DSA_Task

DSA Excerises Solution

This document presents solutions to a series of practice tasks aimed at enhancing knowledge of **Data Structures and Algorithms** (DSA). The problems are classified into various sections, each focusing on specific DSA concepts, such as **Tree Traversals**, **Depth-First Search (DFS)**, **Breadth-First Search (BFS)**, and more.

Each solution is developed according to the following guidelines:

1. **Explanation of Approach**: A detailed explanation is provided for each solution, outlining the chosen strategy, the data structures involved, and the algorithmic approach.
2. **Time and Space Complexity**: An analysis of both **time complexity** and **space complexity** is included to assess the efficiency and scalability of the solution.
3. **Code Quality and Correctness**: The code is written to be clean, readable, and efficient, ensuring correctness through handling of edge cases and thorough testing.
4. **Reasoning Behind the Approach**: Each approach is justified, with reasoning provided for choosing specific algorithms or methods, highlighting performance trade-offs where applicable.
5. **Algorithmic Techniques**: Various techniques, such as **Graph Traversal (BFS/DFS)**, **Tree Manipulation**, and **Dynamic Programming**, are employed to tackle a diverse set of problems.

The solutions in this document are structured for clarity and coherence, making it easy to follow the thought process behind each approach. Each problem has been addressed with a focus on correctness and optimal performance.

we addressed 2 questions from each section.

Pre-Order Traversal

Question 2:Tree to Sequential Structure Conversion (4 pts)

- Transform a given binary tree into a sequential structure where each node's right pointer

indicates the next element in the sequence, and the left pointer is always null. Perform

this transformation in-place.

- Hint: Utilize a pre-order traversal strategy and keep track of the previously processed

node.

Example 1:

- Input: root = [1,2,5,3,4,null,6]

- Output: [1,null,2,null,3,null,4,null,5,null,6]
  Example 2:
- Input: root = []
- Output: []
  Example 3:
- Input: root = [0]
- Output: [0]
  Constraints:
- The number of nodes in the tree is in the range [0, 2000].
- -100 <= Node.val <= 100

Solution

Approach:

We will perform the following steps:

1. Use a **pre-order traversal** to visit each node.
2. For each visited node:
   - Set its left pointer to `null`.
   - Adjust its right pointer to point to the next node in the pre-order traversal sequence.
3. We'll keep track of the previously visited node to help link the current node properly.
4. This will be done **in-place** without using additional data structures except for some temporary variables.

Time Complexity:

- The time complexity of this approach is **O(n)**, where `n` is the number of nodes in the tree, since we need to visit each node exactly once during the traversal.

Space Complexity:

- The space complexity is **O(h)**, where `h` is the height of the tree due to the recursion stack in pre-order traversal.

Code Solution:

```
class TreeNode:
  def __init__(self, val=0, left=None, right=None):
      self.val = val
      self.left = left
      self.right = right

def flatten(root: TreeNode) -> None:
  if not root:
      return
```

```
    # Initialize previous node as None
    prev = None

    # Pre-order traversal function
    def preorder(node: TreeNode):
        nonlocal prev
        if not node:
            return

        # If there was a previously visited node, adjust its right pointer
        if prev:
            prev.right = node
            prev.left = None

        # Move the 'prev' pointer to the current node
        prev = node

        # Temporarily store the right child to process it after the left subtree
        right = node.right

        # Recursively flatten the left subtree
        preorder(node.left)

        # Recursively flatten the right subtree
        preorder(right)

    # Start the pre-order traversal
    preorder(root)
```

Question 3: Tree Reconstruction from Traversal Sequences (5 pts)

- You are provided with two integer sequences representing the preorder and in-order

traversals of a binary tree. Your task is to reconstruct and return the original binary tree

structure.

- Hint: Begin with the first pre-order element as the root, then recursively build the left and

right subtrees.

