Day 2: Recursion, Stack, Queue

1. Rotate array by K elements – Block Swap Algorithm:

public class RotateArrayByK {

public static void reverse(int[] arr, int start, int end) {

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

public static void rotateArray(int[] arr, int k) {

int n = arr.length;

k = k % n; // Adjust k in case it's larger than array length

reverse(arr, 0, n - 1); // Reverse the whole array

reverse(arr, 0, k - 1); // Reverse the first part

reverse(arr, k, n - 1); // Reverse the second part

}

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5, 6, 7};

int k = 3;

System.out.println("Original array:");

for (int num : arr) {System.out.print(num + " ");}

rotateArray(arr, k);

System.out.println("\nArray rotated by " + k + " elements:");

for (int num : arr) {

System.out.print(num + " ");

}

}

}

2. Print all quadruplets with a given sum | 4 sum problem extended

Google Translate Icon

Given an unsorted integer array, print all distinct four elements tuple (quadruplets) in it, having a given sum.

For example,

Input:

A[] = [2, 7, 4, 0, 9, 5, 1, 3]

target = 20

Not showing quaraplets:

public static boolean hasQuadruplet(int[] nums, int n, int target, int count)

{

System.out.println(n+" "+target+" "+count);

// if the desired sum is reached with 4 elements, return true

if (target == 0 && count == 4) {

return true;

}

// return false if the sum is not possible with the current configuration

if (count > 4 || n == 0) {

return false;

}

// Recur with

// 1. Including the current element

// 2. Excluding the current element

return hasQuadruplet(nums, n - 1, target - nums[n - 1], count + 1) ||

hasQuadruplet(nums, n - 1, target, count);

}

public static void main(String[] args)

{

int[] nums = { 2, 7, 4, 0, 9, 5, 1, 3 };

int target = 20;

//quadTuplePrint(A, target);

if (hasQuadruplet(nums, nums.length, target, 0)) {

System.out.println("Quadruplet exists");

}

else {

System.out.println("Quadruplet Doesn't Exist");

}

}

showing quaraplets:

Output: Below are the quadruplets with sum 20

(0, 4, 7, 9)

(1, 3, 7, 9)

(2, 4, 5, 9)

class Main

{

// Function to print all quadruplet present in an array with a given sum

public static void quadTuple(int[] A, int target)

{

// sort the array in ascending order

Arrays.sort(A);

// check if quadruplet is formed by `A[i]`, `A[j]`, and a pair from

// subarray `A[j+1…n)`

for (int i = 0; i <= A.length - 4; i++)

{

for (int j = i + 1; j <= A.length - 3; j++)

{

// `k` stores remaining sum

int k = target - (A[i] + A[j]);

// check for sum `k` in subarray `A[j+1…n)`

int low = j + 1, high = A.length - 1;

while (low < high)

{

if (A[low] + A[high] < k) {

low++;

}

else if (A[low] + A[high] > k) {high--;}

// quadruplet with a given sum found

else {

System.out.println("(" + A[i] + " " + A[j] + " " +

A[low] + " " + A[high] + ")");

low++;high--;

}

}

}

}

}

public static void main(String[] args)

{

int[] A = { 2, 7, 4, 0, 9, 5, 1, 3 };

int target = 20;

quadTuple(A, target);

}

}

3. Find the smallest missing element from a sorted array

Given a sorted array of non-negative distinct integers, find the smallest missing non-negative element in it.

For example,

Input: nums[] = [0, 1, 2, 6, 9, 11, 15]

Output: The smallest missing element is 3

Input: nums[] = [1, 2, 3, 4, 6, 9, 11, 15]

Output: The smallest missing element is 0

Input: nums[] = [0, 1, 2, 3, 4, 5, 6]

Output: The smallest missing element is 7

public static int findSmallestMissing(int[] nums, int left, int right)

{

// base condition

if (left > right) {

return left;

}

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

// if the mid-index matches with its value, then the mismatch

// lies on the right half

if (nums[mid] == mid) {

return findSmallestMissing(nums, mid + 1, right);

}

else {

// mismatch lies on the left half

return findSmallestMissing(nums, left, mid - 1);

}

}

public static void main(String[] args)

{

// int[] nums = { 0, 1, 2, 3, 4, 5, 6 };

// int[] nums = { 1, 2, 3, 4, 6, 9, 11, 15 };

int[] nums = { 0, 1, 2, 6, 9, 11, 15 };

int left = 0, right = nums.length - 1;

System.out.println("The smallest missing element is "

+ findSmallestMissing(nums, left, right));

}

4. Longest Increasing Subsequence using Dynamic Programming

Google Translate Icon

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence’s elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous or unique.

