stock price of the current day.

The stack-based solution relies on maintaining a **monotonic decreasing stack** of indices. For each day's stock price, we check previous days' prices efficiently by "jumping" over days that have lower prices.

## Key Points: (Stack Based Approach)

- **Using a Stack for Indices:**
  - Maintain a stack that stores indices of stock prices in decreasing order.
  - The stack ensures that for each price, only relevant indices are checked (those with higher or equal prices are already processed and removed).
- **Span Calculation Logic:**
  - For each day's stock price at index i:
    - Pop indices from the stack while the stock price at the top of the stack is less than or equal to the current price.
    - If the stack becomes empty, the span is i + 1 (all previous prices are less than or equal).
    - If the stack is not empty, the span is i - stack.top() (distance to the last higher price).
- **Push Current Index:**
  - After calculating the span for day i, push the current index onto the stack to maintain the monotonic property.
- **Efficiency:**
  - **Time Complexity: O(n)**.
  - **Space Complexity: O(n)** for the stack.

## Other Possible Approaches:

- ❖ **Brute Force:**
  - ➢ For each day i, iterate backward to count the span by comparing prices.
  - ➢ Time Complexity: **O($n^2$)**.
  - ➢ Not efficient for large datasets.
- ❖ **Dynamic Programming (Precompute Nearest Greater):**
  - ➢ Precompute the nearest greater element to the left for each price and use it to calculate the span.
  - ➢ Time Complexity: **O(n)**, but requires extra preprocessing.

## Applications:

- ❖ **Stock Market Analysis:** Calculate stock spans to understand trends in stock prices.
- ❖ **Financial Data Analysis:** Useful in scenarios where consecutive patterns of higher prices need to be identified.
- ❖ **Histogram Problems:** Variations of this technique are applied in solving histogram-based problems like finding the largest rectangle.
- ❖ **Weather Trends:** Analyze temperature or weather data for consecutive days with similar or lower temperatures.
- ❖ **Competitive Programming:** Frequently used in coding competitions to demonstrate efficient stack-based solutions.

## CODE:( Stack Based Approach)

```
import java.util.*;
public class Main
{
   public static int[] calculateSpan(int[] prices, int n)
   {
      int[] span = new int[n];
      Stack<Integer> stack = new Stack<>();
      for (int i = 0; i < n; i++)
      {
         while (!stack.isEmpty() && prices[stack.peek()] <= prices[i])
         {
            stack.pop();
         }
         span[i] = (stack.isEmpty()) ? (i + 1) : (i - stack.peek());
         stack.push(i);
      }
```

```
        return span;
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] prices = new int[n];
        for (int i = 0; i < n; i++)
        {
            prices[i] = sc.nextInt();
        }
        int[] spans = calculateSpan(prices, n);
        for (int span : spans)
        {
            System.out.print(span + " ");
        }
        sc.close();
    }
}
```

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

# Priority Queue Using DLL

## Intuition:

A **priority queue** is a data structure that processes elements based on their priority rather than their insertion order. The goal of implementing a priority queue using a doubly linked list (DLL) is to leverage the DLL's bidirectional traversal to efficiently insert and retrieve elements while maintaining their priority order.

The main idea is to:

1. Maintain a **sorted doubly linked list** where the elements are arranged in descending or ascending order of priority.
2. Insertion is performed at the correct position based on the priority, ensuring the list remains sorted.
3. Removal (either the highest or lowest priority element) is performed in constant time **O(1)**, as the list's head or tail can be accessed directly.

