Sliding Window

Given an array of positive numbers and a positive number ‘k’, find the **maximum sum of any contiguous subarray of size ‘k’**.

class MaxSumSubArrayOfSizeK {

public static int findMaxSumSubArray(int k, int[] arr) {

int maxSum = 0, windowSum;

for (int i = 0; i <= arr.length - k; i++) {

windowSum = 0;

for (int j = i; j < i + k; j++) {

windowSum += arr[j];

}

maxSum = Math.max(maxSum, windowSum);

}

return maxSum;

}

public static void main(String[] args) {

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(3, new int[] { 2, 1, 5, 1, 3, 2 }));

System.out.println("Maximum sum of a subarray of size K: "

+ MaxSumSubArrayOfSizeK.findMaxSumSubArray(2, new int[] { 2, 3, 4, 1, 5 }));

}

}

Given a string, find the length of the **longest substring** in it **with no more than K distinct characters**.

import java.util.\*;

class LongestSubstringKDistinct {

public static int findLength(String str, int k) {

if (str == null || str.length() == 0 || str.length() < k)

throw new IllegalArgumentException();

int windowStart = 0, maxLength = 0;

Map<Character, Integer> charFrequencyMap = new HashMap<>();

// in the following loop we'll try to extend the range [windowStart, windowEnd]

for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {

char rightChar = str.charAt(windowEnd);

charFrequencyMap.put(rightChar, charFrequencyMap.getOrDefault(rightChar, 0) + 1);

// shrink the sliding window, until we are left with 'k' distinct characters in the frequency map

while (charFrequencyMap.size() > k) {

char leftChar = str.charAt(windowStart);

charFrequencyMap.put(leftChar, charFrequencyMap.get(leftChar) - 1);

if (charFrequencyMap.get(leftChar) == 0) {

charFrequencyMap.remove(leftChar);

}

windowStart++; // shrink the window

}

maxLength = Math.max(maxLength, windowEnd - windowStart + 1); // remember the maximum length so far

}

return maxLength;

}

public static void main(String[] args) {

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("araaci", 2));

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("araaci", 1));

System.out.println("Length of the longest substring: " + LongestSubstringKDistinct.findLength("cbbebi", 3));

}

}

Given an array of unsorted numbers, find all **unique triplets in it that add up to zero**.

import java.util.\*;

class TripletSumToZero {

public static List<List<Integer>> searchTriplets(int[] arr) {

Arrays.sort(arr);

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

for (int i = 0; i < arr.length - 2; i++) {

if (i > 0 && arr[i] == arr[i - 1]) // skip same element to avoid duplicate triplets

continue;

searchPair(arr, -arr[i], i + 1, triplets);

}

return triplets;

}

private static void searchPair(int[] arr, int targetSum, int left, List<List<Integer>> triplets) {

int right = arr.length - 1;

while (left < right) {

int currentSum = arr[left] + arr[right];

if (currentSum == targetSum) { // found the triplet

triplets.add(Arrays.asList(-targetSum, arr[left], arr[right]));

left++;

right--;

while (left < right && arr[left] == arr[left - 1])

left++; // skip same element to avoid duplicate triplets

while (left < right && arr[right] == arr[right + 1])

right--; // skip same element to avoid duplicate triplets

} else if (targetSum > currentSum)

left++; // we need a pair with a bigger sum

else

right--; // we need a pair with a smaller sum

}

}

public static void main(String[] args) {

System.out.println(TripletSumToZero.searchTriplets(new int[] { -3, 0, 1, 2, -1, 1, -2 }));

System.out.println(TripletSumToZero.searchTriplets(new int[] { -5, 2, -1, -2, 3 }));

}

}

# **Start of LinkedList Cycle (medium)**

class ListNode {

int value = 0;

ListNode next;

ListNode(int value) {

this.value = value;

}

}

class LinkedListCycleStart {

public static ListNode findCycleStart(ListNode head) {

// TODO: Write your code here

return head;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

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

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

head.next.next.next.next.next = new ListNode(6);

head.next.next.next.next.next.next = head.next.next;

System.out.println("LinkedList cycle start: " + LinkedListCycleStart.findCycleStart(head).value);

head.next.next.next.next.next.next = head.next.next.next;

System.out.println("LinkedList cycle start: " + LinkedListCycleStart.findCycleStart(head).value);

head.next.next.next.next.next.next = head;

System.out.println("LinkedList cycle start: " + LinkedListCycleStart.findCycleStart(head).value);

}

}

Given a list of intervals, **merge all the overlapping intervals** to produce a list that has only mutually exclusive intervals.

import java.util.\*;

class Interval {

int start;

int end;

public Interval(int start, int end) {

this.start = start;

this.end = end;

}

};

class MergeIntervals {

public static List<Interval> merge(List<Interval> intervals) {

if (intervals.size() < 2)

return intervals;

// sort the intervals by start time

Collections.sort(intervals, (a, b) -> Integer.compare(a.start, b.start));

List<Interval> mergedIntervals = new LinkedList<Interval>();

Iterator<Interval> intervalItr = intervals.iterator();

Interval interval = intervalItr.next();

int start = interval.start;

int end = interval.end;

while (intervalItr.hasNext()) {

interval = intervalItr.next();

if (interval.start <= end) { // overlapping intervals, adjust the 'end'

end = Math.max(interval.end, end);

} else { // non-overlapping interval, add the previous interval and reset

mergedIntervals.add(new Interval(start, end));

start = interval.start;

end = interval.end;

