A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

* For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

* For example, the next permutation of arr = [1,2,3] is [1,3,2].
* Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
* While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums.

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example usage

int[] nums = {1, 2, 3};

System.out.println("Original array: " + Arrays.toString(nums));

solution.nextPermutation(nums);

System.out.println("Next permutation: " + Arrays.toString(nums));

}

}

class Solution {

public void nextPermutation(int[] nums) {

int idx1 = -1, idx2 = -1;

// Find the first decreasing element from the end

for (int i = nums.length - 2; i >= 0; i--) {

if (nums[i] < nums[i + 1]) {

idx1 = i;

break;

}

}

if (idx1 == -1) {

// If no such element is found, reverse the entire array

reverse(nums, 0, nums.length - 1);

return;

}

// Find the next greater element

for (int i = nums.length - 1; i > idx1; i--) {

if (nums[i] > nums[idx1]) {

idx2 = i;

break;

}

}

// Swap the elements

swap(nums, idx1, idx2);

// Reverse the elements after the swapped index

reverse(nums, idx1 + 1, nums.length - 1);

}

private void swap(int[] arr, int a, int b) {

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

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

while (start < end) {

swap(arr, start, end);

start++;

end--;

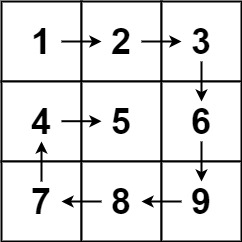
}

}

}

Given an m x n matrix, return all elements of the matrix in spiral order.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example input matrix

int[][] matrix = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

System.out.println("Input matrix:");

for (int[] row : matrix) {

for (int val : row) {

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

}

System.out.println();

}

// Get the spiral order

List<Integer> result = solution.spiralOrder(matrix);

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

}

}

class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

ArrayList<Integer> res = new ArrayList<>();

int n = matrix.length;

int m = matrix[0].length;

int top = 0, left = 0, bottom = n - 1, right = m - 1;

while (top <= bottom && left <= right) {

// Traverse from left to right

for (int i = left; i <= right; i++) {

res.add(matrix[top][i]);

}

top++;

// Traverse from top to bottom

for (int i = top; i <= bottom; i++) {

res.add(matrix[i][right]);

}

right--;

// Traverse from right to left, if top <= bottom

if (top <= bottom) {

for (int i = right; i >= left; i--) {

res.add(matrix[bottom][i]);

}

bottom--;

}

// Traverse from bottom to top, if left <= right

if (left <= right) {

for (int i = bottom; i >= top; i--) {

res.add(matrix[i][left]);

}

left++;

}

}

return res;

}

}

Given a string s, find the length of the **longest**

**substring**

 without repeating characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

import java.util.HashSet;

import java.util.Set;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example test cases

String[] testStrings = {

"abcabcbb",

"bbbbb",

"pwwkew",

"",

"au"

};

for (String test : testStrings) {

System.out.println("Input string: \"" + test + "\"");

int result = solution.lengthOfLongestSubstring(test);

System.out.println("Length of longest substring without repeating characters: " + result);

System.out.println();

}

}

}

class Solution {

public int lengthOfLongestSubstring(String s) {

int n = s.length();

Set<Character> set = new HashSet<>();

int maxLength = 0, i = 0, j = 0;

while (i < n && j < n) {

// Try to extend the range [i, j]

if (!set.contains(s.charAt(j))) {

set.add(s.charAt(j));

maxLength = Math.max(maxLength, j - i + 1);

j++;

} else {

set.remove(s.charAt(i));

i++;

}

}

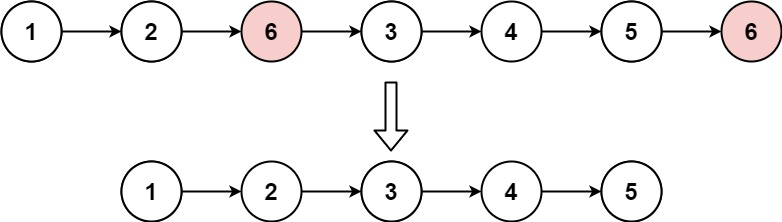
return maxLength;

}

}

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.

**Example 1:**



**Input:** head = [1,2,6,3,4,5,6], val = 6

**Output:** [1,2,3,4,5]

**Example 2:**

**Input:** head = [], val = 1

**Output:** []

**Example 3:**

**Input:** head = [7,7,7,7], val = 7

**Output:** []

class ListNode {

int val;

ListNode next;

ListNode() {}

ListNode(int val) { this.val = val; }

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example linked list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6

ListNode head = new ListNode(1,

new ListNode(2,

new ListNode(6,

new ListNode(3,

new ListNode(4,

new ListNode(5,

new ListNode(6)))))));

System.out.println("Original Linked List:");

printLinkedList(head);

// Remove elements with value 6

int valToRemove = 6;

ListNode updatedHead = solution.removeElements(head, valToRemove);

System.out.println("Linked List after removing " + valToRemove + ":");

printLinkedList(updatedHead);

}

// Utility method to print a linked list

public static void printLinkedList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " -> ");

current = current.next;

}

System.out.println("null");

}

}

class Solution {

public ListNode removeElements(ListNode head, int val) {

// Skip all leading nodes with the value 'val'

while (head != null && head.val == val) {

head = head.next;

}

ListNode current = head;

ListNode prev = null;

// Traverse the list

while (current != null) {

if (current.val == val) {

// Remove the current node by skipping it

prev.next = current.next;

} else {

// Move the prev pointer only if no node is removed

prev = current;

}

current = current.next;

}

return head;

}

}

Given the head of a singly linked list, return true if it is a

palindrome

 or false otherwise.

**Example 1:**



**Input:** head = [1,2,2,1]

**Output:** true

**Example 2:**



**Input:** head = [1,2]

**Output:** false

class ListNode {

int val;

ListNode next;

ListNode() {}

ListNode(int val) { this.val = val; }

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example linked list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6

ListNode head = new ListNode(1,

new ListNode(2,

new ListNode(6,

new ListNode(3,

new ListNode(4,

new ListNode(5,

new ListNode(6)))))));

System.out.println("Original Linked List:");

printLinkedList(head);

// Remove elements with value 6

int valToRemove = 6;

ListNode updatedHead = solution.removeElements(head, valToRemove);

System.out.println("Linked List after removing " + valToRemove + ":");

printLinkedList(updatedHead);

}

// Utility method to print a linked list

public static void printLinkedList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " -> ");

current = current.next;

}

System.out.println("null");

}

}

class Solution {

public ListNode removeElements(ListNode head, int val) {

// Skip all leading nodes with the value 'val'

while (head != null && head.val == val) {

head = head.next;

}

ListNode current = head;

ListNode prev = null;

// Traverse the list

while (current != null) {

if (current.val == val) {

// Remove the current node by skipping it

prev.next = current.next;

} else {

// Move the prev pointer only if no node is removed

prev = current;

}

current = current.next;

}

return head;

}

}