Example 1:

- Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
- Output: [3,9,20,null,null,15,7]

Example 2:

- Input: preorder = [-1], inorder = [-1]
- Output: [-1]

Constraints:

- 1 <= preorder.length <= 3000
- inorder.length == preorder.length
- -3000 <= pre order[i], inorder[i] <= 3000
- preorder and inorder consist of unique values.
- Each value of inorder also appears in preorder.
- preorder is guaranteed to be the preorder traversal of the tree.
- inorder is guaranteed to be the inorder traversal of the tree.

Solution

Approach:

1. **Preorder** traversal provides the root of the tree first. Therefore, the first element of the preorder sequence is the root of the tree.
2. **Inorder** traversal gives us information about the left and right subtrees. The elements to the left of the root in the inorder sequence belong to the left subtree, and the elements to the right belong to the right subtree.
3. We will recursively reconstruct the left and right subtrees by splitting the inorder sequence based on the position of the root and adjusting the preorder sequence accordingly.
4. We'll use a helper function to perform the recursive construction.

Time Complexity:

- The time complexity of this approach is **O(n)**, where `n` is the number of nodes in the tree. Each node is processed once, and we use hashing for quick lookup of node positions in the inorder sequence.

Space Complexity:

- The space complexity is **O(n)** due to the space needed to store the tree nodes and the recursion stack.

Code Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def buildTree(preorder: list[int], inorder: list[int]) -> TreeNode:
    # Hash map to store the index of each node in inorder traversal for O(1) access
    inorder_index_map = {val: idx for idx, val in enumerate(inorder)}
```

```
    # Recursive function to build the tree
    def buildTreeRecursive(pre_start, in_start, in_end):
        if pre_start > len(preorder) - 1 or in_start > in_end:
            return None

        # The first element of preorder is the root node
        root_val = preorder[pre_start]
        root = TreeNode(root_val)

        # Find the root's index in inorder traversal
        in_index = inorder_index_map[root_val]

        # Construct left and right subtrees
        root.left = buildTreeRecursive(pre_start + 1, in_start, in_index - 1)
        root.right = buildTreeRecursive(pre_start + (in_index - in_start + 1),
in_index + 1, in_end)

        return root

    # Build and return the tree
    return buildTreeRecursive(0, 0, len(inorder) - 1)
```

Post-Order Traversal

Question 2: Delete Leaves with a Given Value (5 pts)

- You have a tree structure where some leaves need pruning. Remove all leaf nodes with a

specific value, then repeat this process for any nodes that become new leaves due to the

removal. Continue until no more leaves with the target value remain. Your task is to

return the modified tree's root.

- Hint: Use post-order traversal to process children before the parent, allowing you to identify and remove leaf nodes efficiently.

Example 1:

- Input: root = [1,2,3,2,null,2,4], target = 2
- Output: [1,null,3,null,4]
- Explanation: Leaf nodes in green with value (target = 2) are removed (Picture in left).

Solution

Approach:

1. We will use **post-order traversal** because it processes children before the parent. This way, we can check if a node’s children are leaves with the target value before deciding whether to delete the node.

2. For each node:
    - Recursively process the left and right children.
    - If both children are `None` (indicating it's a leaf) and the node's value equals the target value, we delete the node (return `None`).
    - Otherwise, return the node after its children have been processed.
3. This will allow us to remove all target leaves in a single traversal.

Time Complexity:

- The time complexity is **O(n)**, where `n` is the number of nodes in the tree since we must visit each node exactly once.

Space Complexity:

- The space complexity is **O(h)**, where `h` is the height of the tree due to the recursion stack.

Code Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def removeLeafNodes(root: TreeNode, target: int) -> TreeNode:
    if not root:
        return None

    # Recursively process the left and right children
    root.left = removeLeafNodes(root.left, target)
    root.right = removeLeafNodes(root.right, target)

    # Check if current node is a leaf with the target value
    if not root.left and not root.right and root.val == target:
        return None

    return root
```

Question 3: Trimming and Forest Creation in Binary Tree (5 pts)

- Given a binary tree and a list of values to remove, eliminate all nodes containing these

values. Return a list containing the roots of all trees in the resulting forest.