## Key Points:

- ❖ **Insertion:**
  - ➢ Traverse the list to find the correct position for the new element based on its priority.
  - ➢ Insert the new node at this position by adjusting the prev and next pointers.
  - ➢ Time Complexity: **O(n)** (in the worst case, the new element might need to be inserted at the end).
- ❖ **Deletion:**
  - ➢ Remove either the highest-priority element (head of the list) or the lowest-priority element (tail of the list).
  - ➢ Adjust the pointers of adjacent nodes accordingly.
  - ➢ Time Complexity: **O(1)**.
- ❖ **Peek (Retrieve Priority Element):**
  - ➢ The highest or lowest priority element is at the head or tail, so retrieval is constant time **O(1)**.
- ❖ **Traversal:**
  - ➢ Since the list is doubly linked, forward and backward traversal is efficient.
  - ➢ Useful for iterating through all elements or debugging.
- ❖ **Efficiency:**
  - ➢ Insert: **O(n)** (due to traversal to find the correct position).

    - Delete: **O(1)**.
    - Peek: **O(1)**.

## Other Possible Approaches:

- **Priority Queue Using Arrays:**
    - Maintain a sorted array for priorities.
    - **Insert:** O(n) (due to shifting elements to maintain order).
    - **Delete/Peek:** O(1).
- **Priority Queue Using Heaps (Optimal):**
    - Use a binary heap (min-heap or max-heap) to implement a priority queue.
    - **Insert/Remove:** O(log n).
    - **Peek:** O(1).
    - This is the most efficient and widely used implementation.
- **Priority Queue Using Single Linked List:**
    - Similar to DLL but lacks backward traversal.
    - Insertion and traversal are possible but less efficient compared to DLL.

## Applications:

- **Scheduling Tasks:** Use in operating systems to schedule tasks based on priority.
- **Dijkstra's Algorithm:** Manage the processing of nodes in shortest path algorithms.
- **Data Stream Processing:** Handle dynamic elements with priorities (e.g., streaming logs, alerts).
- **Event Handling:** Simulate or manage real-time events based on priority.
- **Job Scheduling:** Assign tasks to resources in priority order.
- **Real-Time Systems:** Ensure high-priority tasks are processed before lower-priority tasks.

## CODE:

```
import java.util.Scanner;
class Node
{
   int data;
   int priority;
   Node next, prev;
   Node(int data, int priority)
   {
      this.data = data;
      this.priority = priority;
```

```
        this.next = null;
        this.prev = null;
    }
}

public class Main
{
    private Node head;
    public void insert(int data, int priority)
    {
        Node newNode = new Node(data, priority);
        if (head == null)
        {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp != null && temp.priority >= priority)
        {
            temp = temp.next;
        }
        if (temp == null)
        {
            Node tail = head;
            while (tail.next != null)
            {
                tail = tail.next;
            }
            tail.next = newNode;
            newNode.prev = tail;
        }
        else if (temp == head)
        {
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }
        else
        {
            newNode.prev = temp.prev;
            newNode.next = temp;
            temp.prev.next = newNode;
            temp.prev = newNode;
        }
    }

    public int remove()
    {
        if (head == null)
        {
            return -1;
        }
        int value = head.data;
        head = head.next;
        if (head != null)
        {
            head.prev = null;
        }
        return value;
    }
    public void display()
    {
        Node temp = head;
        while (temp != null)
        {
            System.out.print("(" + temp.data + ", " + temp.priority + ") ");
            temp = temp.next;
        }
        System.out.println();
    }
    public static void main(String[] args)
    {
        Main pq = new Main();
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for (int i = 0; i < n; i++)
        {
            int data = sc.nextInt();
            int priority = sc.nextInt();
            pq.insert(data, priority);
        }
        int removeCount = sc.nextInt();
        for (int i = 0; i < removeCount; i++)
        {
            int removed = pq.remove();
            if (removed == -1)
            {
                System.out.println("Priority Queue is empty.");
            }
        }
        pq.display();

    }
}
```

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

# Sort Without Extra Space

## Intuition:

The problem involves sorting elements while using a queue (or queues) as the primary data structure and avoiding extra space for storage or auxiliary arrays. Queues inherently operate in a **FIFO (First-In-First-Out)** manner, so sorting requires careful manipulation of their structure. The idea is to repeatedly **extract the smallest (or largest)** element, reposition it at the end, and ensure that all elements are sorted within the queue itself.

