DAY 2-DSA PRACTICE
RANJIT H
0 - 1 Knapsack Problem

You are given the weights and values of items, and you need to put these items in a knapsack of capacity capacity to achieve the maximum total value in the knapsack. Each item is available in only one quantity.

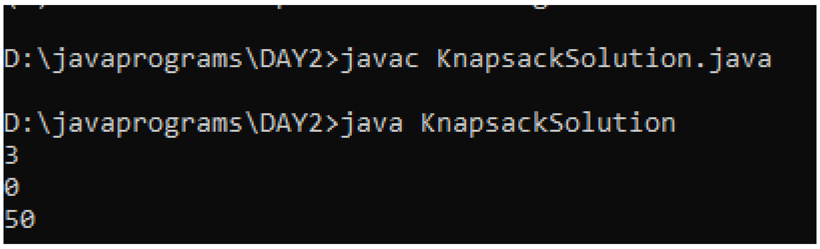
In other words, you are given two integer arrays **val[]** and **wt[]**, which represent the values and weights associated with items, respectively. You are also given an integer **capacity**, which represents the knapsack capacity. Your task is to find the maximum sum of values of a subset of val[] such that the sum of the weights of the corresponding subset is less than or equal to **capacity**. You cannot break an item; you must either pick the entire item or leave it (0-1 property).

Examples :

Input: capacity = 4, val[] = [1, 2, 3], wt[] = [4, 5, 1]
Output: 3
Explanation: Choose the last item, which weighs 1 unit and has a value of 3.
Input: capacity = 3, val[] = [1, 2, 3], wt[] = [4, 5, 6]
Output: 0
Explanation: Every item has a weight exceeding the knapsack's capacity (3).
Input: capacity = 5, val[] = [10, 40, 30, 50], wt[] = [5, 4, 6, 3]
Output: 50
Explanation: Choose the second item (value 40, weight 4) and the fourth item (value 50, weight 3) for a total weight of 7, which exceeds the capacity. Instead, pick the last item (value 50, weight 3) for a total value of 50.

CODE:
```
class KnapsackSolution {

    static int knapSack(int capacity, int values[], int weights[]) {

        int itemCount = values.length;

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


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

            previousRow[i] = values[0];
        }

        for (int i = 1; i < itemCount; i++) {
            int[] currentRow = new int[capacity + 1];
            for (int currentCapacity = 0; currentCapacity <= capacity; currentCapacity++)
{
                int includeItem = Integer.MIN_VALUE;
                if (weights[i] <= currentCapacity) {
                    includeItem = values[i] + previousRow[currentCapacity - weights[i]];
                }
                int excludeItem = previousRow[currentCapacity];
                currentRow[currentCapacity] = Math.max(includeItem, excludeItem);
            }
            previousRow = currentRow;
        }
        return previousRow[capacity];
    }

    public static void main(String[] args) {
        int capacity1 = 4;
        int[] values1 = {1, 2, 3};
        int[] weights1 = {4, 5, 1};
        System.out.println(knapSack(capacity1, values1, weights1));

        int capacity2 = 3;
        int[] values2 = {1, 2, 3};
        int[] weights2 = {4, 5, 6};
        System.out.println(knapSack(capacity2, values2, weights2));


        int capacity3 = 5;

        int[] values3 = {10, 40, 30, 50};

        int[] weights3 = {5, 4, 6, 3};

        System.out.println(knapSack(capacity3, values3, weights3));

    }

}
```

OUTPUT:



```
D:\javaprograms\DAY2>javac KnapsackSolution.java

D:\javaprograms\DAY2>java KnapsackSolution
3
0
50
```

TIME COMPLEXITY:O(n*capacity)

2. Floor in a Sorted Array

Given a sorted array **arr[]** (with unique elements) and an integer **k**, find the index (0-based) of the largest element in arr[] that is less than or equal to k. This element is called the "floor" of k. If such an element does not exist, return -1.

Examples

Input: arr[] = [1, 2, 8, 10, 11, 12, 19], k = 0
Output: -1
Explanation: No element less than 0 is found. So output is -1.
Input: arr[] = [1, 2, 8, 10, 11, 12, 19], k = 5
Output: 1
Explanation: Largest Number less than 5 is 2 , whose index is 1.

Input: arr[] = [1, 2, 8], k = 1
Output: 0
Explanation: Largest Number less than or equal to 1 is 1 , whose index is 0.

