* 1. **Knapsack problem**

1)Approach🡪greedy

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.

Program:

import java.util.Arrays;

class item{

public double ratio;

int value, weight;

item(int value,int weight){

this.value=value;

this.weight=weight;

this.ratio=(double)value/weight;

}

}

public class Knapsack {

public static int maxvalue(int[] value, int[] weight, int capacity) {

int n = weight.length;

item[] items = new item[n];

for (int i = 0; i < n; i++) {

items[i] = new item(value[i],weight[i]);

}

Arrays.sort(items,(a,b)->Double.compare(b.ratio,a.ratio));

int currweight=0;

double finalvalue=0.0;

for(int i=0;i<n;i++){

if(currweight+ weight[i] <=capacity){

currweight+=weight[i]+1;

finalvalue = finalvalue + value[i];

}

else{

int rem=capacity- weight[i];

finalvalue+=(value[i] / weight[i])\*rem;

}

}

return (int) finalvalue;

}

public static void main(String[] args){

int[] value = {60, 100, 120};

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

int capacity = 50;

System.out.println("Maximum value is " + Knapsack.maxvalue(value, weight, capacity));

int[] value1 = {30, 90, 40};

int[] weight1 = {10, 10, 30};

int capacity1 = 40;

System.out.println("Maximum value is " + Knapsack.maxvalue(value1, weight1, capacity1));

int[] value2 = {10, 40, 30, 50};

int[] weight2 = {5, 4, 6, 3};

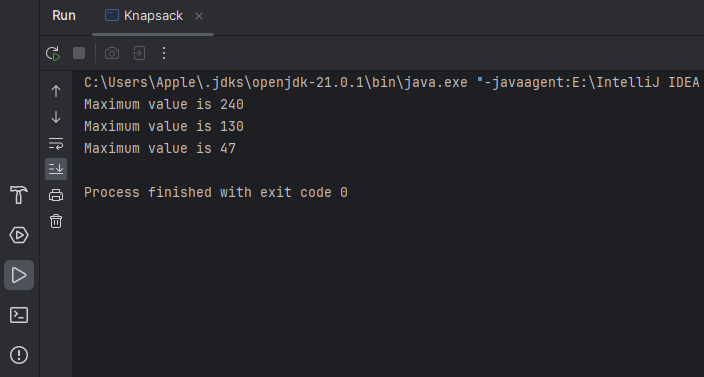
int capacity2 = 5;

System.out.println("Maximum value is " + Knapsack.maxvalue(value2,weight2,capacity2));

}

}

Output:



 **Time Complexity**: O(n log\*n)

 **Space Complexity**: O(n)

**2)Floor in Sorted Array:**

Given a sorted array and a value **x**, the floor of x is the largest element in the array smaller than or equal to x. Write efficient functions to find the floor of x

**Examples:**

***Input:*** *arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 5****Output:*** *2****Explanation:*** *2 is the largest element in   
arr[] smaller than 5*

Program:

public class FloorinSortedArray {

public static void main(String[] args){

FloorinSortedArray obj=new FloorinSortedArray();

int arr[] = { 1, 2, 4, 6, 10, 12, 14 };

int n = arr.length;

int x = 7;

int index = obj.floor(arr,x);

System.out.println(index);

}

public int floor(int[] arr,int target){

int start=0;

int end=arr.length;

while(start<=end){

int mid=start+(end-start)/2;

if(arr[mid]<target){

start=mid+1;

}

else if(arr[mid]>target){

end=mid-1;

}

else{

return mid;

}

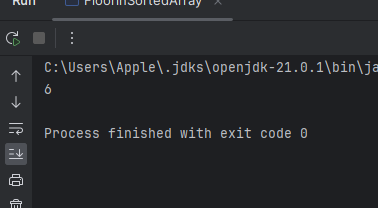
}

return arr[end];

}

}

Output:



 **Time Complexity**: O(log n)

 **Space Complexity**: O(1)

3)Check if two Arrays are equals or not

Given two arrays, **arr1**and **arr2**of equal length**N**, the task is to determine if the given arrays are equal or not. Two arrays are considered equal if:

* Both arrays contain the same set of elements.
* The arrangements (or permutations) of elements may be different.
* If there are repeated elements, the counts of each element must be the same in both arrays.

