**Problem Statement**

Write a program using the linear search algorithm to find the second occurrence of a given key value in an array of integers.

**Question Instructions:﻿**

1. Finally, it should print the index of the second occurrence of the key value in the array.
2. If the key value is not found or is found only once in the array, the program should print an appropriate message.

**Input format :**

The first line of input consists of the size of the array.

The second line of input consists of the elements of the array separated by spaces.

The third line of input consists of the key value to be searched

**Output format :**

The output prints the index of the second occurrence of the key value in the array.

If the key value is not found or is found only once in the array, the program should print an appropriate message.

**Refer to the sample output for the formatting specifications.**

**Sample test cases :**

**Input 1 :**

5

1 2 5 6 2

2

**Output 1 :**

The index of the second occurrence of the key value is: 4

**Input 2 :**

4

1 2 3 4

8

**Output 2 :**

Key not found or found only once

**Solution:**

import java.util.Scanner;

public class SecondOccurrence {

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the size of the array from user input

        int size = scanner.nextInt();

        // Initialize an array of the given size

        int[] array = new int[size];

        // Read the elements of the array from user input

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

            array[i] = scanner.nextInt(); // Store each element in the array

        }

        // Read the key value to be searched from user input

        int key = scanner.nextInt();

        // Variables to track the index of the first and second occurrence of the key

        int firstIndex = -1; // Initialize to -1 indicating the key has not been found yet

        int secondIndex = -1; // Initialize to -1 indicating the second occurrence is not found

        // Traverse the array to find the first and second occurrences of the key

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

            if (array[i] == key) { // Check if the current element is the key

                if (firstIndex == -1) {

                    // If firstIndex is -1, it means this is the first time we are encountering the key

                    firstIndex = i;  // Store the index of the first occurrence

                } else if (secondIndex == -1) {

                    // If secondIndex is -1 and we have already found the first occurrence

                    secondIndex = i;  // Store the index of the second occurrence

                }

            }

        }

        // Output the result based on whether the second occurrence was found

        if (secondIndex != -1) {

            // If secondIndex is not -1, it means the key was found twice

            System.out.println("The index of the second occurrence of the key value is: " + secondIndex);

        } else {

            // If secondIndex is still -1, it means the key was found either once or not at all

            System.out.println("Key not found or found only once");

        }

        // Close the scanner to free up resources

        scanner.close();

    }

}

**Detailed Explanation:**

1. **Import and Scanner Initialization**:
   * import java.util.Scanner;: Imports the Scanner class used for reading input.
   * Scanner scanner = new Scanner(System.in);: Creates a Scanner object to read input from the standard input stream (usually the keyboard).
2. **Reading Input**:
   * int size = scanner.nextInt();: Reads the number of elements in the array.
   * int[] array = new int[size];: Initializes an array of integers with the specified size.
   * for (int i = 0; i < size; i++) { array[i] = scanner.nextInt(); }: Reads each element of the array from user input.
3. **Key Value**:
   * int key = scanner.nextInt();: Reads the key value to be searched in the array.
4. **Tracking Occurrences**:
   * int firstIndex = -1; and int secondIndex = -1;: Variables to keep track of the indices where the key appears for the first and second times. Initialized to -1 to indicate the key has not been found yet.
5. **Finding Occurrences**:
   * The for loop traverses the array.
   * if (array[i] == key): Checks if the current element is equal to the key.
     + if (firstIndex == -1): If the key is found and it's the first occurrence, store the index in firstIndex.
     + else if (secondIndex == -1): If it's the second occurrence, store the index in secondIndex.
6. **Output**:
   * if (secondIndex != -1): If secondIndex is not -1, it means the key was found at least twice, so print the index of the second occurrence.
   * else: If secondIndex is still -1, print that the key was not found or found only once.
7. **Closing Scanner**:
   * scanner.close();: Closes the Scanner to release system resources.

