ADVANCED PROGRAMMING - II

Fast Learners' Lab Assignment

JIYA

22BCS14856

1. Set Matrix Zeroes

Problem Statement: Given an m x n matrix, if an element is 0, set its entire row and column to 0. The modification must be done in place without using additional storage for another matrix.

Example 1: Input: matrix = [[1, 1, 1], [1, 0, 1], [1, 1, 1]] Output: [[1, 0, 1], [0, 0, 0], [1, 0, 1]]

Explanation: The element at position (1,1) is 0. Therefore, the entire row 1 and column 1 are set to 0.

Example 2: Input: matrix = [[0, 1, 2, 0], [3, 4, 5, 2], [1, 3, 1, 5]] Output: [[0, 0, 0, 0], [0, 4, 5, 0], [0, 3, 1, 0]]

Explanation: The zeros in the first row (positions (0,0) and (0,3)) cause the entire first row and their corresponding columns to be set to 0.

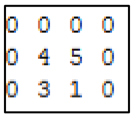
Code:

```
public class Main {
    public static void setZeroes(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return;
        }
        int rows = matrix.length;
        int cols = matrix[0].length;
        boolean firstRowHasZero = false;
        boolean firstColHasZero = false;
        for (int j = 0; j < cols; j++) {
            if (matrix[0][j] == 0) {
                firstRowHasZero = true;
                break;
            }
        }
        for (int i = 0; i < rows; i++) {
            if (matrix[i][0] == 0) {
                firstColHasZero = true;
                break;
            }
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < cols; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < cols; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }
        if (firstRowHasZero) {
            for (int j = 0; j < cols; j++) {
                matrix[0][j] = 0;
            }
        }
        if (firstColHasZero) {
            for (int i = 0; i < rows; i++) {
                matrix[i][0] = 0;
            }
        }
    }
    public static void main(String[] args) {
        int[][] matrix2 = {
            {0, 1, 2, 0},
            {3, 4, 5, 2},
            {1, 3, 1, 5}
        };
        setZeroes(matrix2);
        printMatrix(matrix2);
    }
    private static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }
        System.out.println();
    }
}
```

Input:

```
{0, 1, 2, 0},
{3, 4, 5, 2},
{1, 3, 1, 5}
```

Output:



```
0 0 0 0
0 4 5 0
0 3 1 0
```

Time Complexity: O(m*n)

The code has two nested loops iterating over the rows and columns of the matrix, resulting in a time complexity of O(m*n) where m is the number of rows and n is the number of columns in the matrix.

2. Reverse Linked List - II

Problem Statement: Given the head of a singly linked list and two integers left and right, reverse the nodes of the list from position left to right, and return the modified list.

Example 1:

Input: Linked list: [1, 2, 3, 4, 5]; left = 2; right = 4

Output: [1, 4, 3, 2, 5]

Explanation: The sublist from position 2 to 4 ([2, 3, 4]) is reversed to become [4, 3, 2], while the rest of the list remains unchanged.

Example 2:

Input: Linked list: [1, 2, 3, 4, 5]; left = 1; right = 5

Output: [5, 4, 3, 2, 1]

Explanation: The entire list is reversed because the reversal starts at the first node and ends at the last node.

Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class Main {
    public static ListNode reverseBetween(ListNode head, int left, int right) {
        if (head == null || left == right) {
            return head;
        }

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;

        for (int i = 0; i < left - 1; i++) {
            prev = prev.next;
        }

        ListNode start = prev.next;
        ListNode then = start.next;

        for (int i = 0; i < right - left; i++) {
            start.next = then.next;
            then.next = prev.next;
            prev.next = then;
            then = start.next;
        }

        return dummy.next;
    }

    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {

        ListNode head = new ListNode(7);
        head3.next = new ListNode(9);
        head3.next.next = new ListNode(1);
        head3.next.next.next = new ListNode(8);
        head3.next.next.next.next = new ListNode(5);
        head3.next.next.next.next.next = new ListNode(2);

        ListNode result3 = reverseBetween(head, 3, 5);
        printList(result3);
    }
}
```

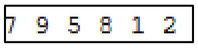
Input:

7->9->1->8->5->2

Left: 3

Right: 5

Output:



```
7 9 5 8 1 2
```

Time Complexity: O(n)

The code iterates through the linked list once, performing constant time operations at each step. Therefore, the time complexity is O(n), where n is the number of nodes in the linked list.

3. Detect a Cycle in a Linked List

Problem Statement: Given the head of a linked list, determine whether the linked list contains a cycle. A cycle occurs if a node's next pointer points to a previous node in the list.

Example 1:

Input: Linked list: [3, 2, 0, -4] with the tail node (-4) pointing to the node with value 2.

Output: true

Explanation: The tail node connects back to an earlier node, forming a cycle.

Example 2:

Input: Linked list: [1, 2] with no cycle (each node points to null at the end).

Output: false

Explanation: There is no cycle since no node points back to a previous node.

Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class Main {
    public static boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }

        ListNode slow = head;
        ListNode fast = head.next;

        while (fast != null && fast.next != null) {
            if (slow == fast) {
                return true;
            }
            slow = slow.next;
            fast = fast.next.next;
        }

        return false;
    }
```

```
        public static void main(String[] args) {
            ListNode head3 = new ListNode(5);
            head3.next = new ListNode(6);
            head3.next.next = new ListNode(7);
            head3.next.next.next = new ListNode(8);
            head3.next.next.next.next = head3.next.next;

            System.out.println(hasCycle(head3));
        }
    }
```

Input:

```
5 -> 6 -> 7 -> 8
          ^     |
          |_____|
```

Output:

```
True
```

Time Complexity: O(n)

The code uses two pointers, slow and fast, to traverse the linked list. The fast pointer moves twice as fast as the slow pointer. If there is a cycle in the linked list, the fast pointer will eventually catch up to the slow pointer. Therefore, the time complexity is O(n) where n is the number of nodes in the linked list.

4. Longest Increasing Subsequence – II

Problem Statement: Given an integer array nums, find the length of the longest strictly increasing subsequence. A subsequence is derived from the array by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: nums = [10, 9, 2, 5, 3, 7, 101, 18]

Output: 4

Explanation: One longest increasing subsequence is [2, 3, 7, 101], which has a length of 4.

Example 2:

Input: nums = [0, 1, 0, 3, 2, 3]

Output: 4

Explanation: One valid subsequence is [0, 1, 2, 3] with a length of 4.

Code:

```
public class Main {
    public static int lengthOfLIS(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int n = nums.length;
        int[] dp = new int[n];
        int maxLength = 1;

        for (int i = 0; i < n; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        int[] nums3 = {7, 2, 5, 8, 1, 9, 3, 10};
        System.out.println(lengthOfLIS(nums3));
    }
}
```

Input:

```
    [7, 2, 5, 8, 1, 9, 3, 10]
```

Output:

```
        5
```

Time Complexity: O(n^2)

The code contains a nested loop where the outer loop runs for 'n' iterations and the inner loop runs for 'i' iterations. Therefore, the overall time complexity is O(n^2).

5. Word Break

Problem Statement: Given a string s and a dictionary wordDict containing a list of words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. The same word can be reused multiple times.

Example 1:

Input: s = "leetcode", wordDict = ["leet", "code"]

Output: true

Explanation: The string "leetcode" can be segmented as "leet code", where both "leet" and "code" are present in the dictionary.

Example 2:

Input: s = "applepenapple", wordDict = ["apple", "pen"]

Output: true

Explanation: The string can be segmented as "apple pen apple". Reusing "apple" is allowed, and both words exist in the dictionary.

Code:

```
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Main {
    public static boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordSet = new HashSet<>(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;

        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordSet.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.length()];
    }

    public static void main(String[] args) {

        List<String> wordDict3 = List.of("cat", "cats", "and", "sand", "dog");
        System.out.println(wordBreak("catsandog", wordDict3));
        }
    }
```

Input:

```
["cat", "cats", "and", "sand", "dog"]
```

Output:

```
false
```

Time Complexity: O(n^2)

The nested loops iterate through the length of the string 's', resulting in a time complexity of O(n^2), where n is the length of the input string 's'.