## Key Points of the Approach:

- ❖ **Selection Sort Using Queue Properties:**
  - ➢ Use the concept of **selection sort** by identifying the smallest (or largest) element in the queue during each iteration.
  - ➢ Once identified, rotate the queue to bring the selected element to the desired position.
- ❖ **Queue Rotation for Reordering:**
  - ➢ After extracting the smallest element, rotate the remaining elements back to their original order to maintain queue consistency.
  - ➢ This ensures no additional space is used for temporary storage.
- ❖ **Iterative Sorting:**
  - ➢ Perform the above process iteratively for the first n elements, then n-1, and so on, effectively sorting the queue in place.
- ❖ **Efficiency:**
  - ➢ **Time Complexity:**
    - ■ For n elements, this results in a total complexity of **O($n^2$)**.
  - ➢ **Space Complexity:**
    - ■ No extra space beyond the queue itself, so **O(1)** auxiliary space.

## Other Possible Approaches:

- ❖ **Bubble Sort Using Queue:**
  - ➢ Repeatedly compare adjacent elements in the queue, swapping their positions if necessary by rotating the queue.
  - ➢ Requires **O($n^2$)** time and **O(1)** space.
- ❖ **Merge Sort Using Multiple Queues:**
  - ➢ Divide the queue into smaller sub-queues recursively, sort them, and merge them back.

        - Requires multiple queues, so it's not strictly in-place.
- **Heap Sort Using Queue:**
    - Use the queue as a heap-like structure to implement heap sort.
    - May require auxiliary space for restructuring elements.

## Applications:

- **Memory-Constrained Systems:** Sort data in environments where additional memory allocation is restricted.
- **Queue-Based Data Processing:** Situations where input and output must remain in queue form, such as in network or task scheduling systems.
- **Streaming Data:** Process and sort elements in queues for streaming applications.
- **Embedded Systems:** Use in low-memory devices where efficient in-place sorting is necessary.
- **Event Management:** Prioritize events in real-time systems without using additional memory.

## CODE:

```
import java.util.*;
public class Main
{
   private static int findMinIndex(Queue<Integer> queue, int sortedIndex)
   {
      int minIndex = -1;
      int minValue = Integer.MAX_VALUE;
      int size = queue.size();
      for (int i = 0; i < size; i++)
      {
         int current = queue.poll();
         if (i <= sortedIndex && current < minValue)
         {
            minValue = current;
            minIndex = i;
         }
         queue.add(current);
      }
      return minIndex;
   }

   private static void moveToRear(Queue<Integer> queue, int minIndex)
```

```
    {
        int size = queue.size();
        int minValue = 0;
        for (int i = 0; i < size; i++)
        {
            int current = queue.poll();
            if (i == minIndex)
            {
                minValue = current;
            }
            else
            {
                queue.add(current);
            }
        }
        queue.add(minValue);
    }
    public static void sortQueue(Queue<Integer> queue)
    {
        int n = queue.size();
        for (int i = 0; i < n; i++)
        {
            int minIndex = findMinIndex(queue, n - 1 - i);
            moveToRear(queue, minIndex);
        }
    }
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        Queue<Integer> queue = new LinkedList<>();
        int n = sc.nextInt();
        for (int i = 0; i < n; i++)
        {
            queue.add(sc.nextInt());
        }
        sortQueue(queue);
        System.out.println(queue);
    }
}
```
---------------------------------------------------------------------

# Stack Permutations

## Intuition:

The **Stack Permutation Problem** asks if a given sequence can be the result of a stack operation starting from an increasing sequence of numbers (typically from 1 to n). The goal is to determine if it is possible to push elements onto a stack and pop them to obtain a particular permutation.

To solve this:

- We simulate the process of pushing and popping elements into/from the stack.
- We start by pushing numbers from 1 to n onto the stack.
- For each number in the desired permutation, check if it is at the top of the stack. If it is, pop it. Otherwise, continue pushing numbers onto the stack until the top of the stack matches the current element in the desired permutation.