This code effectively finds the second occurrence of a key in an array using a linear search approach, with clear and detailed comments explaining each part of the process.

**Problem Statement**

Imagine you are working as a librarian in a large library. The library uses an automated system to keep track of book positions on the shelves. Each book is assigned a unique ID, but some IDs are more common than others due to the popularity of certain editions.

The system logs the positions of books in a sorted list based on their IDs. You need to find the first and last positions of a specific book ID in this sorted list using binary search.

**Example**

**Input:**

10

1 2 2 2 2 3 4 8 8 8

8

**Output:**

7 9

**Explanation:**

The first and last positions of book ID 8 are [7, 9] since the index is 0-based.

**Company Tags:** Amazon and Microsoft

**Input format :**

The first line of input consists of an integer **n**, representing the total number of book IDs.

The second line consists of a sorted list of **n** integers representing the book IDs.

The third line consists of an integer **x**, representing the specific book ID to find.

**Output format :**

The output displays indexes of the first and last occurrence of x, separated by a space.

If book ID x is not present in the shelves, print "NO OCCURRENCES".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 20

1 ≤ book ID, x ≤ 100

**Sample test cases :**

**Input 1 :**

10

1 2 2 2 2 3 4 8 8 8

8

**Output 1 :**

7 9

**Input 2 :**

10

1 2 2 2 2 3 4 8 8 8

5

**Output 2 :**

NO OCCURRENCES

**Solution:**

import java.util.Scanner;

public class BookPositions {

    // Function to find the first occurrence of the book ID

    public static int findFirstOccurrence(int[] arr, int x) {

        // Initialize low and high pointers for binary search

        int low = 0, high = arr.length - 1;

        int result = -1; // Default result if the book ID is not found

        // Perform binary search

        while (low <= high) {

            // Calculate the middle index

            int mid = low + (high - low) / 2;

            // Compare the middle element with the book ID

            if (arr[mid] < x) {

                // If the middle element is less than the book ID, search in the right half

                low = mid + 1;

            } else if (arr[mid] > x) {

                // If the middle element is greater than the book ID, search in the left half

                high = mid - 1;

            } else {

                // If the middle element is equal to the book ID

                result = mid; // Update result to current index

                high = mid - 1; // Continue searching in the left half to find the first occurrence

            }

        }

        return result; // Return the index of the first occurrence or -1 if not found

    }

    // Function to find the last occurrence of the book ID

    public static int findLastOccurrence(int[] arr, int x) {

        // Initialize low and high pointers for binary search

        int low = 0, high = arr.length - 1;

        int result = -1; // Default result if the book ID is not found

        // Perform binary search

        while (low <= high) {

            // Calculate the middle index

            int mid = low + (high - low) / 2;

            // Compare the middle element with the book ID

            if (arr[mid] < x) {

                // If the middle element is less than the book ID, search in the right half

                low = mid + 1;

            } else if (arr[mid] > x) {

                // If the middle element is greater than the book ID, search in the left half

                high = mid - 1;

            } else {

                // If the middle element is equal to the book ID

                result = mid; // Update result to current index

                low = mid + 1; // Continue searching in the right half to find the last occurrence

            }

        }

        return result; // Return the index of the last occurrence or -1 if not found

    }

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of book IDs from user input

        int n = scanner.nextInt();

        // Initialize an array to store the book IDs

        int[] books = new int[n];

        // Read the sorted list of book IDs from user input

        for (int i = 0; i < n; i++) {

            books[i] = scanner.nextInt(); // Store each book ID in the array

        }

        // Read the specific book ID to search for

        int bookId = scanner.nextInt();

        // Find the first occurrence of the book ID

        int firstOccurrence = findFirstOccurrence(books, bookId);

        // Find the last occurrence of the book ID

        int lastOccurrence = findLastOccurrence(books, bookId);

        // Print the result based on whether the book ID was found

        if (firstOccurrence == -1) {

            // If firstOccurrence is -1, the book ID was not found

            System.out.println("NO OCCURRENCES");

        } else {

            // Print the indices of the first and last occurrences

            System.out.println(firstOccurrence + " " + lastOccurrence);

        }

        // Close the scanner to release system resources

        scanner.close();

    }

}

**Problem Statement**

Pavi is working on a project to help a library manage its collection of books efficiently. As part of this project, you need to develop a program to assist the library staff in finding specific books in their extensive catalog.

