Coding Practice Set 7

1. **Next permutation**

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

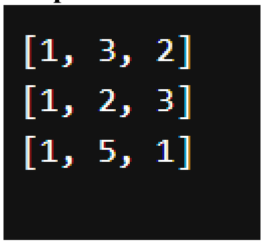
- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.
The replacement must be in place and use only constant extra memory.

Example 1:
Input: nums = [1,2,3]
Output: [1,3,2]
Example 2:
Input: nums = [3,2,1]
Output: [1,2,3]
Example 3:
Input: nums = [1,1,5]
Output: [1,5,1]

Code:
```
class Solution {
    public void nextPermutation(int[] nums) {
        int n = nums.length;
        int i = n - 2;

        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        if (i >= 0) {
            int j = n - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }

        reverse(nums, i + 1, n - 1);
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    private void reverse(int[] nums, int start, int end) {
        while (start < end) {
            swap(nums, start, end);
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        int[] nums1 = {1, 2, 3};
        solution.nextPermutation(nums1);
        System.out.println(java.util.Arrays.toString(nums1));

        int[] nums2 = {3, 2, 1};
        solution.nextPermutation(nums2);
        System.out.println(java.util.Arrays.toString(nums2));

        int[] nums3 = {1, 1, 5};
        solution.nextPermutation(nums3);
        System.out.println(java.util.Arrays.toString(nums3));
    }
}
```

Output:

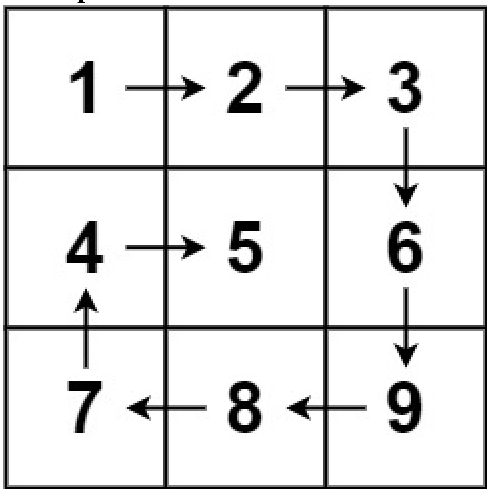


```
[1, 3, 2]
[1, 2, 3]
[1, 5, 1]
```

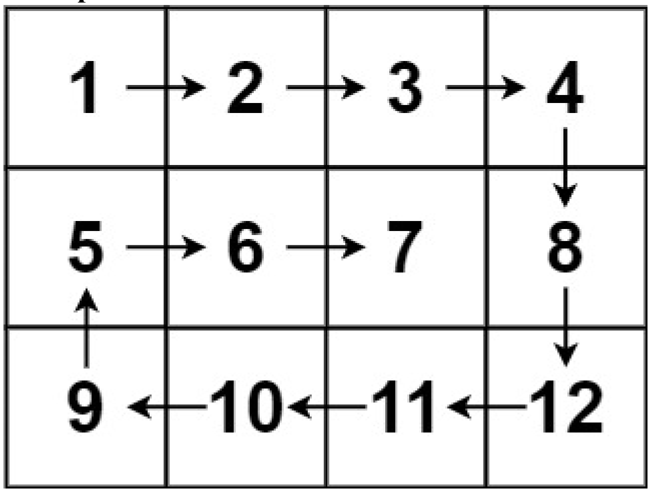
Time Complexity: O(n)

2. **Spiral matrix**
Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

Example 1:



Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
Example 2:



Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]

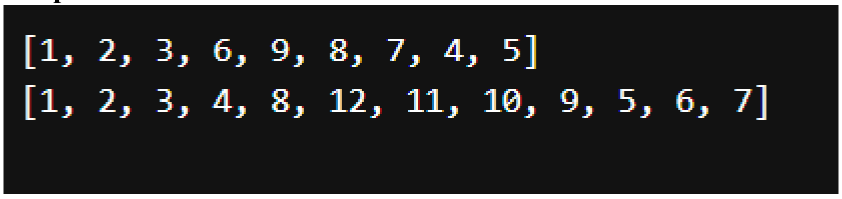
Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 10
- -100 <= matrix[i][j] <= 100

