Exercises and Homework

java.util Methods for Arrays

fill(A, x)

copyOf(A, n)

copyOfRange(A, s, t):

toString(A)

sort(A):

binarySearch(A, x)

|  |  |  |
| --- | --- | --- |
| 1 | R-3.1 | Give the next five pseudorandom numbers generated by the process described on page 113, with a = 12, b = 5, and n = 100, and 92 as the seed for cur.  See page 113  I apologize, but as an AI text-based model, I don't have access to specific pages or books. Therefore, I cannot provide you with the specific process described on page 113.  However, I can provide you with a general explanation of a pseudorandom number generator (PRNG) and how it can be implemented using the given parameters.  A pseudorandom number generator is an algorithm that generates a sequence of numbers that appear to be random but are actually deterministic. The sequence is typically generated based on an initial value called the seed, and it follows a specific mathematical formula or algorithm.  To generate pseudorandom numbers using the given parameters (a = 12, b = 5, n = 100, and seed = 92), we can use the following formula:  cur = (a \* cur + b) % n  where "cur" represents the current value in the sequence.  To find the next five pseudorandom numbers, we can start with the given seed (cur = 92) and apply the formula iteratively:  1. cur = (12 \* 92 + 5) % 100 = 1109 % 100 = 9  2. cur = (12 \* 9 + 5) % 100 = 113 % 100 = 13  3. cur = (12 \* 13 + 5) % 100 = 161 % 100 = 61  4. cur = (12 \* 61 + 5) % 100 = 737 % 100 = 37  5. cur = (12 \* 37 + 5) % 100 = 449 % 100 = 49  Therefore, the next five pseudorandom numbers generated by the process are: 9, 13, 61, 37, 49. |
| 2 | R-3.2 | Write a Java method that repeatedly selects and removes a random entry from an array until the array holds no more entries.  import java.util.Random;  public class RandomEntryRemoval {  public static void removeRandomEntries(Object[] array) {  Random random = new Random();    while (array.length > 0) {  int randomIndex = random.nextInt(array.length);  Object removedEntry = array[randomIndex];    // Move the last entry to the position of the removed entry  array[randomIndex] = array[array.length - 1];    // Create a new array with reduced size  Object[] newArray = new Object[array.length - 1];  System.arraycopy(array, 0, newArray, 0, newArray.length);    // Assign the new array to the original array variable  array = newArray;    System.out.println("Removed entry: " + removedEntry);  }  }    public static void main(String[] args) {  Integer[] numbers = {1, 2, 3, 4, 5};  removeRandomEntries(numbers);  }  } |
| 3 | R-3.3 | Explain the changes that would have to be made to the program of Code Fragment 3.8 so that it could perform the Caesar cipher for messages that are written in an alphabet-based language other than English, such as Greek, Russian, or Hebrew.  public class CaesarCipherGreek {  public static String encrypt(String message, int shift) {  StringBuilder encryptedMessage = new StringBuilder();  int length = message.length();    for (int i = 0; i < length; i++) {  char ch = message.charAt(i);  if (Character.isLetter(ch)) {  char encryptedChar = (char) (ch + shift);  if (encryptedChar > 'Ω') {  encryptedChar -= 24; // Handle wrap-around for Greek alphabet  }  encryptedMessage.append(encryptedChar);  } else {  encryptedMessage.append(ch);  }  }    return encryptedMessage.toString();  }    public static void main(String[] args) {  String message = "Γεια σας!"; // Greek message  int shift = 3;    String encryptedMessage = encrypt(message, shift);  System.out.println("Encrypted Message: " + encryptedMessage);  }  } |
| 4 | R-3.4 | The TicTacToe class of Code Fragments 3.9 and 3.10 has a flaw, in that it allows a player to place a mark even after the game has already been won by someone. Modify the class so that the putMark method throws an IllegalStateException in that case  public class TicTacToe {  // Other existing code...  public void putMark(int row, int col) {  if (row < 0 || row >= SIZE || col < 0 || col >= SIZE) {  throw new IllegalArgumentException("Invalid board position.");  }  if (board[row][col] != EMPTY) {  throw new IllegalArgumentException("Board position already occupied.");  }  if (winner() != EMPTY) {  throw new IllegalStateException("Game has already been won.");  }  board[row][col] = currentPlayer;  currentPlayer = (currentPlayer == X) ? O : X;  }  // Other existing code...  } |
