1.0-1 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.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int knapSack(int ind, int weight, int cap, vector<int>& profit, vector<int>& weights, vector<vector<int>>& memo) {
    if (ind >= profit.size() || weight > cap) return 0;
    if (memo[ind][weight] != -1) return memo[ind][weight];
    int take = 0;
    if (weight + weights[ind] <= cap) {
        take = profit[ind] + knapSack(ind + 1, weight + weights[ind], cap, profit, weights, memo);
    }
    int not_take = knapSack(ind + 1, weight, cap, profit, weights, memo);
    return memo[ind][weight] = max(take, not_take);
}

int main() {
    int tc;
    cin >> tc;
    while (tc--) {
        int n;
        cin >> n;
        vector<int> profit(n), weights(n);
        for (int i = 0; i < n; i++) cin >> profit[i];
        for (int i = 0; i < n; i++) cin >> weights[i];
        int capacity;
        cin >> capacity;
        vector<vector<int>> memo(n, vector<int>(capacity + 1, -1));
        cout << knapSack(0, 0, capacity, profit, weights, memo) << endl;
    }
    return 0;
}
```

```
~/Desktop/practice (1.241s)
g++ knapsack.cpp

~/Desktop/practice (0.046s)
./a.out < knapsacktc.txt
3
0
50
```

Time Complexity: O(n×capacity)

Space Complexity: O(n×capacity) for memorization plus O(n) for the recursive stack.

2.Floor in 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.

```cpp
#include<bits/stdc++.h>
using namespace std;

void FloorInSortedArray(vector<int>&arr, int x){
    int n = arr.size();
    int left =0,right = n-1;
    int ans =0,ind=0;
    while(left<=right){
        int mid = left+(right-left)/2;
        if(arr[mid] == x){
            ans = arr[mid];
            ind = mid;
            break;
        }
        else if(arr[mid]<x){
            left = mid+1;
        }
        else{
            right = mid-1;
        }
    }
    cout<<"Floor of "<<x<<" is "<< ans<<endl;
}
```

```
~/Desktop/practice (3.541s)
g++ floorinsortedarray.cpp

~/Desktop/practice (0.045s)
./a.out <Floorinsortedarray.txt
Floor of 0 is 0
Floor of 5 is 0
Floor of 1 is 1
```

Time Complexity : O(log n)

Space Complexity : O(1)

a3. 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.

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.

```cpp
#include<bits/stdc++.h>
using namespace std;

bool areArraysEqual(vector<int>& arr1, vector<int>& arr2) {
    if (arr1.size() != arr2.size()) return false;
    unordered_map<int, int> countMap;
    for (int num : arr1) countMap[num]++;
    for (int num : arr2) {
        if (countMap.find(num) == countMap.end() || countMap[num] == 0) return false;
        countMap[num]--;
    }
    for (auto entry : countMap) {
        if (entry.second != 0) return false;
    }
    return true;
}

int main() {
    int tc;
    cin >> tc;
    while (tc--) {
        int n;
        cin >> n;
        vector<int> arr1(n), arr2(n);
        for (int i = 0; i < n; i++) cin >> arr1[i];
        for (int i = 0; i < n; i++) cin >> arr2[i];
        cout << (areArraysEqual(arr1, arr2) ? "true" : "false") << endl;
    }
    return 0;
}
```