- Hint: Conduct a post-order traversal, removing specified nodes and adding their children to the result as necessary.
  Example 1:

- Input: root = [1,2,3,4,5,6,7], to_delete = [3,5]
- Output: [[1,2,null,4],[6],[7]]
  Example 2:
- Input: root = [1,2,4,null,3], to_delete = [3]
- Output: 1,2,4
  Constraints:
- The number of nodes in the given tree is at most 1000.
- Each node has a distinct value between 1 and 1000.
- to_delete.length <= 1000
- to_delete contains distinct values between 1 and 1000.

Approach:

1. We will use **post-order traversal** to process each node's children before deciding whether to remove the current node.
2. For each node:
   - Recursively process its left and right children.
   - If the node's value matches one of the values in the list of values to remove, its children (if any) are added to the resulting forest.
   - If the node should not be removed, it is returned and linked to its parent. Otherwise, `None` is returned.
3. After traversal, if the root node is not removed, we add it to the forest. The result is a list of all the remaining subtrees.

Time Complexity:

- The time complexity is **O(n)**, where `n` is the number of nodes in the tree since each node is visited once.

Space Complexity:

- The space complexity is **O(h)**, where `h` is the height of the tree due to recursion stack space.

Code Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def delNodes(root: TreeNode, to_delete: list[int]) -> list[TreeNode]:
    to_delete_set = set(to_delete)
    forest = []
```

```
    def postorder(node: TreeNode) -> TreeNode:
        if not node:
            return None

        # Recursively process left and right children
        node.left = postorder(node.left)
        node.right = postorder(node.right)

        # If the current node needs to be deleted
        if node.val in to_delete_set:
            # If the node has children, they become new roots
            if node.left:
                forest.append(node.left)
            if node.right:
                forest.append(node.right)
            # Return None to indicate this node is deleted
            return None

        return node

    # Begin post-order traversal
    root = postorder(root)

    # If the root itself is not deleted, add it to the forest
    if root:
        forest.append(root)

    return forest
```

In-Order Traversal

Question 3: Locating Specific Element in Sorted Tree Structure (3 pts)

- Given the root of a binary search tree and a positive integer k, identify and return the kth

smallest value among all node values in the tree. Assume k is valid and within the range

of the tree's size.

- Hint: Perform an in-order traversal while maintaining a count of visited nodes.
- Example 1:
- Input: root = [3,1,4,null,2], k = 1
- Output: 1
  Example 2:
- Input: root = [5,3,6,2,4,null,null,1], k = 3
- Output: 3

Constraints:

- The number of nodes in the tree is n.
- 1 <= k <= n <= 104

- 0 <= Node.val <= 104

Approach:

1. **In-order traversal** of a BST produces nodes in sorted order. We will leverage this property to find the k-th smallest element.
2. We will perform an in-order traversal while keeping a count of the nodes visited. Once the count equals `k`, we return the current node's value.
3. To optimize space, we can do this iteratively or recursively.

Time Complexity:

- The time complexity is **O(n)** in the worst case, where `n` is the number of nodes in the tree, since we may need to visit all nodes to find the k-th smallest element.

Space Complexity:

- The space complexity is **O(h)**, where `h` is the height of the tree due to recursion stack space (or the stack used in an iterative solution).

Code Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def kthSmallest(root: TreeNode, k: int) -> int:
    # Initialize a counter to track the number of nodes visited
    count = 0
    result = None

    def inorder(node: TreeNode):
        nonlocal count, result
        if not node or result is not None:
            return

        # Traverse the left subtree
        inorder(node.left)

        # Visit the current node
        count += 1
        if count == k:
            result = node.val
            return

        # Traverse the right subtree
```

```
        inorder(node.right)

    # Perform in-order traversal
    inorder(root)

    return result
```

Question 4: Given the Root of a Binary Search Tree, Return the k-th Smallest Value (1-indexed) (3 pts)

Example 1:

- Input: root = [3,1,4,null,2], k = 1
- Output: 1
  Example 2:
- Input: root = [5,3,6,2,4,null,null,1], k = 3
- Output: 3

This question is the same as question 3, where we need to find the **k-th smallest value** in a Binary Search Tree. The approach remains the same, and we've already provided a solution for this.