| 5 | R-3.13 | What is the difference between a shallow equality test and a deep equality test between two Java arrays, A and B, if they are one-dimensional arrays of type int? What if the arrays are two-dimensional arrays of type int?  In Java, a shallow equality test and a deep equality test are used to compare two arrays. The difference between them depends on the type of arrays being compared.  1. One-Dimensional Arrays of type int:  - Shallow Equality Test: A shallow equality test compares the memory references of the arrays to determine if they refer to the same object. In other words, it checks if array A and array B are the same object in memory.  Example: `A == B`    - Deep Equality Test: A deep equality test compares the actual elements within the arrays to check if they have the same values. It iterates over the arrays and compares each element to ensure they are equal.  Example: `Arrays.equals(A, B)`  It's important to note that for one-dimensional arrays of primitive types like `int`, the `Arrays.equals()` method can be used for deep equality tests because the `equals()` method in the `Arrays` class is overridden to compare the actual values of the elements.  2. Two-Dimensional Arrays of type int:  - Shallow Equality Test: A shallow equality test for two-dimensional arrays compares the memory references of the array objects themselves, not the individual elements within the arrays. It checks if array A and array B are the same object in memory.  Example: `A == B`    - Deep Equality Test: A deep equality test for two-dimensional arrays involves comparing both the outer and inner arrays to ensure that all corresponding elements have the same values. It requires nested loops to iterate over the arrays and compare each element.  Example: Custom implementation using nested loops to compare each element.  Unlike one-dimensional arrays of primitive types, two-dimensional arrays of `int` in Java are considered arrays of arrays, where each element is actually a reference to another array. Therefore, a deep equality test requires comparing the inner arrays as well.  It's worth noting that when using the `equals()` method from the `Arrays` class for deep equality tests, it performs a shallow comparison on the outer array and a deep comparison on the inner arrays. So, for two-dimensional arrays, it only checks if the references to the inner arrays are the same, not the actual elements within the inner arrays.  To perform a complete deep equality test on two-dimensional arrays, a custom implementation using nested loops is typically required to compare each individual element. |
| 6 | R-3.14 | Give three different examples of a single Java statement that assigns variable, backup, to a new array with copies of all int entries of an existing array, original.  int[] backup = new int[original.length];  for (int i = 0; i < original.length; i++) {  backup[i] = original[i];  } |
| 7 | C-3.17 | Let A be an array of size n ≥ 2 containing integers from 1 to n−1 inclusive, one of which is repeated. Describe an algorithm for finding the integer in A that is repeated.  public class FindRepeatedInteger {  public static Integer findRepeatedInteger(int[] A) {  int n = A.length;  boolean[] visited = new boolean[n];  for (int num : A) {  if (visited[num]) {  return num;  }  visited[num] = true;  }  return null; // No repeated integer found  }  public static void main(String[] args) {  int[] A = {1, 2, 3, 4, 2}; // Example array with a repeated integer  Integer repeatedInteger = findRepeatedInteger(A);  if (repeatedInteger != null) {  System.out.println("Repeated Integer: " + repeatedInteger);  } else {  System.out.println("No repeated integer found.");  }  }  } |
| 8 | C-3.18 | Let B be an array of size n ≥ 6 containing integers from 1 to n−5 inclusive, five of which are repeated. Describe an algorithm for finding the five integers in B that are repeated.  import java.util.ArrayList;  import java.util.List;  public class FindRepeatedIntegers {  public static List<Integer> findRepeatedIntegers(int[] B) {  int n = B.length;  boolean[] visited = new boolean[n];  List<Integer> repeatedIntegers = new ArrayList<>();  for (int num : B) {  if (visited[num]) {  if (!repeatedIntegers.contains(num)) {  repeatedIntegers.add(num);  }  } else {  visited[num] = true;  }  }  if (repeatedIntegers.size() < 5) {  return null; // Not enough repeated integers found  }  return repeatedIntegers;  }  public static void main(String[] args) {  int[] B = {1, 2, 3, 4, 2, 3, 4, 5, 1, 6}; // Example array with five repeated integers  List<Integer> repeatedIntegers = findRepeatedIntegers(B);  if (repeatedIntegers != null) {  System.out.println("Repeated Integers: " + repeatedIntegers);  } else {  System.out.println("Not enough repeated integers found.");  }  }  }  **Algorithm:**   1. Create a set S to store the distinct elements encountered so far. Initialize S to an empty set. 2. Iterate through the array B: a. For each element b in B: i. If b is not in S, add b to S. This indicates that the element b has been seen once. ii. If b is already in S, then b is a repeated element. Add b to a list of repeated elements. 3. Since there are five repeated elements, continue iterating through B until you find five distinct elements that are repeated. 4. The list of repeated elements contains the five repeated integers in B.   **Analysis:**  Time Complexity: O(n), where n is the size of the array B. This is because the algorithm iterates through the array B only once, and each operation takes constant time. |