The intuition is that a stack follows a **Last-In-First-Out (LIFO)** order, so elements can only be removed in the reverse order in which they were added.

## Key Points:

- **Push and Pop Operations:**
  - We simulate pushing elements onto the stack from 1 to n.
  - If the element at the top of the stack matches the current element in the target permutation, we pop it.
  - Continue this process until all elements are popped in the same order as the target permutation.
- **Order Constraints:**
  - If at any point, the element at the top of the stack doesn't match the current element in the permutation, and we have already pushed all available elements, then it is not possible to achieve that permutation.
- **Efficiency:**
  - **Time Complexity:**
    - The process involves pushing and popping each element at most once, so the time complexity is **O(n)**, where n is the number of elements.
  - **Space Complexity:**
    - The space complexity is **O(n)** due to the stack.
- **Correctness:**

- The stack can generate the desired permutation if and only if the stack simulation correctly mirrors the input and output sequences based on the constraints of stack operations.

## Other Possible Approaches:

- **Brute Force:**
  - Generate all permutations of the sequence and check if the stack operations can match each permutation.
  - **Time Complexity:** This approach would be much slower, with a time complexity of **O(n!)** (for generating permutations).
- **Recursive Backtracking:**
  - This approach can also explore the push and pop operations recursively, backtracking when an invalid operation is encountered.
  - **Time Complexity: O(n!)**, as it would involve generating and checking all possible sequences.
- **Queue Simulation:**
  - While a stack is the natural choice, a queue-based approach could simulate the process with a different set of constraints, though it is not typically as efficient or intuitive for this problem.

## Applications:

- **Expression Evaluation:** In evaluating expressions (e.g., infix to postfix conversion), understanding stack permutations can help validate if a given sequence of operations is achievable.
- **Compiler Design:** Used in scenarios where operations involve processing sequences based on last-in-first-out constraints.
- **Real-Time Scheduling:** Stack-based operations can be applied in scheduling tasks where the order of execution follows strict last-in-first-out rules.
- **Reversible Operations:** Stack permutations can be used in problems that involve reversible operations or backtracking.
- **Data Structure Simulation:** Helps in simulating how certain data structures (like stacks) behave under different operational constraints.

## CODE:

```
import java.util.*;
public class Main
```

```
{
   public static boolean isStackPermutation(int[] input, int[] output)
   {
      int n = input.length;
      Stack<Integer> stack = new Stack<>();
      int j = 0;
      for (int i = 0; i < n; i++)
      {
         stack.push(input[i]);
         while (!stack.isEmpty() && stack.peek() == output[j])
         {
            stack.pop();
            j++;
         }
      }
      return stack.isEmpty();
   }

   public static void main(String[] args)
   {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      if (n <= 0)
      {
         System.out.println("0");
         return;
      }
      int[] input = new int[n];
      for (int i = 0; i < n; i++)
      {
         input[i] = sc.nextInt();
      }
      int[] output = new int[n];
      for (int i = 0; i < n; i++)
      {
         output[i] = sc.nextInt();
      }
      boolean isValid = isStackPermutation(input, output);
      if (isValid)
      {
            System.out.println("Valid");
        }
        else
        {
            System.out.println("Not Valid.");
        }

    }
}
```

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

# Max Sliding Window

## Intuition:

The **Max Sliding Window Problem** involves finding the maximum value in every subarray (or "window") of size k as the window slides across an array. The **monotonic deque** approach leverages the following insights:

- Use a deque (double-ended queue) to store indices of elements, maintaining a decreasing order of values in the deque.
- The maximum element of the current window is always at the front of the deque.
- As the window slides, remove indices that are out of the window's range and those that are no longer useful for finding the maximum (smaller values).

