**Question 1**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class ContainerAllocator {

public static void main(String[] args) {

int[] shipments = {10, 20, 30};

int[] containerLimits = {15, 15, 20, 10};

int result = minContainers(shipments, containerLimits);

System.out.println("Minimum number of containers needed: " + result);

}

public static int minContainers(int[] shipments, int[] containerLimits) {

Arrays.sort(shipments);

reverse(shipments);

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

for (int shipment : shipments) {

boolean fitted = false;

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

if (shipment <= containerLimits[i]) {

containerLimits[i] -= shipment;

fitted = true;

break;

}

}

if (!fitted) {

containers.add(shipment);

}

}

for (int limit : containerLimits) {

if (limit < 0) {

return -1;

}

containers.add(limit);

}

return containers.size();

}

private static void reverse(int[] arr) {

int left = 0;

int right = arr.length - 1;

while (left < right) {

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

}

**Output:-** Minimum number of containers needed: 5

* **Explanation of code:-**

1. Sorting Shipments: We start by sorting the shipments array in descending order. This ensures that we process larger shipments first, which helps optimize container allocation.
2. Initializing Containers: We create an empty list called containers to keep track of the containers we’ll use.
3. Processing Shipments:

For each shipment in the sorted shipments array:

* We iterate through the available containers.
* If the shipment can fit into a container (i.e., its size is less than or equal to the container’s limit), we subtract the shipment size from that container’s limit.
* If no container can accommodate the entire shipment, we create a new container and add the shipment size to it.

1. Adding Remaining Containers:

After processing all shipments, we add any remaining container limits to the containers list. These are the containers that were not used for any shipment.

1. Checking Insufficient Capacity:

We check if any container has a negative limit (i.e., insufficient capacity). If so, we return -1 to indicate that the problem cannot be solved with the given container limits.

1. Returning the Result:

Finally, we return the total number of containers used (which is the size of the containers list).

The reverse method is a helper function that reverses an array in-place. It’s used to sort the shipments array in descending order.

**Question 2**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public class LinkedListCycleDetector {

public boolean hasCycle(ListNode head) {

if (head == null || head.next == null) {

return false; // No cycle if the list is empty or has only one node

}

ListNode tortoise = head;

ListNode hare = head;

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

tortoise = tortoise.next; // Move tortoise one step

hare = hare.next.next; // Move hare two steps

if (tortoise == hare) {

return true; // Cycle detected

}

}

return false; // No cycle found

}

public static void main(String[] args) {

// Example usage

ListNode head1 = new ListNode(20);

head1.next = new ListNode(30);

head1.next.next = new ListNode(40);

head1.next.next.next = new ListNode(60);

head1.next.next.next.next = new ListNode(80);

head1.next.next.next.next.next = head1.next.next; // Create a cycle

LinkedListCycleDetector detector = new LinkedListCycleDetector();

System.out.println("Example 1: " + detector.hasCycle(head1)); // Output: true

ListNode head2 = new ListNode(6);

head2.next = new ListNode(4);

head2.next.next = new ListNode(2);

head2.next.next.next = new ListNode(8);

System.out.println("Example 2: " + detector.hasCycle(head2)); // Output: false

}

}

**Output:-**

* **Explanation of code:-**

1. We initialize two pointers (tortoise and hare) at the head of the linked list.
2. The tortoise moves one step at a time, while the hare moves two steps.
3. If there is a cycle, the hare will eventually catch up to the tortoise.
4. If the hare reaches the end of the list (i.e., hare.next is null), there is no cycle.