Please note that the problem specifically targets subsequences that need not be contiguous, i.e., subsequences are not required to occupy consecutive positions within the original sequences.

For example, the longest increasing subsequence of [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15] is [0, 2, 6, 9, 11, 15].

This subsequence has length 6; the input sequence has no 7–member increasing subsequences. The longest increasing subsequence in this example is not unique.

For instance, [0, 4, 6, 9, 11, 15] or [0, 4, 6, 9, 13, 15] are other increasing subsequences of equal length in the same input sequence.

class Main

{

// Function to find the length of the longest increasing subsequence

public static int LIS(int[] arr, int i, int n, int prev)

{

// Base case: nothing is remaining

if (i == n) {

return 0;

}

// case 1: exclude the current element and process the

// remaining elements

int excl = LIS(arr, i + 1, n, prev);

// case 2: include the current element if it is greater

// than the previous element in LIS

int incl = 0;

if (arr[i] > prev) {

incl = 1 + LIS(arr, i + 1, n, arr[i]);

}

// return the maximum of the above two choices

return Integer.max(incl, excl);

}

public static void main(String[] args)

{

// int[] arr = { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 };// 6

int[] arr = { 0, 8, 4, 12, 2, 10, 6, 14, 9, 5, 13, 3, 11, 7 };// 5

System.out.print("The length of the LIS is "

+ LIS(arr, 0, arr.length, Integer.MIN\_VALUE));

}

}

5. Recursive solution to sort a stack

Google Translate Icon

Given a stack, sort it using recursion. The use of any other data structures (like containers in STL or Collections in Java) is not allowed.

For example,

Stack before sorting : 5 | -2 | 9 | -7 | 3 where 3 is the top element

Stack after sorting : -7 | -2 | 3 | 5 | 9 where 9 is the top element

import java.util.Stack;

public class RecursiveStackSort {

public static void insertSorted(Stack<Integer> stack, int element) {

if (stack.isEmpty() || element > stack.peek()) {

stack.push(element);

} else {

int temp = stack.pop();

insertSorted(stack, element);

stack.push(temp);

}

}

public static void sortStack(Stack<Integer> stack) {

if (!stack.isEmpty()) {

int temp = stack.pop();sortStack(stack);insertSorted(stack, temp);

}

}

public static void main(String[] args) {

Stack<Integer> inputStack = new Stack<>();

inputStack.push(34);inputStack.push(3); inputStack.push(31); inputStack.push(98); inputStack.push(92);inputStack.push(23); System.out.println("Original stack: " + inputStack);

sortStack(inputStack);

System.out.println("Sorted stack: " + inputStack);

}

}

6. Merging Overlapping Intervals

Given a set of intervals, print all non-overlapping intervals after merging the overlapping intervals.

For example,

Input: {1, 5}, {2, 3}, {4, 6}, {7, 8}, {8, 10}, {12, 15}

Output: Intervals after merging overlapping intervals are {1, 6}, {7, 10}, {12, 15}.

class Interval {

int start;

int end;

@Override

public String toString() {

return "{" + start + ", " + end + "}";

}

public Interval(int start, int end) {

this.start = start;

this.end = end;

}

}

public static void mergeIntervalsUsingStackAndList(List<Interval> intervals)

{

// sort the intervals in increasing order of their starting time

Collections.sort(intervals, Comparator.comparingInt(a -> a.start));

// create an empty stack

Stack<Interval> stack = new Stack<>();

// do for each interval

for (Interval curr: intervals)