The library's book collection is organized in such a way that each book is assigned a unique numerical identifier. The library staff has provided you with a list of these identifiers, and you need to write a program to help them find a book's identifier efficiently.

Therefore, you decide to implement a **binary search** algorithm to quickly locate books based on their unique identifiers.

**Input format :**

The first line of input contains an integer,**n**, representing the number of books in the library's collection.

The second line of input consists of n space-separated integers, where each integer represents the unique identifier of a book in the library's collection.

The third line of input contains a single integer t, representing the unique identifier of the book the library staff is searching for.

**Output format :**

If the book with the identifier target is found in the library's collection, the output prints the index (0-based) of the book in the collection.

If the book is not found, print "target Not present", where the target is the unique identifier of a book.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

The test cases will fall under the following constraints:

1 ≤ n ≤ 20

1 ≤ Unique identifier ≤ 100 (Should be in ascending order)

1 ≤ t ≤ 100

**Sample test cases :**

**Input 1 :**

5

12 13 14 15 16

14

**Output 1 :**

2

**Input 2 :**

5

10 14 26 23 35

11

**Output 2 :**

11 not present

import java.util.Scanner;

public class LibrarySearch {

    // Function to perform binary search

    public static int binarySearch(int[] arr, int target) {

        // Initialize low and high pointers for binary search

        int low = 0, high = arr.length - 1;

        // Perform binary search loop

        while (low <= high) {

            // Calculate the middle index

            int mid = low + (high - low) / 2;

            // Check if the middle element is the target

            if (arr[mid] == target) {

                return mid; // Target found, return the index

            } else if (arr[mid] < target) {

                // If the middle element is less than the target, search in the right half

                low = mid + 1;

            } else {

                // If the middle element is greater than the target, search in the left half

                high = mid - 1;

            }

        }

        // If the target is not found, return -1

        return -1;

    }

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of books from the user input

        int n = scanner.nextInt();

        // Initialize an array to store book identifiers

        int[] bookIds = new int[n];

        // Read the sorted list of book identifiers from user input

        for (int i = 0; i < n; i++) {

            bookIds[i] = scanner.nextInt(); // Store each book ID in the array

        }

        // Read the target book identifier to search for

        int target = scanner.nextInt();

        // Call the binarySearch function to find the target book ID

        int result = binarySearch(bookIds, target);

        // Print the result based on whether the target book ID was found

        if (result == -1) {

            // If result is -1, the target was not found in the array

            System.out.println(target + " not present");

        } else {

            // If result is not -1, print the index of the target book ID

            System.out.println(result);

        }

        // Close the scanner to release system resources

        scanner.close();

    }

}

**Detailed Comments and Explanations:**

1. **Imports and Scanner Initialization**:

java

Copy code

import java.util.Scanner;

* + **Purpose**: Imports the Scanner class to handle user input.

1. **Binary Search Function**:

java

Copy code

public static int binarySearch(int[] arr, int target) {

int low = 0, high = arr.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target) {

return mid; // Target found, return the index

} else if (arr[mid] < target) {

low = mid + 1; // Search in the right half

} else {

high = mid - 1; // Search in the left half

}

}

return -1; // Target not found

}

* + **Purpose**: Performs binary search on a sorted array to find the target value.
  + **Logic**:
    - Initialize pointers low and high to define the search range.
    - Calculate the middle index and compare the middle element with the target.
    - Adjust the search range based on comparisons:
      * If the middle element is less than the target, move the low pointer to mid + 1.
      * If the middle element is greater than the target, move the high pointer to mid - 1.
    - If the target is found, return the index.
    - If the loop completes without finding the target, return -1.