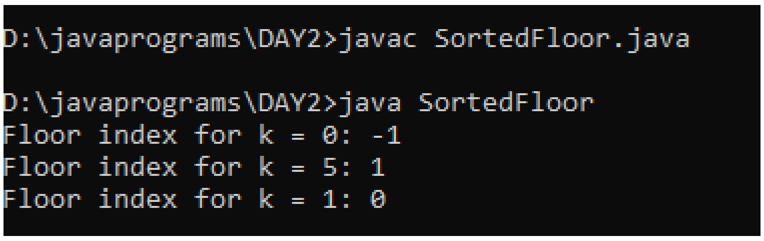
CODE:

```
class  SortedFloor{

    // Method to find the floor of k in a sorted array

    static int findFloor(int[] arr, int k) {

        int n = arr.length;

        for (int i = n - 1; i >= 0; i--) {

            if (arr[i] <= k) {

                return i;

            }

        }

        return -1;

    }


    // Main method to test the findFloor method

    public static void main(String[] args) {

        // Test cases

        int[] arr1 = {1, 2, 8, 10, 11, 12, 19};

        int k1 = 0;

        System.out.println("Floor index for k = " + k1 + ": " + findFloor(arr1, k1)); //
Output: -1


        int[] arr2 = {1, 2, 8, 10, 11, 12, 19};

        int k2 = 5;

        System.out.println("Floor index for k = " + k2 + ": " + findFloor(arr2, k2)); //
Output: 1


        int[] arr3 = {1, 2, 8};

        int k3 = 1;
```

        System.out.println("Floor index for k = " + k3 + ": " + findFloor(arr3, k3)); // Output: 0

    }

}

OUTPUT: 

```
D:\javaprograms\DAY2>javac SortedFloor.java

D:\javaprograms\DAY2>java SortedFloor
Floor index for k = 0: -1
Floor index for k = 5: 1
Floor index for k = 1: 0
```

TIME COMPLEXITY:O(N)

3. Check Equal Arrays

Given two arrays **arr1** and **arr2** of equal size, the task is to find whether the given arrays are equal. Two arrays are said to be equal if both contain the same set of elements, arrangements (or permutations) of elements may be different though.
Note: If there are repetitions, then counts of repeated elements must also be the same for two arrays to be equal.

Examples:

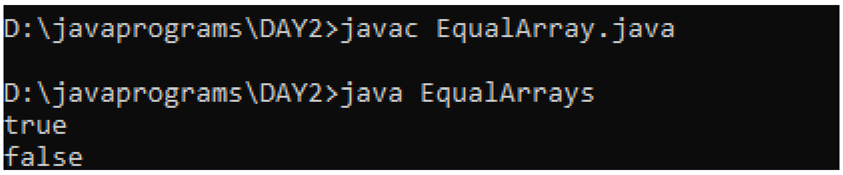
Input: arr1[] = [1, 2, 5, 4, 0], arr2[] = [2, 4, 5, 0, 1]
Output: true
Explanation: Both the array can be rearranged to [0,1,2,4,5]
Input: arr1[] = [1, 2, 5], arr2[] = [2, 4, 15]
Output: false
Explanation: arr1[] and arr2[] have only one common value.

CODE:

```
import java.util.HashMap;


class Solution {

    public static boolean check(int[] arr1, int[] arr2) {

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

```
        return false;
    }
    HashMap<Integer, Integer> frequencyMap = new HashMap<>();
    for (int i : arr1) {
        frequencyMap.put(i, frequencyMap.getOrDefault(i, 0) + 1);
    }
    for (int j : arr2) {
        if (!frequencyMap.containsKey(j) || frequencyMap.get(j) == 0) {
            return false;
        }
        frequencyMap.put(j, frequencyMap.get(j) - 1);
    }
    return true;
}

public static void main(String[] args) {
    int[] arr1 = {1, 2, 3, 4};
    int[] arr2 = {4, 3, 2, 1};
    System.out.println(check(arr1, arr2)); // Output: true

    int[] arr3 = {1, 2, 2, 3};
    int[] arr4 = {1, 2, 3, 3};
    System.out.println(check(arr3, arr4)); // Output: false
}
```

}

OUTPUT: 