## Key Points: (Deque Approach)

- **Deque Properties:**
  - Stores indices of array elements.
  - Maintains elements in decreasing order of values from the front to the back.
  - Allows efficient addition and removal of elements from both ends.
- **Sliding Window Logic:**
  - Add the current element's index to the deque.
  - Remove indices from the back of the deque if their corresponding values are smaller than the current element (as they will never be the maximum).
  - Remove indices from the front of the deque if they are out of the current window's range.
- **Extract Maximum:**
  - The maximum value of the current window is the element at the front of the deque.
- **Efficiency:**
  - **Time Complexity:**
    - Each element is added and removed from the deque at most once, resulting in **O(n)**.
  - **Space Complexity:**
    - The deque stores at most k indices, so **O(k)** space is used.

## Other Possible Approaches:

- **Brute Force:**

- Iterate through each window of size k and find the maximum value.
- **Time Complexity: O(n × k)** (inefficient for large n or k).

- **Heap-Based Approach:**
  - Use a max-heap to store elements of the current window.
  - Remove elements outside the window and peek at the maximum.
  - **Time Complexity: O(n log k)** (efficient but slower than the deque approach).

## Applications:

- **Stock Market Analysis:**
  - Track the maximum stock price in a rolling time window.
- **Sensor Data Monitoring:**
  - Identify peaks in real-time sensor readings for a sliding time frame.
- **Image Processing:**
  - Used in max-pooling layers for convolutional neural networks (CNNs).
- **Financial Modeling:**
  - Find maximum values in rolling datasets, such as revenue or traffic data.
- **Weather Data Analysis:**
  - Monitor maximum temperatures or precipitation over sliding time windows.
- **Dynamic Problem Constraints:**
  - Optimize solutions for problems involving dynamic subarray constraints.

## CODE: (Deque Approach)

```
import java.util.*;
public class Main
{
   public static int[] maxSlidingWindow(int[] nums, int k)
   {
      if (nums == null || nums.length == 0 || k <= 0)
      {
         return new int[0];
      }
      int n = nums.length;
      int[] result = new int[n - k + 1];
      Deque<Integer> deque = new LinkedList<>();
      for (int i = 0; i < n; i++)
      {
         if (!deque.isEmpty() && deque.peekFirst() < i - k + 1)
         {
             deque.pollFirst();
         }
         while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i])
         {
             deque.pollLast();
         }
         deque.offerLast(i);
         if (i >= k - 1)
         {
             result[i - k + 1] = nums[deque.peekFirst()];
         }
      }
      return result;
   }

   public static void main(String[] args)
   {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      if (n <= 0)
      {
         System.out.println(0);
         return;
      }
      int[] nums = new int[n];
      for (int i = 0; i < n; i++)
      {
         nums[i] = sc.nextInt();
      }
      int k = sc.nextInt();
      if (k <= 0 || k > n)
      {
         System.out.println(n);
         return;
      }
      int[] result = maxSlidingWindow(nums, k);
      for (int num : result) {
         System.out.print(num + " ");
      }

      sc.close();
   }
}
```

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

# Recover the BST

## Intuition:

In a **Binary Search Tree (BST)**, for any node, the value of the left child should be smaller, and the value of the right child should be larger. If two nodes are swapped in the tree, the BST property is violated. The goal is to **recover the tree** by identifying and swapping the two nodes that were mistakenly swapped.

The main idea for recovery is:

- **Inorder Traversal:**
  - In a valid BST, an **inorder traversal** (left subtree → root → right subtree) of the tree will return the nodes in sorted order.
  - If two nodes are swapped, their values will be out of order during an inorder traversal.
- **Identify the Nodes to Swap:**
  - During the inorder traversal, keep track of the previous node and compare it to the current node.
  - The first violation where previous > current indicates a node that needs to be corrected.
  - The second violation (the next occurrence where previous > current) points to the second node to be swapped.
- **Efficient Recovery:**
  - After identifying the two misplaced nodes, simply swap their values to recover the tree.

## Key Points:

- **Inorder Traversal:**
  - Traverse the tree in inorder to identify the out-of-order nodes. This is done once, ensuring **O(n)** time complexity.
  - The traversal uses **O(h)** space where h is the height of the tree, if a recursive approach is used. Alternatively, an iterative approach can use **O(1)** extra space.
- **Two Nodes to Swap:**
  - Identify two nodes that are swapped by comparing adjacent nodes during the inorder traversal.
  - There are two cases of violation:

        - The first violation occurs when the previous node’s value is greater than the current node’s value.
        - The second violation happens when the next node’s value is greater than the current node’s value.