```
~/Desktop/practice (3.8s)
g++ CheckEqualArrays.cpp

~/Desktop/practice (0.045s)
./a.out < checkequalarrays.txt
true
false
```

Time Complexity: O(n)

Space Complexity : O(n)

4. Palindrome Linked List

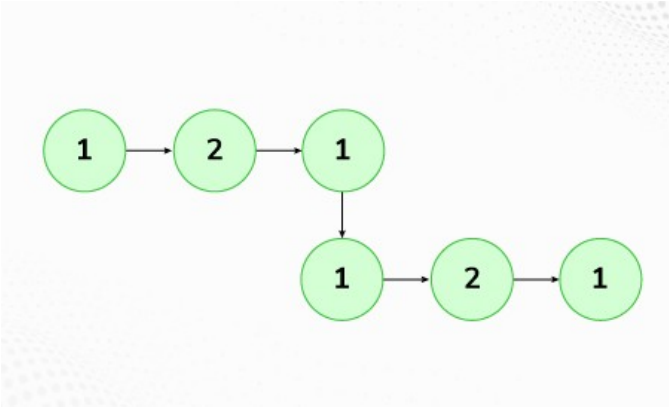
Given a singly linked list of integers. The task is to check if the given linked list is palindrome or not.

Examples:

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

Output: true

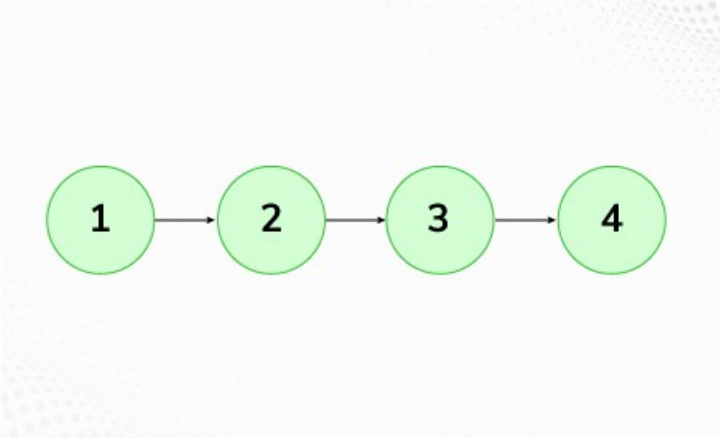
Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.



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.



as

```cpp
#include <iostream>
using namespace std;

struct ListNode {
    int data;
    ListNode* next;
    ListNode(int x) : data(x), next(nullptr) {}
};

bool isPalindrome(ListNode* head) {
    if (!head || !head->next) return true;
    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    ListNode* prev = nullptr;
    while (slow) {
        ListNode* nextNode = slow->next;
        slow->next = prev;
        prev = slow;
        slow = nextNode;
    }
    ListNode* left = head;
    ListNode* right = prev;
    while (right) {
        if (left->data != right->data) return false;
        left = left->next;
        right = right->next;
    }
    return true;
}
```

```
~/Desktop/practice (0.693s)
g++ PalindromeLinkedList.cpp

~/Desktop/practice (0.038s)
./a.out < palindromeLinkedList.txt
true
false
```

Time Complexity: O(n)

Space Complexity : O(1)

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.

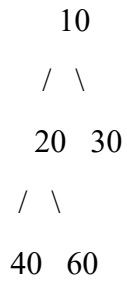
Examples:

Input:

Output: 0

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

Input:



Output: 1

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

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

pair<int, bool> checkHeightBalance(TreeNode* node) {
    if (!node) return {0, true};

    auto left = checkHeightBalance(node->left);
    auto right = checkHeightBalance(node->right);

    int height = 1 + max(left.first, right.first);
    bool balanced = left.second && right.second && abs(left.first - right.first) <= 1;

    return {height, balanced};
}

bool isBalanced(TreeNode* root) {
    return checkHeightBalance(root).second;
}
```

```
~/Desktop/practice (0.744s)
g++ BalancedTree.cpp

~/Desktop/practice (0.037s)
./a.out
0
1
```

Time Complexity : O(n);

Space Complexity : **Best case (balanced tree)**: O(logN) space complexity.

Worst case (unbalanced tree): O(N) space complexity.

6.Triplet Sum in Array

Given an array arr of size n and an integern 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.

```cpp
#include <bits/stdc++.h>
using namespace std;
void threeSum(vector<int>& nums, int target) {
    int len = nums.size();
    sort(nums.begin(), nums.end());
    bool found = false;
    for (int i = 0; i < len - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;  // Skip duplicates for the first e
        int j = i + 1;
        int k = len - 1;
        while (j < k) {
            int sum = nums[i] + nums[j] + nums[k];
            if (sum == target) {
                cout << nums[i] << " " << nums[j] << " " << nums[k] << endl;
                found = true;
                // Skip duplicates for the second and third elements
                while (j < k && nums[j] == nums[j + 1]) j++;
                while (j < k && nums[k] == nums[k - 1]) k--;
                j++;
                k--;
            } else if (sum < target) {
                j++;
            } else {
                k--;
            }
        }
    }
    if (!found) {
        cout << 0 << endl;  // No triplets found
    }
}
```

```
~/Desktop/practice (2.788s)
g++ TripletSumInArray.cpp

~/Desktop/practice (0.038s)
./a.out < triplet.txt
1 4 8
1 2 7
1 3 6
0
```

Time Complexity : O(n^2) . Space Complexity : O(1)