package com.iimtiaz.day\_16;  
  
import java.util.Arrays;  
import java.util.HashMap;  
  
public class Main {  
 public static void main(String[] args) {  
 int[] nums = {1, 2, 0};  
 int[] nums2 = {3, 4, -1, 1};  
 int[] nums3 = {7, 8, 9, 11, 12};  
 System.*out*.println(new Solution\_1().firstMissingPositive(nums));  
 System.*out*.println(new Solution\_2().firstMissingPositive(nums2));  
 System.*out*.println(new Solution\_3().firstMissingPositive(nums3));  
  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(n log n)  
 \* Sorting: The Arrays.sort(nums) operation dominates the time complexity, taking O(n log n) time in general.  
 \* Linear Scan: The subsequent loop iterates through the sorted array once, taking O(n) time, but this is overshadowed  
 \* by the sorting time.  
 \* Space Complexity: O(1)  
 \* Constant Extra Space: The algorithm modifies the input array in-place and doesn't create any additional data  
 \* structures that grow with input size, leading to constant space complexity.  
 \*/*class Solution\_1 {  
 public int firstMissingPositive(int[] nums) {  
 Arrays.*sort*(nums);  
 int smallestPositiveInt = 1;  
 for (int num : nums) {  
 if (num == smallestPositiveInt) {  
 smallestPositiveInt++;  
 }  
 }  
 return smallestPositiveInt;  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(n)  
 \* HashMap Insertions: Each map.put(num, true) operation takes O(1) time on average.  
 \* Iteration Over Positives: The loop iterates through the array once, taking O(n) time.  
 \* Finding Missing Integer: The while loop iterates at most n times, but often much less, as it stops when a missing  
 \* positive is found. Overall, it contributes O(n) time.  
 \* Space Complexity: O(n)  
 \* HashMap Storage: The map can potentially store all positive integers up to n,  
 \*/*class Solution\_2 {  
 public int firstMissingPositive(int[] nums) {  
 HashMap<Integer, Boolean> map = new HashMap<>();  
 for (int num : nums) {  
 if (num > 0) {  
 map.put(num, true);  
 }  
 }  
 int smallestPositiveInt = 1;  
 while (map.containsKey(smallestPositiveInt)) {  
 smallestPositiveInt++;  
 }  
 return smallestPositiveInt;  
 }  
}  
  
*/\*\*  
 \* Time Complexity: O(n)  
 \* Constant-Time Operations: Each loop iterates through the array once, performing constant-time operations within each iteration.  
 \* No Nested Loops or Sorting: The absence of nested loops or sorting algorithms contributes to the linear time complexity.  
 \* Space Complexity: O(1)  
 \* In-Place Modification: The algorithm modifies the input array in-place without using any additional data structures  
 \* that grow with input size, leading to constant space complexity.  
 \*/*class Solution\_3 {  
 public int firstMissingPositive(int[] nums) {  
  
 int n = nums.length;  
  
 // Handle Out-of-Range Numbers:  
 for (int i = 0; i < n; i++) {  
 if (nums[i] <= 0 || nums[i] > n) {  
 nums[i] = n + 1;  
 }  
 }  
  
 // Mark Present Numbers:  
 for (int i = 0; i < n; i++) {  
 int num = Math.*abs*(nums[i]);  
 if (num > n) {  
 continue;  
 }  
 num--;  
 if (nums[num] > 0) {  
 nums[num] = -1 \* nums[num];  
 }  
 }  
  
 // Find Missing Positive:  
 for (int i = 0; i < n; i++) {  
 if (nums[i] >= 0) {  
 return i + 1;  
 }  
 }  
  
 return n + 1;  
 }  
}  
  
*/\*\*  
 \* Given an unsorted array of integers nums, find the smallest positive integer that is not present in the array.  
 \*  
 \* Handle Out-of-Range Numbers:  
 \* Iterate through the array (nums.length times):  
 \* If a number is less than or equal to 0, or greater than n (the array length), replace it with n + 1. This marks  
 \* those numbers as irrelevant for finding the missing positive.  
 \* Mark Present Numbers:  
 \* Iterate through the array again:  
 \* Get the absolute value of the current number (num) to handle potential negative values.  
 \* If num is still greater than n, skip it as it was already marked.  
 \* Otherwise, access the index num - 1 in the array (offset by 1 for zero-based indexing).  
 \* If the value at that index is positive, negate it to mark the presence of num. This clever technique uses the array  
 \* itself as a lookup table.  
 \* Find Missing Positive:  
 \* Iterate through the array one more time:  
 \* If a positive number is found, it means the corresponding index (plus 1) is the missing positive, so return it.  
 \* Handle All Numbers Present:  
 \* If no positive numbers are found, it means the array contains all numbers from 1 to n, so return n + 1 as the  
 \* smallest missing positive.  
 \*/*