| 9 | C-3.19 | Give Java code for performing add(e) and remove(i) methods for the Scoreboard class, as in Code Fragments 3.3 and 3.4, except this time, don’t maintain the game entries in order. Assume that we still need to keep n entries stored in indices 0 to n−1. You should be able to implement the methods without using any loops, so that the number of steps they perform does not depend on n.  public class Scoreboard {  private GameEntry[] board;  private int n = 0; // Number of entries in the scoreboard  public Scoreboard(int capacity) {  board = new GameEntry[capacity];  }  public void add(GameEntry entry) {  if (n < board.length) {  board[n] = entry;  n++;  } else {  int minScoreIndex = findMinScoreIndex();  if (entry.getScore() > board[minScoreIndex].getScore()) {  board[minScoreIndex] = entry;  }  }  }  public GameEntry remove(int i) {  if (i < 0 || i >= n) {  throw new IndexOutOfBoundsException("Invalid index: " + i);  }  GameEntry removedEntry = board[i];  n--;  if (i != n) {  board[i] = board[n];  }  board[n] = null; // Clear the reference to the removed entry  return removedEntry;  }  private int findMinScoreIndex() {  int minIndex = 0;  for (int i = 1; i < n; i++) {  if (board[i].getScore() < board[minIndex].getScore()) {  minIndex = i;  }  }  return minIndex;  }  // Other methods of the Scoreboard class...  public static void main(String[] args) {  Scoreboard scoreboard = new Scoreboard(5);  scoreboard.add(new GameEntry("Player 1", 100));  scoreboard.add(new GameEntry("Player 2", 80));  scoreboard.add(new GameEntry("Player 3", 90));  scoreboard.add(new GameEntry("Player 4", 120));  scoreboard.add(new GameEntry("Player 5", 70));  scoreboard.remove(2);  // Print the remaining entries in the scoreboard  for (int i = 0; i < scoreboard.n; i++) {  System.out.println(scoreboard.board[i]);  }  }  } |
| 10 | C-3.20 | Give examples of values for a and b in the pseudorandom generator given on page 113 of this chapter such that the result is not very random looking, for n = 1000.  X\_i+1 = (a \* X\_i + b) mod n  To find examples of values for a and b such that the result is not very random-looking for n = 1000, we need to select values that do not have good properties of randomness, such as a short period or poor distribution.  Here are a few examples of values for a and b that may result in a non-random-looking sequence for n = 1000:   1. a = 2, b = 0: In this case, the LCG will generate a sequence where every other number is 0 and the rest of the numbers are increasing by 2. The resulting sequence will have a very obvious pattern and not appear random. 2. a = 5, b = 0: This choice of a and b will generate a sequence where every number is a multiple of 5. The sequence will have a clear periodicity and lack randomness. 3. a = 1001, b = 0: Here, a is equal to n + 1, which is not a good choice for a as it will result in a repeating sequence where every number is 0. The generator will fail to produce any other value apart from 0. 4. a = 2, b = 1: In this case, the LCG will generate a sequence where every other number is 1, and the rest of the numbers alternate between 1 and 0. The pattern will be apparent and not resemble random behavior. |
| 11 | C-3.21 | Suppose you are given an array, A, containing 100 integers that were generated using the method r.nextInt(10), where r is an object of type java.util.Random. Let x denote the product of the integers in A. There is a single number that x will equal with probability at least 0.99. What is that number and what is a formula describing the probability that x is equal to that number?  In this case, the integers in array A were generated using the method r.nextInt(10), where r is an object of type java.util.Random. This method generates random integers between 0 (inclusive) and 10 (exclusive), meaning the integers in array A are in the range of 0 to 9.  To find the number that the product x will equal with a probability of at least 0.99, we need to determine which number has the highest probability of occurrence in the array A, and that number will be the one that x is most likely to equal.  Since each integer in array A is chosen uniformly at random from the range 0 to 9, the probability of any specific number appearing at any position in the array is the same, which is 1/10 or 0.1.  To maximize the probability of the product x equaling a specific number, we need to find the number that has the highest occurrence in the array. In this case, the number with the highest occurrence is the number that appears most frequently in the array A.  To calculate the probability that x is equal to a specific number, let's say k, we need to determine the probability of each integer in array A being equal to k. Since the probability of any specific number appearing at any position in the array is 0.1, the probability that x is equal to k can be calculated using the binomial distribution.  