```
D:\javaprograms\DAY2>javac EqualArray.java

D:\javaprograms\DAY2>java EqualArrays
true
false
```

TIMECOMPLEXITY:O(N)

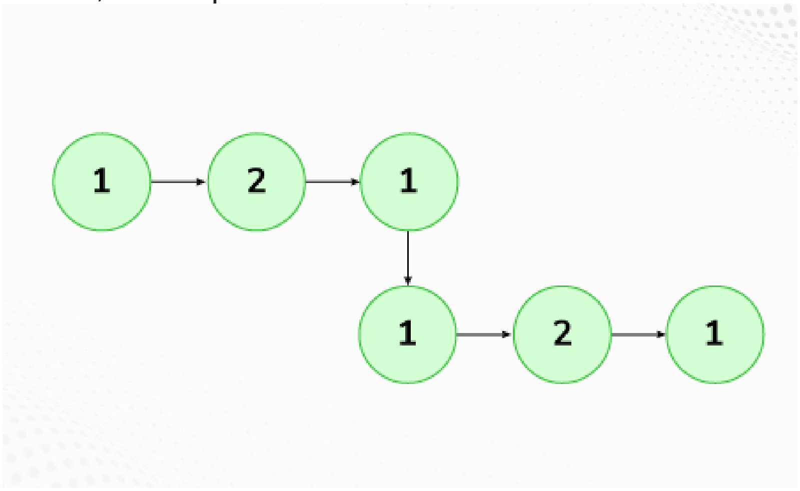
4. Palindrome Linked List

Given a singly linked list of integers. The task is to check if the given linked list is palindrome or not. **Examples:**

Input: LinkedList: 1->2->1->1->2->1

Output: true

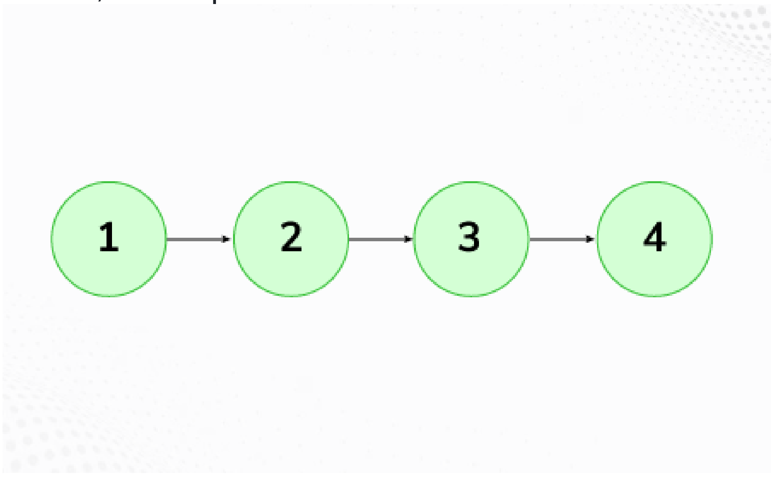
Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.



Input: LinkedList: 1->2->3->4

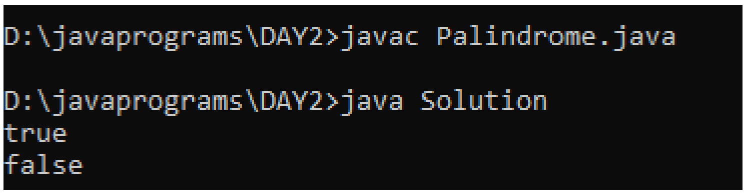
Output: false

Explanation: The given linked list is 1->2->3->4, which is not a palindrome and Hence, the output is false.



CODE:

```
class Node {
    int data;
    Node next;

    Node(int d) {
        data = d;
        next = null;
    }
}

class Solution {
    boolean isPalindrome(Node head) {
        if (head == null || head.next == null) {
            return true;
        }

        Node slow = head;
        Node fast = head;
        Node prev = null, temp;

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

        prev = slow;
        slow = slow.next;
        prev.next = null;

        while (slow != null) {
            temp = slow.next;
            slow.next = prev;
            prev = slow;
            slow = temp;
        }

        fast = head;
        slow = prev;

        while (slow != null) {
            if (fast.data != slow.data) return false;
            fast = fast.next;
            slow = slow.next;
        }

        return true;
    }

    public static void main(String[] args) {
        Node head1 = new Node(1);
        head1.next = new Node(2);
        head1.next.next = new Node(1);
        head1.next.next.next = new Node(1);
        head1.next.next.next.next = new Node(2);
        head1.next.next.next.next.next = new Node(1);
        Solution solution1 = new Solution();
        System.out.println(solution1.isPalindrome(head1));

        Node head2 = new Node(1);
        head2.next = new Node(2);
        head2.next.next = new Node(3);
        head2.next.next.next = new Node(4);

        Solution solution2 = new Solution();
        System.out.println(solution2.isPalindrome(head2));
    }
}
```

OUTPUT: 

```
D:\javaprograms\DAY2>javac Palindrome.java

D:\javaprograms\DAY2>java Solution
true
false
```

TIME COMPLEXITY:O(N)

5.BALANCED TREE CHECK

Given a binary tree, find if it is height balanced or not. A tree is height balanced if difference between heights of left and right subtrees is **not more than one** for all nodes of tree.

Examples:

Input:

```
      1
    /
   2
    \
     3
```

Output: 0

Explanation: The max difference in height of left subtree and right subtree is 2, which is greater than 1. Hence unbalanced

Input:

```
       10
```

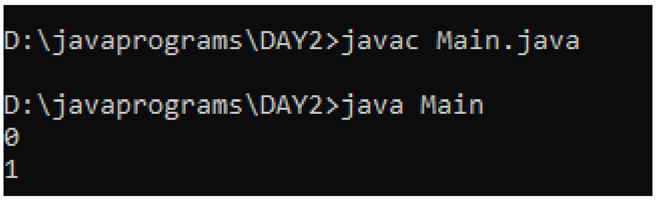
```
     /   \
    20   30
  /   \
 40   60
```

Output: 1

Explanation: The max difference in height of left subtree and right subtree is 1. Hence balanced.