{

// if the stack is empty or the top interval in the stack does not overlap

// with the current interval, push it into the stack

if (stack.empty() || curr.start > stack.peek().end) {

stack.push(curr);

}

// if the top interval of the stack overlaps with the current interval,

// merge two intervals by updating the end of the top interval

// to the current interval

if (stack.peek().end < curr.end) {

stack.peek().end = curr.end;

}

}

// print all non-overlapping intervals

while (!stack.empty()) {

System.out.println(stack.pop());

}

}

public static void mergeIntervals(Interval[] intervals) {

if (intervals == null || intervals.length <= 1) {

return;

}

// Sort intervals based on start points

for (int i = 0; i < intervals.length - 1; i++) {

for (int j = 0; j < intervals.length - i - 1; j++) {

if (intervals[j].start > intervals[j + 1].start) {

Interval temp = intervals[j];

intervals[j] = intervals[j + 1];

intervals[j + 1] = temp;

}

}

}

// Merge overlapping intervals

Interval currentInterval = intervals[0];

for (int i = 1; i < intervals.length; i++) {

if (currentInterval.end >= intervals[i].start) {

currentInterval.end = Math.max(currentInterval.end, intervals[i].end);

} else {

System.out.print("{" + currentInterval.start + ", " + currentInterval.end + "} ");

currentInterval = intervals[i];

}

}

System.out.println("{" + currentInterval.start + ", " + currentInterval.end + "}");

}

public static void main(String[] args) {

Interval[] intervals = {

new Interval(1, 5), new Interval(2, 3),

new Interval(4, 6), new Interval(7, 8),

new Interval(8, 10), new Interval(12, 15)

};

mergeIntervals(intervals);

List<Interval> intervalsList = Arrays.asList(

new Interval(1, 5), new Interval(2, 3),

new Interval(4, 6), new Interval(7, 8),

new Interval(8, 10), new Interval(12, 15)

);

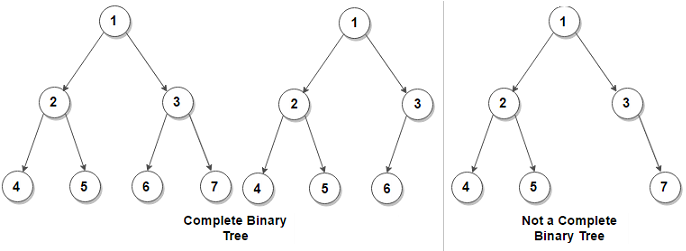
mergeIntervalsUsingStackAndList(intervalsList);

}

7. **Check if a binary tree is a complete binary tree or not**

Given a binary tree, check if it is a complete binary tree or not.

A complete binary tree is a binary tree in which every level, except possibly the last, is filled, and all nodes are as far left as possible. For example, the following binary trees are complete:



static class Node

{

int key;

Node left = null, right = null;

Node(int key) {

this.key = key;

}

}

public static void main(String[] args)

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ \ / \

4 5 6 7

\*/

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.right.left = new Node(6);

root.right.right = new Node(7);

if (isComplete(root, 0, size(root))) {

System.out.println("Complete binary tree");

}

else {

System.out.println("Not a complete binary tree");

}

}

private static int size(Node root)

{

if (root == null) {

return 0;

}

return 1 + size(root.left) + size(root.right);

}

// Recursive function to check if a given binary tree is a complete tree or not

public static boolean isComplete(Node root, int i, int n)

{

// return if the tree is empty

if (root == null) {

return true;

}

if ((root.left != null && 2\*i + 1 >= n) ||

!isComplete(root.left, 2\*i + 1, n)) {

return false;

}

if ((root.right != null && 2\*i + 2 >= n) ||

!isComplete(root.right, 2\*i + 2, n)) {

return false;

}

return true;

}

8. Maximum number of robots within the budget,You have n robots. You are given two 0-indexed integer arrays, chargeTimes and runningCosts, both of length n. The ith robot costs chargeTimes[i] units to charge and costs runningCosts[i] units to run. You are also given an integer budget.