1. **Main Method**:

java

Copy code

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

int[] bookIds = new int[n];

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

bookIds[i] = scanner.nextInt();

}

int target = scanner.nextInt();

int result = binarySearch(bookIds, target);

if (result == -1) {

System.out.println(target + " not present");

} else {

System.out.println(result);

}

scanner.close();

}

* + **Reading Inputs**:
    - Create a Scanner object to read inputs.
    - Read the number of book IDs.
    - Initialize an array bookIds and populate it with the sorted book IDs.
    - Read the target book ID to search for.
  + **Performing Search**:
    - Call binarySearch with the bookIds array and target.
  + **Output**:
    - If binarySearch returns -1, print that the target is not present.
    - Otherwise, print the index where the target is found.
  + **Closing Scanner**:
    - scanner.close() is used to release the resources associated with the Scanner.

**Problem Statement**

Sarah, an event organizer, is in need of a program to efficiently manage guest lists for her events using **linear search** algorithm. She wants a tool that allows her to input a list of attendees and a set of keys (representing guest IDs) that she wants to search for.

The program should identify the positions of the keys in the attendee list or notify Sarah if any of the keys are not found in the list.

**Input format :**

The first line of input is an integer, n, representing the number of attendees in the list.

The second line of input consists of n space-separated integers, representing the guest IDs of the attendees.

The third line of input is an integer, k, representing the number of keys (guest IDs) Sarah wants to search for.

The fourth line of input consists of k space-separated integers, representing the guest IDs (keys) Sarah wants to search for.

**Output format :**

The output displays in the following format:

For each key, print either "[key] found at position [x]" where x is the position (1-indexed) of the key in the attendee list, separated by a line.

If the key is not present, print "[key] not found in the array".

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

The test cases will fall under the following constraints:

2 ≤ n, k ≤10

1 ≤ elements, keys ≤ 100

The index should begin at 1

**Sample test cases :**

**Input 1 :**

5

10 20 30 40 50

4

20 30 40 60

**Output 1 :**

20 found at position 2

30 found at position 3

40 found at position 4

60 not found in the array

**Input 2 :**

4

11 22 33 44

2

22 55

**Output 2 :**

22 found at position 2

55 not found in the array

**Input 3 :**

10

89 67 47 38 29 10 36 27 100 1

10

78 67 45 36 27 18 19 100 1 15

**Output 3 :**

78 not found in the array

67 found at position 2

45 not found in the array

36 found at position 7

27 found at position 8

18 not found in the array

19 not found in the array

100 found at position 9

1 found at position 10

15 not found in the array

import java.util.Scanner;

class GuestListManager {

    // Function to perform linear search

    public static int linearSearch(int[] array, int key) {

        // Iterate through each element in the array

        for (int i = 0; i < array.length; i++) {

            // Check if the current element matches the key

            if (array[i] == key) {

                // Return the position of the key (1-indexed)

                return i + 1;

            }

        }

        // If the key is not found, return -1

        return -1;

    }

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of attendees from the user

        int n = scanner.nextInt();

        // Initialize an array to store the attendee IDs

        int[] attendees = new int[n];

        // Read the attendee IDs from user input

        for (int i = 0; i < n; i++) {

            attendees[i] = scanner.nextInt();

        }

        // Read the number of keys to search for

        int k = scanner.nextInt();

        // Initialize an array to store the keys to search

        int[] keys = new int[k];

        // Read the key IDs from user input

        for (int i = 0; i < k; i++) {

            keys[i] = scanner.nextInt();

        }

        // Iterate through each key to search for its position

        for (int key : keys) {

            // Perform linear search to find the position of the key

            int position = linearSearch(attendees, key);

            // Print the result based on whether the key was found

            if (position != -1) {

                // Key found, print its position (1-indexed)

                System.out.println(key + " found at position " + position);

            } else {

                // Key not found, print the appropriate message

                System.out.println(key + " not found in the array");

            }

        }

        // Close the scanner to release system resources

        scanner.close();

    }

}

**Problem Statement**

Arun is given an array of integers where all elements appear twice, except for one element that appears only once. The goal is to find this single, non-duplicate element using a **linear search** algorithm. If the element is not found, print the appropriate message.