```
class Node {
    int data;
    Node left, right;

    // Constructor
    Node(int value) {
        data = value;
        left = right = null;
    }
}

class Tree {
    // Helper function to check height and balance.
    int checkHeight(Node node) {
        if (node == null) {
            return 0; // Height of an empty tree is 0
        }

        int leftHeight = checkHeight(node.left);
        if (leftHeight == -1) {
            return -1; // Left subtree is not balanced
        }

        int rightHeight = checkHeight(node.right);
        if (rightHeight == -1) {
            return -1; // Right subtree is not balanced
        }
        if (Math.abs(leftHeight - rightHeight) > 1) {
            return -1; // Current node is not balanced
        }

        return 1 + Math.max(leftHeight, rightHeight); // Return height
    }

    // Main function to check if tree is balanced
    boolean isBalanced(Node root) {
        return checkHeight(root) != -1; // If -1 is returned, the tree is not balanced
    }
}

public class Main {
    public static void main(String[] args) {
        // Example 1: Unbalanced tree
        Node root1 = new Node(1);
        root1.left = new Node(2);
        root1.left.right = new Node(3);

        Tree tree1 = new Tree();
        System.out.println(tree1.isBalanced(root1) ? 1 : 0); // Output: 0

        // Example 2: Balanced tree
        Node root2 = new Node(10);
        root2.left = new Node(20);
        root2.right = new Node(30);
        root2.left.left = new Node(40);
        root2.left.right = new Node(60);

        Tree tree2 = new Tree();
        System.out.println(tree2.isBalanced(root2) ? 1 : 0); // Output: 1
    }
```

}

OUTPUT: 

```
D:\javaprograms\DAY2>javac Main.java

D:\javaprograms\DAY2>java Main
0
1
```

TIME COMPLEXITY:O(N)

6:Tripet Sum of Array

Given an array arr of size **n** and an integer **x**. Find if there's a triplet in the array which sums up to the given integer **x**.

Examples

Input:n = 6, x = 13, arr[] = [1,4,45,6,10,8]

Output: 1

Explanation: The triplet {1, 4, 8} in the array sums up to 13.

Input: n = 6, x = 10, arr[] = [1,2,4,3,6,7]

Output: 1

Explanation: Triplets {1,3,6} & {1,2,7} in the array sum to 10.

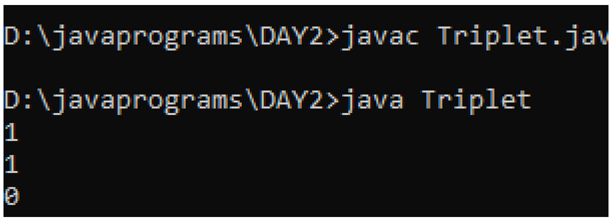
Input: n = 6, x = 24, arr[] = [40,20,10,3,6,7]

Output: 0

Explanation: There is no triplet with sum 24.

CODE: import java.util.Arrays;

```
class Solution {

    // Function to find if there is a triplet with a given sum

    public static boolean find3Numbers(int arr[], int n, int x) {

        if (n < 3) return false;


        Arrays.sort(arr);

        for (int i = 0; i < n - 2; i++) {
            int j = i + 1;
            int k = n - 1;
            while (j < k) {
                int sum = arr[i] + arr[j] + arr[k];
                if (sum == x) {
                    return true;
                } else if (sum < x) {
                    j++;
                } else {
                    k--;
                }
            }
        }
        return false;
    }
}

public class Triplet {
    public static void main(String[] args) {
        int arr1[] = {1, 4, 45, 6, 10, 8};
        int x1 = 13;
        System.out.println(Solution.find3Numbers(arr1, arr1.length, x1) ? 1 : 0);

        int arr2[] = {1, 2, 4, 3, 6, 7};
        int x2 = 10;
        System.out.println(Solution.find3Numbers(arr2, arr2.length, x2) ? 1 : 0);
        int arr3[] = {40, 20, 10, 3, 6, 7};

        int x3 = 24;

        System.out.println(Solution.find3Numbers(arr3, arr3.length, x3) ? 1 : 0);

    }

}
```

OUTPUT: 

```
D:\javaprograms\DAY2>javac Triplet.jav

D:\javaprograms\DAY2>java Triplet
1
1
0
```

TIME COMPLEXITY:O(N^2)