**LAVANYAA A AIML**

1. **Palindrome Linked List**

package dsa;

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

public class PalindromeLL {

public static boolean isPalindrome(ListNode head) {

if (head == null || head.next == null) return true;

ListNode slow = head, fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

}

ListNode secondHalf = reverse(slow);

ListNode firstHalf = head;

while (secondHalf != null) {

if (firstHalf.val != secondHalf.val) return false;

firstHalf = firstHalf.next;

secondHalf = secondHalf.next;

}

return true;

}

private static ListNode reverse(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode nextNode = head.next;

head.next = prev;

prev = head;

head = nextNode;

}

return prev;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(2);

head.next.next.next.next = new ListNode(1);

System.out.println(isPalindrome(head)); // Expected output: true

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Check Equal Arrays**

package dsa;

public class EqualArrays {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 4, 5};

int[] arr2 = {1, 2, 3, 4, 5};

if (arr1.length != arr2.length) {

System.out.println("Arrays are not equal");

return;

}

boolean areEqual = true;

for (int i = 0; i < arr1.length; i++) {

if (arr1[i] != arr2[i]) {

areEqual = false;

break;

}

}

if (areEqual) {

System.out.println("Arrays are equal");

} else {

System.out.println("Arrays are not equal");

}

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(1)

1. **Balanced Tree Check**

package dsa;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class HeightBalancedTree {

public static int checkBalance(TreeNode node) {

if (node == null) return 0; // height of an empty tree is 0

int leftHeight = checkBalance(node.left);

int rightHeight = checkBalance(node.right); if (leftHeight == -1 || rightHeight == -1) {

return -1;

}

if (Math.abs(leftHeight - rightHeight) > 1) {

return -1;

}

return Math.max(leftHeight, rightHeight) + 1;

}

public static boolean isBalanced(TreeNode root) {

return checkBalance(root) != -1; // If -1 is returned, the tree is not balanced

}

public static void main(String[] args) {

TreeNode root = new TreeNode(25);

root.left = new TreeNode(20);

root.right = new TreeNode(36);

root.left.left = new TreeNode(10);

root.left.left.left = new TreeNode(5);

root.left.right = new TreeNode(22);

root.left.left.right = new TreeNode(12);

root.right.left = new TreeNode(30);

root.right.left.left = new TreeNode(28);

root.right.right = new TreeNode(40);

System.out.println(isBalanced(root)); // Expected output: false

}

}

TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)

1. **Triplet Sum**

package dsa;

import java.util.\*;

public class TripletSum {

public static List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> result = new ArrayList<>();

Arrays.sort(nums);

for (int i = 0; i < nums.length - 2; i++) {

if (i > 0 && nums[i] == nums[i - 1]) {

continue;

}

int left = i + 1;

int right = nums.length - 1;

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (sum == 0) {

result.add(Arrays.asList(nums[i], nums[left], nums[right]));

while (left < right && nums[left] == nums[left + 1]) {

left++;

}

while (left < right && nums[right] == nums[right - 1]) {

right--;

}

left++;

right--;

} else if (sum < 0) {

left++;

} else {

right--;

}

}

}

return result;

}

public static void main(String[] args) {

int[] nums = {-1, 0, 1, 2, -1, 4};

System.out.println(threeSum(nums)); // Expected output: [[-1, -1, 2], [-1, 0, 1]]

}

}

TIME COMPLEXITY: O(n log n)

SPACE COMPLEXITY: O(n2)

1. **Floor in Sorted Array**

package dsa;

public class FloorArray {

public static int findFloor(int[] nums, int target) {

int left = 0;

int right = nums.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return right;

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5};

int target = 5;

System.out.println(findFloor(arr, target)); // Expected output: 4 (index of 5)

}

}

TIME COMPLEXITY: O(log n)

SPACE COMPLEXITY: O(1)

1. **0/1 Knapsack problem**

package dsa;

public class Knapsack {

static int knapsack(int[] weights, int[] values, int itemCount, int capacity) {

int[] dp = new int[capacity + 1];

for (int i = weights[0]; i <= capacity; i++) {

dp[i] = values[0];

}

for (int i = 1; i < itemCount; i++) {

for (int j = capacity; j >= weights[i]; j--) {

dp[j] = Math.max(dp[j], values[i] + dp[j - weights[i]]);

}

}

return dp[capacity];

}

public static void main(String[] args) {

int[] weights = {1, 2, 4, 5};

int[] values = {5, 4, 8, 6};

int capacity = 5;

int itemCount = weights.length;

System.out.println(knapsack(weights, values, itemCount, capacity));

}

}

TIME COMPLEXITY: O(n\*m)

SPACE COMPLEXITY: O(n)