}

}

// add the last interval

mergedIntervals.add(new Interval(start, end));

return mergedIntervals;

}

public static void main(String[] args) {

List<Interval> input = new ArrayList<Interval>();

input.add(new Interval(1, 4));

input.add(new Interval(2, 5));

input.add(new Interval(7, 9));

System.out.print("Merged intervals: ");

for (Interval interval : MergeIntervals.merge(input))

System.out.print("[" + interval.start + "," + interval.end + "] ");

System.out.println();

input = new ArrayList<Interval>();

input.add(new Interval(6, 7));

input.add(new Interval(2, 4));

input.add(new Interval(5, 9));

System.out.print("Merged intervals: ");

for (Interval interval : MergeIntervals.merge(input))

System.out.print("[" + interval.start + "," + interval.end + "] ");

System.out.println();

input = new ArrayList<Interval>();

input.add(new Interval(1, 4));

input.add(new Interval(2, 6));

input.add(new Interval(3, 5));

System.out.print("Merged intervals: ");

for (Interval interval : MergeIntervals.merge(input))

System.out.print("[" + interval.start + "," + interval.end + "] ");

System.out.println();

}

}

Cyclic Sort:

### **Problem Statement** [#](https://www.educative.io/courses/grokking-the-coding-interview/B8qXVqVwDKY#problem-statement)

We are given an array containing ‘n’ objects. Each object, when created, was assigned a unique number from 1 to ‘n’ based on their creation sequence. This means that the object with sequence number ‘3’ was created just before the object with sequence number ‘4’.

Write a function to sort the objects in-place on their creation sequence number in

O(n)

*O*(*n*) and without any extra space. For simplicity, let’s assume we are passed an integer array containing only the sequence numbers, though each number is actually an object.

class CyclicSort {

public static void sort(int[] nums) {

int i = 0;

while (i < nums.length) {

int j = nums[i] - 1;

if (nums[i] != nums[j])

swap(nums, i, j);

else

i++;

}

}

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

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

CyclicSort.sort(arr);

for (int num : arr)

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

System.out.println();

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

CyclicSort.sort(arr);

for (int num : arr)

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

System.out.println();

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

CyclicSort.sort(arr);

for (int num : arr)

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

System.out.println();

}

}

Given the head of a Singly LinkedList, reverse the LinkedList. Write a function to return the new head of the reversed LinkedList.

class ListNode {

int value = 0;

ListNode next;

ListNode(int value) {

this.value = value;

}

}

class ReverseLinkedList {

public static ListNode reverse(ListNode head) {

ListNode current = head; // current node that we will be processing

ListNode previous = null; // previous node that we have processed

ListNode next = null; // will be used to temporarily store the next node

while (current != null) {

next = current.next; // temporarily store the next node

current.next = previous; // reverse the current node

previous = current; // before we move to the next node, point previous to the current node

current = next; // move on the next node

}

// after the loop current will be pointing to 'null' and 'previous' will be the new head

return previous;

}

public static void main(String[] args) {

ListNode head = new ListNode(2);

head.next = new ListNode(4);

head.next.next = new ListNode(6);

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

head.next.next.next.next = new ListNode(10);

ListNode result = ReverseLinkedList.reverse(head);

System.out.print("Nodes of the reversed LinkedList are: ");

while (result != null) {

System.out.print(result.value + " ");

result = result.next;

}

}

}

BFS Breadth First Search

Given a binary tree, populate an array to represent its level-by-level traversal. You should populate the values of all **nodes of each level from left to right** in separate sub-arrays.

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) {

val = x;

}

};

class LevelOrderTraversal {

public static List<List<Integer>> traverse(TreeNode root) {

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

if (root == null)

return result;

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

List<Integer> currentLevel = new ArrayList<>(levelSize);

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

TreeNode currentNode = queue.poll();

// add the node to the current level

currentLevel.add(currentNode.val);

// insert the children of current node in the queue

if (currentNode.left != null)

queue.offer(currentNode.left);

if (currentNode.right != null)

queue.offer(currentNode.right);

}

result.add(currentLevel);

}

return result;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(12);

root.left = new TreeNode(7);

root.right = new TreeNode(1);

root.left.left = new TreeNode(9);

root.right.left = new TreeNode(10);

root.right.right = new TreeNode(5);

List<List<Integer>> result = LevelOrderTraversal.traverse(root);

System.out.println("Level order traversal: " + result);

}

}

**Problem 1:** Given a binary tree, find its maximum depth (or height).

import java.util.\*;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) {

val = x;

}

};

class MaximumBinaryTreeDepth {

public static int findDepth(TreeNode root) {

if (root == null)

return 0;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

int maximumTreeDepth = 0;

while (!queue.isEmpty()) {

maximumTreeDepth++;

int levelSize = queue.size();

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

TreeNode currentNode = queue.poll();

// insert the children of current node in the queue

if (currentNode.left != null)

queue.add(currentNode.left);

if (currentNode.right != null)

queue.add(currentNode.right);

}

}

return maximumTreeDepth;

}

public static void main(String[] args) {

TreeNode root = new TreeNode(12);

root.left = new TreeNode(7);

root.right = new TreeNode(1);

root.right.left = new TreeNode(10);

root.right.right = new TreeNode(5);

System.out.println("Tree Maximum Depth: " + MaximumBinaryTreeDepth.findDepth(root));

root.left.left = new TreeNode(9);

root.right.left.left = new TreeNode(11);

System.out.println("Tree Maximum Depth: " + MaximumBinaryTreeDepth.findDepth(root));

}

}