The total cost of running k chosen robots is equal to max(chargeTimes) + k \* sum(runningCosts), where max(chargeTimes) is the largest charge cost among the k robots and sum(runningCosts) is the sum of running costs among the k robots.

Return the maximum number of consecutive robots you can run such that the total cost does not exceed budget.

Example 1:

Input: chargeTimes = [3,6,1,3,4], runningCosts = [2,1,3,4,5], budget = 25

Output: 3

Explanation:

It is possible to run all individual and consecutive pairs of robots within budget.

To obtain answer 3, consider the first 3 robots. The total cost will be max(3,6,1) + 3 \* sum(2,1,3) = 6 + 3 \* 6 = 24 which is less than 25.

It can be shown that it is not possible to run more than 3 consecutive robots within budget, so we return 3.

Example 2:

Input: chargeTimes = [11,12,19], runningCosts = [10,8,7], budget = 19

Output: 0

Explanation: No robot can be run that does not exceed the budget, so we return 0.

public static int maxConsecutiveRobots(int[] chargeTimes, int[] runningCosts, int budget) {

int n = chargeTimes.length;

int maxConsecutive = 0;

int totalCost = 0;

int windowStart = 0;

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

totalCost += chargeTimes[windowEnd] + (runningCosts[windowEnd] \* (windowEnd - windowStart + 1));

while (totalCost > budget) {

totalCost -= chargeTimes[windowStart] + (runningCosts[windowStart] \* (windowEnd - windowStart + 1));

windowStart++;

}

maxConsecutive = Math.max(maxConsecutive, windowEnd - windowStart + 1);

}

return maxConsecutive;

}

public static void main(String[] args) {

int[] chargeTimes1 = {3, 6, 1, 3, 4};

int[] runningCosts1 = {2, 1, 3, 4, 5};

int budget1 = 25;

System.out.println(maxConsecutiveRobots(chargeTimes1, runningCosts1, budget1));// Output: 3

int[] chargeTimes2 = {11, 12, 19};

int[] runningCosts2 = {10, 8, 7};

int budget2 = 19;

System.out.println(maxConsecutiveRobots(chargeTimes2, runningCosts2, budget2));// Output: 0

}

9. Jump game VI,

You are given a 0-indexed integer array nums and an integer k.

You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] inclusive.

You want to reach the last index of the array (index n - 1). Your score is the sum of all nums[j] for each index j you visited in the array.

Return the maximum score you can get.

Example 1:

Input: nums = [1,-1,-2,4,-7,3], k = 2

Output: 7

Explanation: You can choose your jumps forming the subsequence [1,-1,4,3] (underlined above). The sum is 7.

Example 2:

Input: nums = [10,-5,-2,4,0,3], k = 3

Output: 17

Explanation: You can choose your jumps forming the subsequence [10,4,3] (underlined above). The sum is 17.

Example 3:

Input: nums = [1,-5,-20,4,-1,3,-6,-3], k = 2

Output: 0

public static int maxResult(int[] nums, int k) {

int n = nums.length;

int[] dp = new int[n];

dp[0] = nums[0];

Deque<Integer> maxDeque = new ArrayDeque<>();

maxDeque.offer(0);

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

while (!maxDeque.isEmpty() && maxDeque.peekFirst() < i - k) {

maxDeque.pollFirst();

}

dp[i] = dp[maxDeque.peekFirst()] + nums[i];

while (!maxDeque.isEmpty() && dp[i] >= dp[maxDeque.peekLast()]) {

maxDeque.pollLast();

}

maxDeque.offerLast(i);

}

return dp[n - 1];

}

public static void main(String[] args) {

int[] nums1 = {1, -1, -2, 4, -7, 3};

int k1 = 2;

System.out.println(maxResult(nums1, k1)); // Output: 7

int[] nums2 = {10, -5, -2, 4, 0, 3};

int k2 = 3;

System.out.println(maxResult(nums2, k2)); // Output: 17

int[] nums3 = {1, -5, -20, 4, -1, 3, -6, -3};

int k3 = 2;

System.out.println(maxResult(nums3, k3)); // Output: 0

}