Code:

```
import java.util.List;
import java.util.ArrayList;

class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        int top = 0, bottom = matrix.length - 1, left = 0, right = matrix[0].length - 1;
        while (top <= bottom && left <= right) {
            for (int i = left; i <= right; i++) result.add(matrix[top][i]);
            top++;

            for (int i = top; i <= bottom; i++) result.add(matrix[i][right]);
            right--;

            if (top <= bottom) {
                for (int i = right; i >= left; i--) result.add(matrix[bottom][i]);
                bottom--;
            }

            if (left <= right) {
                for (int i = bottom; i >= top; i--) result.add(matrix[i][left]);
                left++;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        int[][] matrix1 = {{1,2,3},{4,5,6},{7,8,9}};
        System.out.println(solution.spiralOrder(matrix1));

        int[][] matrix2 = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
        System.out.println(solution.spiralOrder(matrix2));
    }
}
```

Output:



```
[1, 2, 3, 6, 9, 8, 7, 4, 5]
[1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7]
```

Time Complexity: O(m * n)

3. **Longest substring without repeating characters**
   Given a string s, find the length of the **longest substring** without repeating characters.

   Example 1:
   Input: s = "abcabcbb"
   Output: 3
   Explanation: The answer is "abc", with the length of 3.
   Example 2:
   Input: s = "bbbbb"
   Output: 1
   Explanation: The answer is "b", with the length of 1.
   Example 3:
   Input: s = "pwwkew"
   Output: 3
   Explanation: The answer is "wke", with the length of 3.
   Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

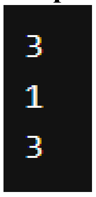
   Constraints:

- 0 <= s.length <= 5 * 10^4
- s consists of English letters, digits, symbols and spaces.

Code:

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

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int maxLength = 0;
        Set<Character> set = new HashSet<>();
        int left = 0;

        for (int right = 0; right < n; right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }
            set.add(s.charAt(right));
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        String s1 = "abcabcbb";
        System.out.println(solution.lengthOfLongestSubstring(s1));

        String s2 = "bbbbb";
        System.out.println(solution.lengthOfLongestSubstring(s2));

        String s3 = "pwwkew";
        System.out.println(solution.lengthOfLongestSubstring(s3));
    }
}
```

Output:

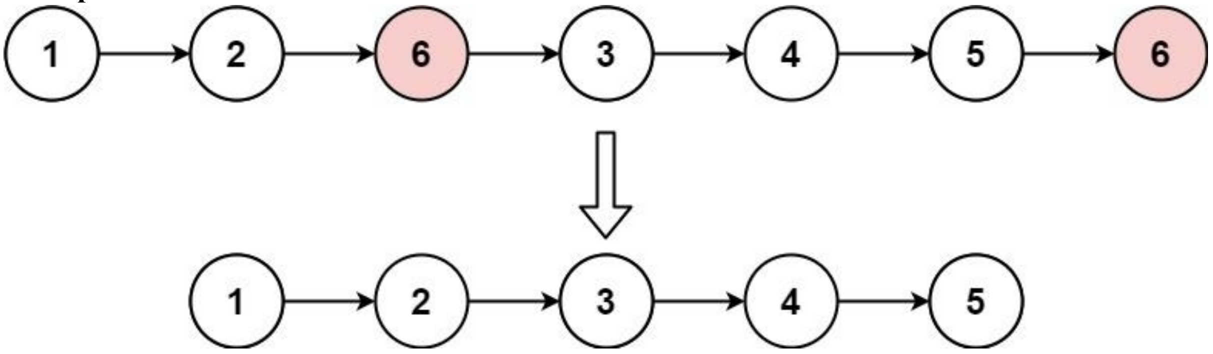