Assist Arun in solving this challenge.

**Input format :**

The first line of input consists of an integer **N,** representing the number of elements in the array.

The second line consists of **N** space-separated integers, representing the array elements.

**Output format :**

The output prints an integer, representing the single non-duplicate element in the given array.

If no such element is found, print "No non-duplicate element found".

**Refer to the sample output for formatting specifications.**

**Code constraints :**

The test cases will fall under the following constraints:

1 ≤ N ≤ 15

1 ≤ elements of the array ≤ 30

**Sample test cases :**

**Input 1 :**

9

1 1 2 3 3 4 4 8 8

**Output 1 :**

2

**Input 2 :**

7

12 16 12 15 17 16 15

**Output 2 :**

17

**Input 3 :**

4

14 26 14 26

**Output 3 :**

No non-duplicate element found

import java.util.Scanner;

import java.util.HashMap;

class SingleNonDuplicate {

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of elements in the array

        int N = scanner.nextInt();

        // Initialize an array to store the elements

        int[] array = new int[N];

        // Read the elements into the array

        for (int i = 0; i < N; i++) {

            array[i] = scanner.nextInt();

        }

        // Create a HashMap to store the count of each element

        HashMap<Integer, Integer> countMap = new HashMap<>();

        // Populate the HashMap with counts of each element

        for (int num : array) {

            // Update the count of the current element

            countMap.put(num, countMap.getOrDefault(num, 0) + 1);

        }

        // Variable to track if the non-duplicate element is found

        boolean found = false;

        // Iterate over the keys in the HashMap to find the non-duplicate element

        for (int num : countMap.keySet()) {

            // Check if the count of the current element is 1

            if (countMap.get(num) == 1) {

                // Print the non-duplicate element

                System.out.println(num);

                found = true;

                break; // Exit the loop as we found the non-duplicate element

            }

        }

        // If no non-duplicate element was found, print a message

        if (!found) {

            System.out.println("No non-duplicate element found");

        }

        // Close the scanner to release system resources

        scanner.close();

    }

}

**Detailed Comments and Explanations:**

1. **Imports and Class Declaration**:

java

Copy code

import java.util.Scanner;

import java.util.HashMap;

class SingleNonDuplicate {

* + **Purpose**: Imports necessary classes for user input and data storage and defines the SingleNonDuplicate class.

1. **Main Method**:

java

Copy code

public static void main(String[] args) {

* + **Purpose**: The main entry point of the program where the execution begins.

1. **Reading Inputs**:

java

Copy code

Scanner scanner = new Scanner(System.in);

int N = scanner.nextInt();

int[] array = new int[N];

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

array[i] = scanner.nextInt();

}

* + **Purpose**: Initializes a Scanner to read user inputs.
  + **Logic**:
    - Read the number of elements N.
    - Initialize an array array to store these elements.
    - Populate the array with user-provided values.

1. **Creating and Populating HashMap**:

java

Copy code

HashMap<Integer, Integer> countMap = new HashMap<>();

for (int num : array) {

countMap.put(num, countMap.getOrDefault(num, 0) + 1);

}

* + **Purpose**: Uses a HashMap to store the count of each element in the array.
  + **Logic**:
    - Create a HashMap called countMap to keep track of the frequency of each number.
    - Iterate over the array and update the count of each number in countMap.