The formula for the probability that x is equal to k is given by:  P(x = k) = (0.1^m) \* (0.9^(100 - m)) \* C(100, m)  Where:   * m is the number of occurrences of k in the array A. * C(100, m) represents the number of ways to choose m positions out of 100 positions for k to occur.   To find the value of k that maximizes the probability, we can calculate the probability for each possible value of k (0 to 9) using the above formula and select the value of k that yields the highest probability.  By calculating the probability for each value of k, we find that the number with the highest probability of occurring as the value of x is the number that appears most frequently in the array A.  In summary, to find the number that x will equal with a probability of at least 0.99, we need to calculate the probability for each possible value of k (0 to 9) using the formula provided above. The value of k that yields the highest probability will be the number that x is most likely to equal. |
| 12 | C-3.22 | Write a method, shuffle(A), that rearranges the elements of array A so that every possible ordering is equally likely. You may rely on the nextInt(n) method of the java.util.Random class, which returns a random number between 0 and n−1 inclusive.  import java.util.Random;  public class ArrayShuffler {  public static void shuffle(int[] A) {  Random random = new Random();  int n = A.length;  for (int i = n - 1; i > 0; i--) {  int j = random.nextInt(i + 1);  swap(A, i, j);  }  }  private static void swap(int[] A, int i, int j) {  int temp = A[i];  A[i] = A[j];  A[j] = temp;  }  public static void main(String[] args) {  int[] array = {1, 2, 3, 4, 5};  shuffle(array);  // Print the shuffled array  for (int num : array) {  System.out.print(num + " ");  }  System.out.println();  }  } |
| 13 | C-3.23 | Suppose you are designing a multiplayer game that has n ≥ 1000 players, numbered 1 to n, interacting in an enchanted forest. The winner of this game is the first player who can meet all the other players at least once (ties are allowed). Assuming that there is a method meet(i, j), which is called each time a player i meets a player j (with i 6= j), describe a way to keep track of the pairs of meeting players and who is the winner.  import java.util.HashSet;  import java.util.Set;  public class MultiplayerGame {  private Set<Pair> meetingPairs;  private int[] meetingCount;  private int n;  private int winner;  public MultiplayerGame(int numPlayers) {  n = numPlayers;  meetingPairs = new HashSet<>();  meetingCount = new int[n + 1]; // Indexing from 1 to n  winner = -1;  }  public void meet(int i, int j) {  if (i != j) {  Pair pair = new Pair(i, j);  meetingPairs.add(pair);  meetingCount[i]++;  meetingCount[j]++;  if (meetingCount[i] == n - 1 || meetingCount[j] == n - 1) {  winner = (meetingCount[i] == n - 1) ? i : j;  }  }  }  private class Pair {  private int player1;  private int player2;  public Pair(int player1, int player2) {  this.player1 = player1;  this.player2 = player2;  }  @Override  public int hashCode() {  return player1 \* n + player2;  }  @Override  public boolean equals(Object obj) {  if (this == obj) {  return true;  }  if (obj == null || getClass() != obj.getClass()) {  return false;  }  Pair other = (Pair) obj;  return (player1 == other.player1 && player2 == other.player2) ||  (player1 == other.player2 && player2 == other.player1);  }  }  public int getWinner() {  return winner;  }  public static void main(String[] args) {  int numPlayers = 1000;  MultiplayerGame game = new MultiplayerGame(numPlayers);  // Simulate some meetings  game.meet(1, 2);  game.meet(1, 3);  game.meet(2, 3);  game.meet(4, 5);  game.meet(4, 6);  int winner = game.getWinner();  if (winner != -1) {  System.out.println("Player " + winner + " is the winner!");  } else {  System.out.println("No winner yet.");  }  }  }  Here's a strategy to track pairs of meeting players and determine the  4. Handling Ties: |
| 14 | C-3.24 | Write a Java method that takes two three-dimensional integer arrays and adds them componentwise.  public class ArrayAddition {  public static int[][][] addArrays(int[][][] array1, int[][][] array2) {  int xSize = array1.length;  int ySize = array1[0].length;  int zSize = array1[0][0].length;  int[][][] result = new int[xSize][ySize][zSize];  for (int i = 0; i < xSize; i++) {  for (int j = 0; j < ySize; j++) {  for (int k = 0; k < zSize; k++) {  result[i][j][k] = array1[i][j][k] + array2[i][j][k];  }  }  }  return result;  }  public static void main(String[] args) {  int[][][] array1 = {  {{1, 2, 3}, {4, 5, 6}},  {{7, 8, 9}, {10, 11, 12}}  };  int[][][] array2 = {  {{10, 20, 30}, {40, 50, 60}},  {{70, 80, 90}, {100, 110, 120}}  };  int[][][] sum = addArrays(array1, array2);  // Print the result  for (int i = 0; i < sum.length; i++) {  for (int j = 0; j < sum[0].length; j++) {  for (int k = 0; k < sum[0][0].length; k++) {  System.out.print(sum[i][j][k] + " ");  }  System.out.println();  }  System.out.println();  }  }  }  } |