```
3
1
3
```

Time Complexity: O(n)

4. **Remove linked list elements**
Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.

Example 1:

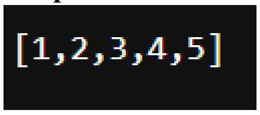


Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
Example 2:
Input: head = [], val = 1
Output: []
Example 3:
Input: head = [7,7,7,7], val = 7
Output: []

Constraints:
- The number of nodes in the list is in the range [0, 10^4].
- 1 <= Node.val <= 50
- 0 <= val <= 50

Code:
```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode temp = new ListNode(0), curr = temp;
        temp.next = head;
        while (curr.next != null) {
            if (curr.next.val == val) curr.next = curr.next.next;
            else curr = curr.next;
        }
        return temp.next;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();

        ListNode head1 = new ListNode(1, new ListNode(2, new ListNode(6, new ListNode(3,
new ListNode(4, new ListNode(5, new ListNode(6)))))));
        ListNode result1 = solution.removeElements(head1, 6);
        printList(result1);
    }

    private static void printList(ListNode head) {
        ListNode current = head;
        System.out.print("[");
        while (current != null) {
            System.out.print(current.val);
            if (current.next != null) System.out.print(", ");
            current = current.next;
        }
        System.out.println("]");
    }
}
```

Output:

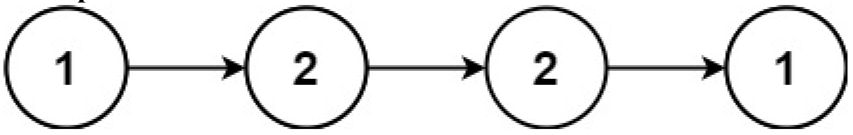


```
[1,2,3,4,5]
```

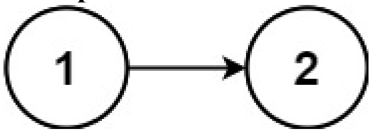
Time Complexity: O(n)

5. **Palindrome linked list**
Given the head of a singly linked list, return true *if it is a palindrome or* false *otherwise*.

Example 1:



Input: head = [1,2,2,1]
Output: true
Example 2:



Input: head = [1,2]
Output: false

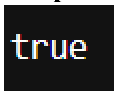
Constraints:

- The number of nodes in the list is in the range [1, 10^5].
- 0 <= Node.val <= 9

Code:

```
import java.util.ArrayList;
import java.util.List;

class Solution {
   public boolean isPalindrome(ListNode head) {
      List<Integer> list = new ArrayList<>();
      while (head != null) {
         list.add(head.val);
         head = head.next;
      }

      int left = 0;
      int right = list.size() - 1;
      while (left < right && list.get(left) == list.get(right)) {
         left++;
         right--;
      }
      return left >= right;
   }

   public static void main(String[] args) {
      Solution solution = new Solution();

      ListNode head1 = new ListNode(1, new ListNode(2, new ListNode(2, new
ListNode(1))));
      System.out.println(solution.isPalindrome(head1));
   }
}
```

Output:

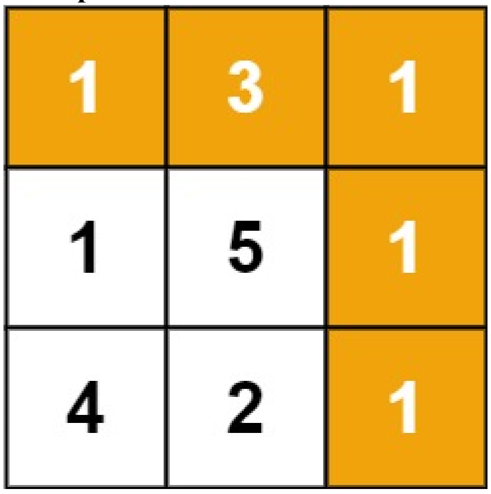