1. **Finding the Non-Duplicate Element**:

java

Copy code

boolean found = false;

for (int num : countMap.keySet()) {

if (countMap.get(num) == 1) {

System.out.println(num);

found = true;

break;

}

}

* + **Purpose**: Checks for an element with a count of 1, which is the non-duplicate element.
  + **Logic**:
    - Initialize a boolean variable found to track whether a non-duplicate element has been found.
    - Iterate over the keys in countMap.
    - If an element's count is 1, print the element and set found to true. Exit the loop.

1. **Handling Case When No Non-Duplicate Element is Found**:

java

Copy code

if (!found) {

System.out.println("No non-duplicate element found");

}

* + **Purpose**: Prints a message if no non-duplicate element was found during the search.

1. **Closing the Scanner**:

java

Copy code

scanner.close();

* + **Purpose**: Closes the Scanner to free up system resources.

**Notes:**

* **HashMap Usage**: HashMap provides average time complexity of O(1) for insertions and lookups, making it efficient for counting frequencies.
* **Handling Edge Cases**: This program assumes there is at least one non-duplicate element if the input is valid. If all elements appear twice, it will correctly output the message "No non-duplicate element found".

This code effectively finds a single non-duplicate element in the array by leveraging the efficiency of HashMap to count occurrences.

**Problem Overview**

Given an array where all elements appear twice except for one element which appears only once, our goal is to find that unique element.

**Example**

Let's use the following array for illustration:

java

Copy code

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

In this array:

* The numbers 1, 2, and 4 appear twice.
* The number 3 appears only once.

**Step-by-Step Explanation**

**1. Creating and Populating the HashMap**

java

Copy code

HashMap<Integer, Integer> countMap = new HashMap<>();

// Populate the HashMap with counts of each element

for (int num : array) {

// Update the count of the current element

countMap.put(num, countMap.getOrDefault(num, 0) + 1);

}

* **Initialization**: We create a HashMap called countMap. This will store each unique number from the array as a key and its frequency (how many times it appears) as the value.
* **Population**:
  + **Iteration**: We iterate over each number in the array.
  + **Counting**:
    - For each number, we use countMap.getOrDefault(num, 0) to get the current count of that number. If the number is not yet in the HashMap, getOrDefault returns 0.
    - We then add 1 to this count and update the HashMap with the new count.

**Example Breakdown**:

For the array {1, 2, 3, 2, 1, 4, 4}:

* When processing 1:
  + countMap was empty.
  + countMap.getOrDefault(1, 0) returns 0.
  + countMap.put(1, 1) updates the map to {1=1}.
* Next, processing 2:
  + countMap.getOrDefault(2, 0) returns 0.
  + countMap.put(2, 1) updates the map to {1=1, 2=1}.
* Processing 3:
  + countMap.getOrDefault(3, 0) returns 0.
  + countMap.put(3, 1) updates the map to {1=1, 2=1, 3=1}.
* Processing 2 again:
  + countMap.getOrDefault(2, 1) returns 1.
  + countMap.put(2, 2) updates the map to {1=1, 2=2, 3=1}.
* Processing 1 again:
  + countMap.getOrDefault(1, 1) returns 1.
  + countMap.put(1, 2) updates the map to {1=2, 2=2, 3=1}.
* Processing 4:
  + countMap.getOrDefault(4, 0) returns 0.
  + countMap.put(4, 1) updates the map to {1=2, 2=2, 3=1, 4=1}.
* Processing 4 again:
  + countMap.getOrDefault(4, 1) returns 1.
  + countMap.put(4, 2) updates the map to {1=2, 2=2, 3=1, 4=2}.

After processing all elements, the countMap looks like this:

java

Copy code

{1=2, 2=2, 3=1, 4=2}

**2. Finding the Non-Duplicate Element**

java

Copy code

boolean found = false;

// Iterate over the keys in the HashMap to find the non-duplicate element

for (int num : countMap.keySet()) {

// Check if the count of the current element is 1

if (countMap.get(num) == 1) {

// Print the non-duplicate element

System.out.println(num);

found = true;

break; // Exit the loop as we found the non-duplicate element

}

}

* **Initialization**: We set a boolean variable found to false initially.
* **Iteration**: We iterate over each key (number) in the HashMap.
* **Check Count**:
  + For each key, we check if its count is 1.
  + If it is, we print this number as the non-duplicate element and set found to true.