- **Swap the Nodes:**
    - After identifying the two misplaced nodes, swap their values to restore the BST property.
- **Time Complexity:**
    - The time complexity is **O(n)** because we perform an inorder traversal of the tree to detect the swapped nodes.
- **Space Complexity:**
    - **O(h)**, where h is the height of the tree, for the recursive call stack. Alternatively, **O(1)** space if an iterative inorder traversal is used.

## Other Possible Approaches:

- **Recursive Inorder Traversal with Extra Space:**
    - Use extra space (e.g., a stack or array) to store the inorder traversal results and compare the values. However, this approach requires **O(n)** space, which is less optimal.
- **Morris Traversal (Iterative Inorder Traversal):**
    - An iterative approach that uses **O(1)** extra space and performs the inorder traversal by modifying the tree structure temporarily.
    - This method uses **threaded binary trees** to keep track of nodes during the traversal without using a stack.
- **Level-order Traversal (Breadth-First Search):**
    - Although not ideal for recovering a BST, you could explore using a level-order traversal to identify swapped nodes and correct the tree, but this would be more complex and less efficient than an inorder traversal.

## Applications:

- **Data Structure Validation:** Detect and correct invalid BSTs in data structure operations, ensuring the tree maintains its structural integrity.
- **Database Systems:** Fix corrupted tree structures in indexing systems that rely on BSTs for fast data retrieval and sorting.
- **Memory Management:** Recover or restore data in cases where pointers or references in tree structures are swapped.
- **Graph Traversal:** Applied when working with traversal algorithms in cases where trees might be misrepresented or corrupted due to node-swapping errors.

- **Autocompletion Systems:** If a tree representing a dictionary of words is corrupted, recovering the tree helps in restoring the dictionary's correct order.

## CODE:

```
import java.util.*;
class Node
{
   int data;
   Node left, right;
   Node(int value)
   {
      data = value;
      left = right = null;
   }
}

class Main
{
   static Node buildTree(int[] nodes, int n, int index)
   {
      if (index >= n || nodes[index] == -1)
         return null;
      Node root = new Node(nodes[index]);
      root.left = buildTree(nodes, n, 2 * index + 1);
      root.right = buildTree(nodes, n, 2 * index + 2);
      return root;
   }

   static void correctBSTUtil(Node root, Node[] first, Node[] middle, Node[] last, Node[] prev)
   {
      if (root == null)
         return;
      correctBSTUtil(root.left, first, middle, last, prev);
      if (prev[0] != null && root.data < prev[0].data)
      {
         if (first[0] == null)
         {
            first[0] = prev[0];
            middle[0] = root;
         }
```

```
        else
        {
            last[0] = root;
        }
    }
    prev[0] = root;
    correctBSTUtil(root.right, first, middle, last, prev);
}
static void correctBST(Node root)
{
    Node[] first = {null}, middle = {null}, last = {null}, prev = {null};
    correctBSTUtil(root, first, middle, last, prev);
    if (first[0] != null && last[0] != null)
        swap(first[0], last[0]);
    else if (first[0] != null && middle[0] != null)
        swap(first[0], middle[0]);
}
static void swap(Node a, Node b)
{
    int temp = a.data;
    a.data = b.data;
    b.data = temp;
}
static void printInorder(Node node)
{
    if (node == null)
        return;
    printInorder(node.left);
    System.out.print(node.data + " ");
    printInorder(node.right);
}

public static void main(String[] args)
{
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] nodes = new int[n];
    for (int i = 0; i < n; i++)
    {
        nodes[i] = sc.nextInt();
        }
        Node root = buildTree(nodes, n, 0);
        correctBST(root);
        printInorder(root);
    }
}
```

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