```
true
```

Time Complexity: O(n)

6. **Minimum path sum**
Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.

Example 1:



Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
Example 2:
Input: grid = [[1,2,3],[4,5,6]]
Output: 12

Constraints:

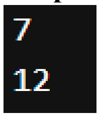
- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 200
- 0 <= grid[i][j] <= 200

Code:

```
class Solution {
  public int minPathSum(int[][] grid) {
    int m = grid.length;
    int n = grid[0].length;

    for (int i = 1; i < m; i++) {
      grid[i][0] += grid[i - 1][0];
    }

    for (int j = 1; j < n; j++) {
      grid[0][j] += grid[0][j - 1];
    }

    for (int i = 1; i < m; i++) {
      for (int j = 1; j < n; j++) {
        grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
      }
```

```
        }

        return grid[m - 1][n - 1];
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        int[][] grid1 = {{1, 3, 1}, {1, 5, 1}, {4, 2, 1}};
        System.out.println(solution.minPathSum(grid1));

        int[][] grid2 = {{1, 2, 3}, {4, 5, 6}};
        System.out.println(solution.minPathSum(grid2));
    }
}
```

Output:



```
7
12
```

Time **complexity**: **O(m × n)**

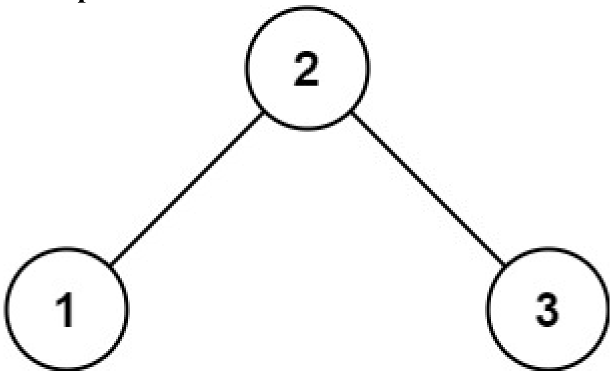
7. **Validate binary search tree**

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

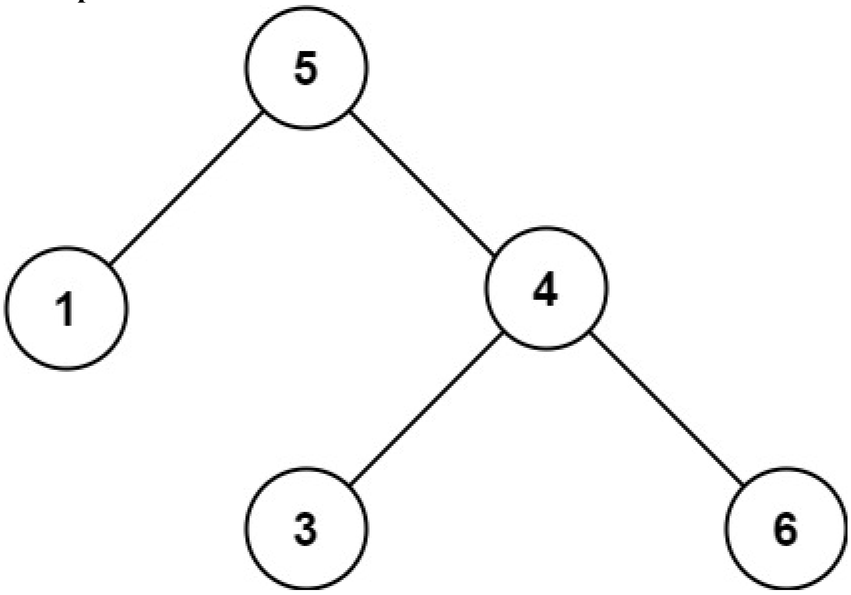
A **valid BST** is defined as follows:

- The left subtree of a node contains only nodes with keys **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



Input: root = [2,1,3]
Output: true
Example 2:



Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.

Constraints:

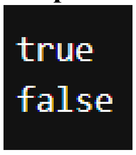
- The number of nodes in the tree is in the range [1, 10^4].
- -2^{31} <= Node.val <= $2^{31} - 1$

Code:

```
class Solution {
   public boolean isValidBST(TreeNode root) {
      return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);
   }
```

```
    private boolean validate(TreeNode node, long min, long max) {
        if (node == null) return true;
        if (node.val <= min || node.val >= max) return false;
        return validate(node.left, min, node.val) && validate(node.right, node.val, max);
    }

    public static void main(String[] args) {
        Solution sol = new Solution();

        TreeNode root1 = new TreeNode(2, new TreeNode(1), new TreeNode(3));
        System.out.println(sol.isValidBST(root1));

        TreeNode root2 = new TreeNode(5, new TreeNode(1),
                         new TreeNode(4, new TreeNode(3), new TreeNode(6)));
        System.out.println(sol.isValidBST(root2));
    }
}
```

Output:



```
true
false
```

Time Complexity: O(n)

8. **Word Ladder**

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- s_k == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the **number of words** in the **shortest transformation sequence** from* beginWord *to* endWord*, or* 0 *if no such sequence exists.*

Example 1:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.
Example 2:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

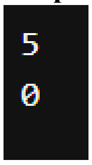
Constraints:

- 1 <= beginWord.length <= 10
- endWord.length == beginWord.length
- 1 <= wordList.length <= 5000
- wordList[i].length == beginWord.length
- beginWord, endWord, and wordList[i] consist of lowercase English letters.
- beginWord != endWord
- All the words in wordList are **unique**.

Code:
```
import java.util.*;

class Solution {
   public int ladderLength(String beginWord, String endWord, List<String> wordList) {
      Set<String> wordSet = new HashSet<>(wordList);
      if (!wordSet.contains(endWord)) return 0;

      Queue<String> queue = new LinkedList<>();
      queue.add(beginWord);
      int level = 1;

      while (!queue.isEmpty()) {
         int size = queue.size();
         for (int i = 0; i < size; i++) {
            String word = queue.poll();
            char[] wordChars = word.toCharArray();
            for (int j = 0; j < wordChars.length; j++) {
               char originalChar = wordChars[j];
               for (char c = 'a'; c <= 'z'; c++) {
                  wordChars[j] = c;
```

```
                        String newWord = new String(wordChars);
                        if (newWord.equals(endWord)) return level + 1;
                        if (wordSet.contains(newWord)) {
                            queue.add(newWord);
                            wordSet.remove(newWord);
                        }
                    }
                    wordChars[j] = originalChar;
                }
            }
            level++;
        }
        return 0;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();

        List<String> wordList1 = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");
        System.out.println(sol.ladderLength("hit", "cog", wordList1));

        List<String> wordList2 = Arrays.asList("hot", "dot", "dog", "lot", "log");
        System.out.println(sol.ladderLength("hit", "cog", wordList2));
    }
}
```

Output:



```
5
0
```

Time Complexity: O(M×N)

9. **Word ladder II**

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- s_k == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *all the **shortest transformation sequences** from* beginWord *to* endWord*, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words* [beginWord, s_1, s_2, ..., s_k].

Example 1:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
Example 2:
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

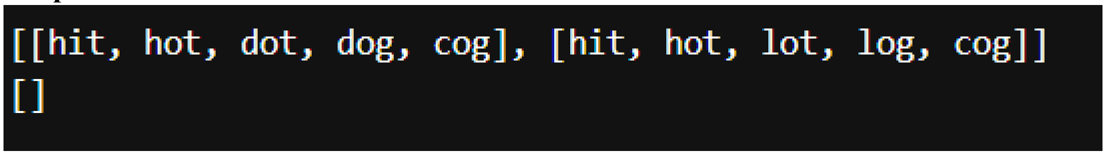
- 1 <= beginWord.length <= 5
- endWord.length == beginWord.length
- 1 <= wordList.length <= 500
- wordList[i].length == beginWord.length
- beginWord, endWord, and wordList[i] consist of lowercase English letters.
- beginWord != endWord
- All the words in wordList are **unique**.
- The **sum** of all shortest transformation sequences does not exceed 10^5.

Code:

```
import java.util.*;
class Solution {
   public List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {
        List<List<String>> result = new ArrayList<>();
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return result;

        Map<String, List<String>> graph = new HashMap<>();
        Map<String, Integer> distance = new HashMap<>();
        bfs(beginWord, endWord, wordSet, graph, distance);

        List<String> path = new ArrayList<>();
        path.add(beginWord);
        dfs(beginWord, endWord, graph, distance, path, result);

        return result;
```

```
    }

    private void bfs(String beginWord, String endWord, Set<String> wordSet, Map<String,
List<String>> graph, Map<String, Integer> distance) {
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        distance.put(beginWord, 0);
        wordSet.add(endWord);

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String word = queue.poll();
                for (String neighbor : getNeighbors(word, wordSet)) {
                    graph.computeIfAbsent(word, k -> new ArrayList<>()).add(neighbor);
                    if (!distance.containsKey(neighbor)) {
                        distance.put(neighbor, distance.get(word) + 1);
                        queue.add(neighbor);
                    }
                }
            }
        }
    }

    private void dfs(String current, String endWord, Map<String, List<String>> graph,
Map<String, Integer> distance, List<String> path, List<List<String>> result) {
        if (current.equals(endWord)) {
            result.add(new ArrayList<>(path));
            return;
        }
        if (!graph.containsKey(current)) return;

        for (String neighbor : graph.get(current)) {
            if (distance.get(neighbor) == distance.get(current) + 1) {
                path.add(neighbor);
                dfs(neighbor, endWord, graph, distance, path, result);
                path.remove(path.size() - 1);
            }
        }
    }

    private List<String> getNeighbors(String word, Set<String> wordSet) {
        List<String> neighbors = new ArrayList<>();
        char[] wordChars = word.toCharArray();
        for (int i = 0; i < wordChars.length; i++) {
            char originalChar = wordChars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                wordChars[i] = c;
                String newWord = new String(wordChars);
                if (wordSet.contains(newWord) && !newWord.equals(word)) {
                    neighbors.add(newWord);
                }
            }
            wordChars[i] = originalChar;
        }
        return neighbors;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();

        List<String> wordList1 = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");
        System.out.println(sol.findLadders("hit", "cog", wordList1));
        List<String> wordList2 = Arrays.asList("hot", "dot", "dog", "lot", "log");
        System.out.println(sol.findLadders("hit", "cog", wordList2));
    }
}
```

Output:



```
[[hit, hot, dot, dog, cog], [hit, hot, lot, log, cog]]
[]
```

Time Complexity: DFS O(V+E)

10. **Course Schedule**

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you **must** take course b_i first if you want to take course a_i.

- 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.

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.

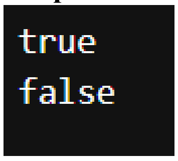
Constraints:

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= a_i, b_i < numCourses
- All the pairs prerequisites[i] are **unique**.

Code:

```
import java.util.*;

class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        int[] inDegree = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] prerequisite : prerequisites) {
            graph.get(prerequisite[1]).add(prerequisite[0]);
            inDegree[prerequisite[0]]++;
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                queue.add(i);
            }
        }

        int count = 0;
        while (!queue.isEmpty()) {
            int course = queue.poll();
            count++;
            for (int neighbor : graph.get(course)) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    queue.add(neighbor);
                }
            }
        }
        return count == numCourses;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();

        int numCourses1 = 2;
        int[][] prerequisites1 = {{1, 0}};
        System.out.println(sol.canFinish(numCourses1, prerequisites1));

        int numCourses2 = 2;
        int[][] prerequisites2 = {{1, 0}, {0, 1}};
        System.out.println(sol.canFinish(numCourses2, prerequisites2));
    }
}
```

Output:



```
true
false
```

Time Complexity: O(V+E)