**Examples:**

***Input:*** *arr1[] = {1, 2, 5, 4, 0}, arr2[] = {2, 4, 5, 0, 1}****Output:*** *Yes*

***Input:*** *arr1[] = {1, 2, 5, 4, 0, 2, 1}, arr2[] = {2, 4, 5, 0, 1, 1, 2}****Output:*** *Yes*

Program:

import java.util.Arrays;

public class CheckequalArrays {

public static void main(String[] args){

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

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

CheckequalArrays obj=new CheckequalArrays();

System.out.println(obj.check(arr1,arr2));

}

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

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

return false;

}

Arrays.sort(arr1);

Arrays.sort(arr2);

if(Arrays.equals(arr1,arr2)){

return true;

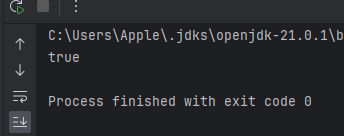
}

return false;

}

}

Output:



 **Time Complexity**: O(n log\*n)

 **Space Complexity**: O(1)

**4)Palindrome LinkedList**

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

**Examples:**

***Input:*** *head: 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:*** *head: 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.*

*Program:*

class Node{

int data;

Node next;

public Node(int data){

this.data=data;

}

}

public class PalindromeLinkedlist {

public Node reverse(Node head){

Node node=reverse(head.next);

Node front=head.next;

front.next=head;

head.next=null;

return head;

}

public boolean palindrome(Node head){

Node slow=head;

Node fast=head;

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

slow=slow.next;

fast=fast.next.next;

}

Node newhead=reverse(slow.next);

Node first=head;

Node second=newhead;

while(second!=null){

if(first.data!=second.data){

reverse(newhead);

return false;

}

first=first.next;

second=second.next;

}

reverse(newhead);

return true;

}

public static void main(String[] args){

Node head = new Node(1);

head.next = new Node(2);

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

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

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

PalindromeLinkedlist obj=new PalindromeLinkedlist();

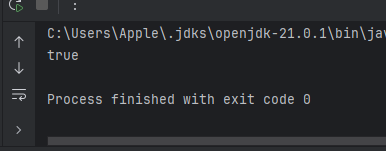
boolean result = obj.palindrome(head);

System.out.println(result);

}

}

Output:



 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

5)Balanced BinaryTree

Program:

class LNode{

int val;

LNode left;

LNode right;

public LNode(int val) {

this.val = val;

this.left = null;

this.right = null;

}

}

public class CheckBalancedBinaryTree {

LNode root;

public int height(LNode node){

if(node==null){

return 0;

}

else{

return 1+Math.max(height(node.left),height(node.right));

}

}

public boolean check(LNode node){

if(node==null){

return true;

}

int val = Math.abs(height(node.left) - height(node.right));

if(val<=1 && check(node.left) && check(node.right)){

return true;

}

return false;

}

public static void main(String[] args){

CheckBalancedBinaryTree tree = new CheckBalancedBinaryTree();

tree.root = new LNode(1);

tree.root.left = new LNode(2);

tree.root.right = new LNode(3);

tree.root.left.left = new LNode(4);

tree.root.left.right = new LNode(5);

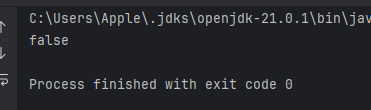
tree.root.left.left.left = new LNode(8);

System.out.println(tree.check(tree.root));

}

}

Output:



 **Time Complexity**: O( n\*2)

 **Space Complexity**: O(n)

**3 Sum – Triplet Sum in Array**

Program:

import java.util.Arrays;

public class triplesum {

public static void main(String[] args){

triplesum obj=new triplesum();

int[] arr = { 1, 4, 45, 6, 10, 8 };

int sum = 22;

int result= obj.check(arr,sum);

System.out.println(result);

}

public int check(int[] arr,int sum) {

Arrays.sort(arr);

int n = arr.length;

for (int i = 0; i < n - 2; i++) {

int start = i + 1;

int end = n - 1;

while (start < end) {

int sum1 = arr[i] + arr[start] + arr[end];

if (sum1 == sum) {

return sum1;

} else if (sum1 > sum) {

end--;

} else {

start++;

}

}

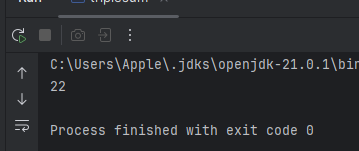
}

return -1;

}

}

Output:



 **Time Complexity**: O(n\*2)

 **Space Complexity**: O(1)