To recap:

- **Approach**: Use **in-order traversal** to traverse the nodes in ascending order, keeping a count to track when the k-th smallest element is reached.
- **Time Complexity**: **O(n)** (traversing all nodes).
- **Space Complexity**: **O(h)** (recursion stack space).

Code Solution:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def kthSmallest(root: TreeNode, k: int) -> int:
    count = 0
    result = None

    def inorder(node: TreeNode):
        nonlocal count, result
        if not node or result is not None:
            return

        inorder(node.left)
        count += 1
        if count == k:
```

```
            result = node.val
            return
        inorder(node.right)

    inorder(root)
    return result
```

Depth First Search

Question 1: Connected Region Counter (5 pts)

- You are given a m x n 2D grid where '1' represents land and '0' represents water. Count and return the number of distinct land masses, where a land mass is a group of adjacent land cells (horizontally or vertically). You may assume all four edges of the grid are all surrounded by water.
- Hint: Use DFS to explore each land cell, marking visited cells to avoid double-counting.

Example 1:

```
Input: grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]
```

- Output: 1
  Example 2:

```
Input: grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]
```

```
]
```

- Output: 3
  Constraints:
- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 300
- `grid[i][j]` is '0' or '1'.

Approach:

1. Use **Depth First Search (DFS)** to explore each cell in the grid. When encountering a `'1'`, use DFS to explore all the adjacent cells that are also `'1'`, marking them as visited (i.e., changing `'1'` to `'0'` or using a visited set).
2. Every time DFS is initiated from an unvisited `'1'`, it indicates the start of a new connected land mass, so increment the land mass counter.
3. Continue the process until all cells are processed.

Time Complexity:

- The time complexity is **O(m * n)**, where `m` is the number of rows and `n` is the number of columns in the grid, as we must visit each cell once.

Space Complexity:

- The space complexity is **O(m * n)** due to the recursion stack in the worst case when the entire grid is land.

Code Solution:

```
def numIslands(grid: list[list[str]]) -> int:
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    island_count = 0

    # Helper function to perform DFS
    def dfs(r, c):
        # Base case: if out of bounds or the cell is water
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == '0':
            return

        # Mark the cell as visited (change land to water)
```

```
        grid[r][c] = '0'

        # Explore the four adjacent cells
        dfs(r - 1, c)  # up
        dfs(r + 1, c)  # down
        dfs(r, c - 1)  # left
        dfs(r, c + 1)  # right

    # Iterate through each cell in the grid
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':  # Start a DFS for every unvisited land cell
                island_count += 1
                dfs(r, c)

    return island_count
```

Question 2: Prerequisite Fulfillment Checker (5 pts)

- In an academic system with numCourses (labeled from 0 to numCourses-1), you're given a list of prerequisite pairs where prerequisites[i] = [a1, b1] indicates that you must take course b1 first if you want to take course a1.
- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
- Return true if you can finish all courses. Otherwise, return false.
- Hint: Use DFS to identify any circular dependencies in the course prerequisite structure.

Example 1:

- Input: numCourses = 2, prerequisites = 1,0
- Output: true
- Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.
  Example 2:
- Input: numCourses = 2, prerequisites = 1,0],[0,1
- Output: false
- Explanation: There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.
  Constraints:
- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= a1, b1 < numCourses
- All the pairs prerequisites[i] are unique.

Approach:

1. The problem is about detecting a cycle in a **Directed Acyclic Graph (DAG)** where courses are nodes, and prerequisites are directed edges. If a cycle exists, it means that a course depends on

itself, either directly or indirectly, making it impossible to complete the courses.

2. We can use **DFS** to detect cycles. During DFS traversal, mark nodes as:
   - **Unvisited** (not yet processed),
   - **Visiting** (currently being processed, i.e., in the DFS stack),
   - **Visited** (fully processed).
3. If we encounter a node that is currently **Visiting**, we have detected a cycle.
4. If the graph is acyclic, we can finish all courses.

Time Complexity:

- The time complexity is **O(numCourses + prerequisites.length)** because we visit each node (course) and edge (prerequisite) once.

Space Complexity:

- The space complexity is **O(numCourses)** due to the recursion stack and additional data structures to store the graph and state of each node.

Code Solution:

```python
def canFinish(numCourses: int, prerequisites: list[list[int]]) -> bool:
    # Create an adjacency list to represent the graph
    graph = {i: [] for i in range(numCourses)}
    for course, prereq in prerequisites:
        graph[prereq].append(course)

    # State of each course: 0 = unvisited, 1 = visiting, 2 = visited
    state = [0] * numCourses

    # DFS function to check for cycles
    def dfs(course):
        if state[course] == 1:  # Cycle detected
            return False
        if state[course] == 2:  # Already processed
            return True

        # Mark as visiting
        state[course] = 1

        # Visit all the courses that depend on this course
        for next_course in graph[course]:
            if not dfs(next_course):
                return False

        # Mark as visited
        state[course] = 2
        return True
```

```
    # Check each course for cycles
    for course in range(numCourses):
        if not dfs(course):
            return False

    return True
```

Breadth First Search

Question 1: As Far From Land As Possible (4 pts)

-Imagine a map represented by a n x n grid, where some cells are islands with value 1 and others are ocean with value 0. Find a water cell such that its distance to the nearest land cell is maximized, and return the distance. If no land or water exists in the grid, return -1. The distance used in this problem is the Manhattan distance: the distance between two cells (x0, y0) and (x1, y1) is |x0 - x1| + |y0 - y1|.

Example 1:

- Input: grid = 1,0,1],[0,0,0],[1,0,1
- Output: 2
- Explanation: The cell (1, 1) is as far as possible from all the land with distance 2.
  Example 2:
- Input: grid = 1,0,0],[0,0,0],[0,0,0
- Output: 4
- Explanation: The cell (2, 2) is as far as possible from all the land with distance 4.
  Constraints:
- n == grid.length
- n == grid[i].length
- 1 <= n <= 100D
- `grid[i][j]` is 0 or 1

Approach:

1. **Breadth-First Search (BFS)** is perfect for finding the shortest distance from multiple sources (land cells) to the farthest target (water cell).
2. We will:
   - Start the BFS from all the land cells (`1` s) simultaneously, treating all of them as starting points.
   - Use BFS to explore each level outward, layer by layer, until we reach the farthest water cell (`0`).
   - The last layer explored in BFS gives the maximum distance.
3. We need to track visited cells to avoid reprocessing them.

Time Complexity:

- The time complexity is **O(n²)** because we need to process all `n²` cells in the grid.

Space Complexity:

- The space complexity is **O(n²)** because we need to store the queue and visited state for all n^2 cells.

Code Solution:

```
from collections import deque

def maxDistance(grid: list[list[int]]) -> int:
    n = len(grid)
    queue = deque()

    # Start by adding all land cells (1) to the queue
    for r in range(n):
        for c in range(n):
            if grid[r][c] == 1:
                queue.append((r, c))

    # If there are no land or no water cells, return -1
    if len(queue) == 0 or len(queue) == n * n:
        return -1

    # Perform BFS from all land cells
    distance = -1
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Possible 4-directional moves

    while queue:
        distance += 1
        for _ in range(len(queue)):
            r, c = queue.popleft()

