1.

Description | Accepted | Editorial | Solutions | Submissions

16. 3Sum Closest

Solved

Medium | Topics | Companies

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`.

Return *the sum of the three integers.*

You may assume that each input would have exactly one solution.

Example 1:

> **Input:** nums = [-1,2,1,-4], target = 1
> **Output:** 2
> **Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Example 2:

> **Input:** nums = [0,0,0], target = 1
> **Output:** 0

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        int closest = nums[0] + nums[1] + nums[2];

        for (int i = 0; i < nums.length - 2; i++) {
            for (int j = i + 1; j < nums.length - 1; j++) {
                for (int k = j + 1; k < nums.length; k++) {
                    int currentSum = nums[i] + nums[j] + nums[k];
                    if (Math.abs(target - currentSum) < Math.abs(target - closest)) {
                        closest = currentSum;
                    }
                }
            }
        }

        return closest;
    }
}
```

Time Complexity: O(n^2)

98. Validate Binary Search Tree

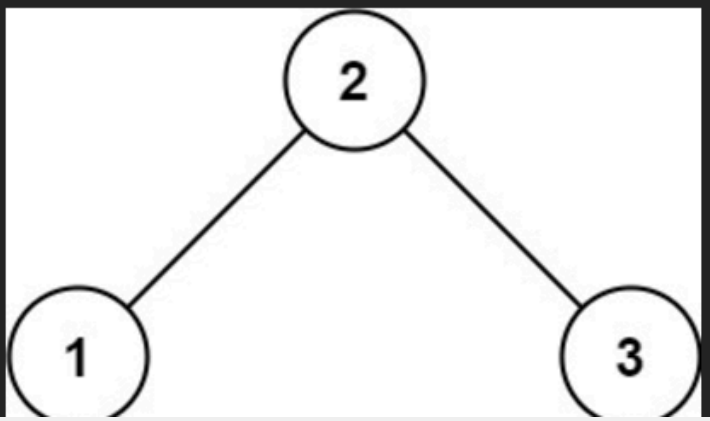
Medium | Topics | Companies

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:




```java
class Solution {
    public boolean isValidBST(TreeNode root) {
   if (root == null) return true;
   Stack<TreeNode> stack = new Stack<>();
   TreeNode pre = null;
   while (root != null || !stack.isEmpty()) {
      while (root != null) {
         stack.push(root);
         root = root.left;
      }
      root = stack.pop();
      if(pre != null && root.val <= pre.val) return false;
      pre = root;
      root = root.right;
   }
   return true;
}
}
```

Time Complexity: : O(n)

3.

127. Word Ladder

Solved

Hard | Topics | Companies

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

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

```java
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<>();
        Boolean isPresent = false;
        wordSet.addAll(wordList);
        for (String currWord : wordList) {
            if (endWord.equals(currWord)) {
                isPresent = true;
                break;
            }
        }
        if (!isPresent) return 0;
        Queue<String> wordQueue = new LinkedList<>();.
        wordQueue.add(beginWord);
        int distance = 0;
        while (!wordQueue.isEmpty()) {
            int size = wordQueue.size();
            distance++;
            while (size-- != 0) {
                String currWord = wordQueue.poll();
                for (int i = 0; i < currWord.length(); i++) {
```

```
                    char[] temp = currWord.toCharArray();
                    for (char j = 'a'; j <= 'z'; j++) {
                        temp[i] = j;

                        String newWord = new String(temp);

                        if (newWord.equals(endWord)) return distance + 1;
                        if (wordSet.contains(newWord)) {
                            wordQueue.add(newWord);
                            wordSet.remove(newWord);
                            System.out.println(newWord);
                        }
                    }
                }
            }
        }
        return 0;
    }
}
```

Time Complexity : O(n*m)

4.

126. Word Ladder II

Solved

Hard | Topics | Companies

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> s1 -> s2 -> ... -> sk` such that:

- Every adjacent pair of words differs by a single letter.
- Every `si` for `1 <= i <= k` is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- `sk == 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, s1, s2, ..., sk]`.

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"

```
class Solution {
```

```java
    public List<List<String>> findLadders(String beginWord, String endWord,
List<String> wordList) {
        List<List<String>> ans = new ArrayList<>();
        Map<String, Set<String>> reverse = new HashMap<>();
        Set<String> wordSet = new HashSet<>(wordList);
        wordSet.remove(beginWord);
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        Set<String> nextLevel = new HashSet<>();
        boolean findEnd = false;
        while (!queue.isEmpty()) {
            String word = queue.remove();
            for (String next : wordSet) {
                if (isLadder(word, next)) {
                    Set<String> reverseLadders = reverse.computeIfAbsent(next, k ->
new HashSet<>());
                    reverseLadders.add(word);
                    if (endWord.equals(next)) {
                        findEnd = true;
                    }
                    nextLevel.add(next);
                }
            }
            if (queue.isEmpty()) {
                if (findEnd) break;
                queue.addAll(nextLevel);
                wordSet.removeAll(nextLevel);
                nextLevel.clear();
            }
        }
        if (!findEnd) return ans;
        Set<String> path = new LinkedHashSet<>();
        path.add(endWord);
        findPath(endWord, beginWord, reverse, ans, path);
        return ans;
    }


    private void findPath(String endWord, String beginWord, Map<String, Set<String>>
graph,
                                List<List<String>> ans, Set<String> path) {
        Set<String> next = graph.get(endWord);
        if (next == null) return;
        for (String word : next) {
            path.add(word);
            if (beginWord.equals(word)) {
                List<String> shortestPath = new ArrayList<>(path);
                Collections.reverse(shortestPath);
                ans.add(shortestPath);
            } else {
                findPath(word, beginWord, graph, ans, path);
```

```
            }
            path.remove(word);
        }
    }

    private boolean isLadder(String s, String t) {
        if (s.length() != t.length()) return false;
        int diffCount = 0;
        int n = s.length();
        for (int i = 0; i < n; i++) {
            if (s.charAt(i) != t.charAt(i)) diffCount++;
            if (diffCount > 1) return false;
        }
        return diffCount == 1;
    }
}
```

5.

207. Course Schedule

Solved

Medium | Topics | Companies | Hint

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.

```
class Solution {
    public boolean canFinish(int n, int[][] prerequisites) {
        List<Integer>[] adj = new List[n];
        int[] indegree = new int[n];
        List<Integer> ans = new ArrayList<>();

        for (int[] pair : prerequisites) {
            int course = pair[0];
            int prerequisite = pair[1];
            if (adj[prerequisite] == null) {
                adj[prerequisite] = new ArrayList<>();
            }
            adj[prerequisite].add(course);
```

```
            indegree[course]++;
        }

        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }

        while (!queue.isEmpty()) {
            int current = queue.poll();
            ans.add(current);

            if (adj[current] != null) {
                for (int next : adj[current]) {
                    indegree[next]--;
                    if (indegree[next] == 0) {
                        queue.offer(next);
                    }
                }
            }
        }

        return ans.size() == n;
    }
}
```