            # Explore the 4 possible directions
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < n and 0 <= nc < n and grid[nr][nc] == 0:
                    grid[nr][nc] = 1  # Mark the water cell as visited by changing
it to land
                    queue.append((nr, nc))

    return distance
```

Question 2: Contamination Spread Simulator (5 pts)

- You're presented with a m x n grid where each cell can be in one of three states: 0 (empty), 1 (fresh), or 2 (contaminated). Each minute, any fresh cell adjacent (up, down, left, right) to a contaminated cell becomes contaminated. Return the minimum number of minutes required for all fresh cells to become contaminated, or return -1 if it's impossible.
- Hint: Implement BFS starting from all contaminated cells simultaneously to model the spread of contamination.

Example 1:

- Input: grid = 2,1,1],[1,1,0],[0,1,1
- Output: 4
  Example 2:
- Input: grid = 2,1,1],[0,1,1],[1,0,1
- Output: -1
- Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.
  Example 3:
- Input: grid = 0,2
- Output: 0
- Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.
  Constraints:
- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 10
- `grid[i][j]` is 0, 1, or 2.

Approach:

1. **Breadth-First Search (BFS)** can be used to simulate the contamination process. The BFS starts from all contaminated cells (`2` s) simultaneously, and at each step, spreads to adjacent fresh cells (`1` s).
2. We will:
   - Initialize the BFS with all contaminated cells as the starting points.
   - For each level in BFS (representing one minute), we contaminate all adjacent fresh cells.
   - Keep track of the total number of fresh cells. If we manage to contaminate all of them, return the total time; otherwise, return `-1` .
3. If there are no fresh cells at the beginning, return `0` .

Time Complexity:

- The time complexity is **O(m * n)**, where `m` and `n` are the dimensions of the grid, as we need to process all cells.

Space Complexity:

- The space complexity is **O(m * n)** because we need space for the queue and for marking visited cells.

Code Solution:

```python
from collections import deque

def minMinutesToSpread(grid: list[list[int]]) -> int:
    m, n = len(grid), len(grid[0])
    queue = deque()
    fresh_count = 0

    # Initialize the queue with all contaminated cells and count fresh cells
    for r in range(m):
        for c in range(n):
            if grid[r][c] == 2:
                queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1

    # If there are no fresh cells, return 0
    if fresh_count == 0:
        return 0

    # Directions for 4-directional movement
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    minutes = 0

    # Perform BFS to spread the contamination
    while queue:
        minutes += 1
        for _ in range(len(queue)):
            r, c = queue.popleft()

            # Explore adjacent cells
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < m and 0 <= nc < n and grid[nr][nc] == 1:
                    grid[nr][nc] = 2  # Contaminate the fresh cell
                    fresh_count -= 1
                    queue.append((nr, nc))

        # If there are no fresh cells left, we are done
        if fresh_count == 0:
            return minutes

    # If there are still fresh cells left, return -1
    return -1
```

Expert challenge

Question 1: Car Routes - Least Number of Cars to Travel (8 pts)

- You are given an array routes representing car routes where routes[i] is a car route that the ith car repeats forever.
- For example, if routes[0] = [1, 5, 7], this means that the 0th car travels in the sequence 1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ... forever.

- You will start at the car stop source (You are not on any car initially), and you want to go to the car stop target. You can travel between car stops by cars only.
- Return the least number of cars you must take to travel from source to target. Return -1 if it is not possible.
  Example 1:
- Input: routes = 1,2,7],[3,6,7, source = 1, target = 6
- Output: 2
- Explanation: The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.
  Example 2:
- Input: routes = 7,12],[4,5,15],[6],[15,19],[9,12,13, source = 15, target = 12
- Output: -1
  Constraints:
- 1 <= routes.length <= 500.
- 1 <= routes[i].length <= 105
- All the values of routes[i] are unique.
- sum(routes[i].length) <= 105
- 0 <= routes[i][j] < 106
- 0 <= source, target < 106

Approach:

1. This is a graph traversal problem where each car's route represents a node, and an edge exists between two nodes (routes) if they share a common stop.
2. We will build a **graph** of routes, where each route (car) is connected to another if they share any stop. Then, we will perform a **Breadth-First Search (BFS)** starting from the routes that include the `source` stop.
3. The BFS will explore the minimum number of car changes required to reach the `target` stop.

Steps:

1. Create a mapping between each stop and the list of routes (cars) that visit that stop.
2. Perform BFS starting from the routes that visit the `source`. For each route, check all the other routes connected via shared stops. Track the visited routes to avoid cycles.
3. If a route visits the `target`, return the number of route switches taken. If BFS completes without reaching the target, return `-1`.

Time Complexity:

- The time complexity is **O(N + E)**, where `N` is the total number of stops across all routes and `E` is the total number of route-to-route connections (edges in the graph).

Space Complexity:

- The space complexity is **O(N + E)** due to the graph representation and the BFS queue.

Code Solution:

```
from collections import defaultdict, deque

def numBusesToDestination(routes: list[list[int]], source: int, target: int) -> int:
    if source == target:
        return 0

    # Map each stop to the list of routes that visit it
    stop_to_routes = defaultdict(list)
    for i, route in enumerate(routes):
        for stop in route:
            stop_to_routes[stop].append(i)

    # BFS initialization
    visited_routes = set()
    visited_stops = set([source])
    queue = deque([(source, 0)])  # (current stop, number of buses taken)

    while queue:
        stop, bus_count = queue.popleft()

        # Check all routes passing through the current stop
        for route in stop_to_routes[stop]:
            if route in visited_routes:
                continue

            # Mark the route as visited
            visited_routes.add(route)

            # Traverse all stops of this route
            for next_stop in routes[route]:
                if next_stop == target:
                    return bus_count + 1
                if next_stop not in visited_stops:
                    visited_stops.add(next_stop)
                    queue.append((next_stop, bus_count + 1))

    return -1
```

Question 2: Crack the Safe (8 pts)

- There is a safe protected by a password. The password is a sequence of n digits where each digit can be in the range [0, k - 1]. The safe has a peculiar way of checking the password. When you enter in a sequence, it checks the most recent n digits that were entered each time you type a digit.
- For example, the correct password is "345" and you enter in "012345":
  - After typing 0, the most recent 3 digits is "0", which is incorrect.
  - After typing 1, the most recent 3 digits is "01", which is incorrect.

    - After typing 2, the most recent 3 digits is "012", which is incorrect.
    - After typing 3, the most recent 3 digits is "123", which is incorrect.
    - After typing 4, the most recent 3 digits is "234", which is incorrect.
    - After typing 5, the most recent 3 digits is "345", which is correct and the safe unlocks.
- Return any string of minimum length that will unlock the safe at some point of entering it.

Example 1:

- Input: n = 1, k = 2
- Output: "10"
- Explanation: The password is a single digit, so enter each digit. "01" would also unlock the safe. Example 2:
- Input: n = 2, k = 2
- Output: "01100"
- Explanation: For each possible password:

    - "00" is typed in starting from the 4th digit.
    - "01" is typed in starting from the 1st digit.
    - "10" is typed in starting from the 3rd digit.
    - "11" is typed in starting from the 2nd digit.

    Thus "01100" will unlock the safe. "10011", and "11001" would also unlock the safe.

Approach:

1. We will use algorithm to construct the sequence using a depth-first search (DFS) approach. This algorithm constructs an circuit on a graph where:

- Each node represents a combination of `n-1` digits.
- Each edge represents an additional digit, forming an `n`-digit sequence.

2. The idea is to construct the shortest path that visits every edge exactly once, ensuring that all combinations of `n` digits appear as a sub-sequence in the final result.

Steps:

1. Initialize an empty result string and start building the sequence.
2. Use DFS to explore all possible edges starting from a node.
3. Append the digits from the edges visited during the traversal to construct the sequence.

Time Complexity:

- The time complexity is **O(k^n)**, where `k` is the number of possible digits and `n` is the length of the password.

Space Complexity:

- The space complexity is **O(k^n)** due to the storage needed for the graph and the result string.

Code Solution:

```python
def crackSafe(n: int, k: int) -> str:
    # Initialize the starting node
    start = '0' * (n - 1)
    visited = set()
    result = []

    def dfs(node):
        for x in map(str, range(k)):
            next_node = node + x
            if next_node not in visited:
                visited.add(next_node)
                dfs(next_node[1:])
                result.append(x)

    dfs(start)
    return ''.join(result) + start
```