**Example Breakdown**:

For the countMap {1=2, 2=2, 3=1, 4=2}:

* Checking 1: count is 2, so we continue.
* Checking 2: count is 2, so we continue.
* Checking 3: count is 1, which is what we are looking for. We print 3 and set found to true.

If no key with count 1 is found, we would print "No non-duplicate element found."

**Conclusion**

Using HashMap allows us to efficiently count occurrences of each element in the array. By leveraging the HashMap, we can easily find and identify the non-duplicate element by checking counts. The complexity of counting and looking up values in a HashMap is generally O(1), making this method efficient for the problem at hand.

import java.util.Scanner;

public class SingleNonDuplicate {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read the number of elements

        int N = scanner.nextInt();

        // Read the array elements

        int[] array = new int[N];

        for (int i = 0; i < N; i++) {

            array[i] = scanner.nextInt();

        }

        // Variable to store the result of XOR operations

        int result = 0;

        // XOR all elements in the array

        for (int num : array) {

            result ^= num;

        }

        // Output the result

        // If the result is 0 and N is even, then no single non-duplicate is found.

        if (result != 0) {

            System.out.println(result);

        } else {

            System.out.println("No non-duplicate element found");

        }

        scanner.close();

    }

}

**XOR (Exclusive OR)**

**XOR** is a bitwise operation that stands for **Exclusive OR**. It operates on the individual bits of binary numbers. The XOR operation compares the bits of two numbers and returns a new number, where each bit is set to 1 if the corresponding bits of the operands are different, and 0 if they are the same.

**XOR Operation Table**

| **A** | **B** | **A XOR B** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Key Properties of XOR:**

1. **Self-Canceling**: a ^ a = 0 (any number XORed with itself results in 0).
2. **Identity**: a ^ 0 = a (any number XORed with 0 remains unchanged).
3. **Commutative**: a ^ b = b ^ a (order of operands does not affect the result).
4. **Associative**: (a ^ b) ^ c = a ^ (b ^ c) (grouping of operands does not affect the result).

**Example**:

* 5 in binary is 0101
* 3 in binary is 0011
* 5 ^ 3 results in 0110, which is 6 in decimal.

**Use in Finding Non-Duplicate Elements**: In problems where all elements except one appear twice, XOR can be used to find the single non-duplicate element efficiently because pairs cancel each other out.

**HashMap**

**HashMap** is a data structure provided by Java (and similar structures exist in other programming languages) that implements the Map interface. It stores key-value pairs, allowing for fast retrieval of values based on their keys.

**Key Features of HashMap:**

1. **Key-Value Storage**: Each entry in a HashMap consists of a key and a value. The key is used to retrieve the corresponding value.
2. **Hashing**: Internally, HashMap uses a hash table to store entries. Keys are hashed to determine their position in the table.
3. **Constant-Time Complexity**: The average time complexity for get, put, and remove operations is O(1), though it can degrade to O(n) in the worst-case scenarios (e.g., many keys hash to the same index).

**Key Operations:**

1. **Put**: map.put(key, value) adds or updates a key-value pair in the map.
2. **Get**: map.get(key) retrieves the value associated with the key.
3. **Remove**: map.remove(key) removes the key-value pair associated with the key.

**Problem Statement**

Navigating through a bustling city, a traveler seeks to ascertain the availability of a particular bus number within a list of bus numbers.

In your role as a programmer aiding the traveler, your objective is to develop a program that employs the **binary search** approach to determine whether the provided bus number can be located within the list.

**Input format :**

The first line consists of an integer **n,** representing the number of bus numbers.

The second line consists of **n** space-separated integers, representing the list of bus numbers.

The third line consists of an integer **x,** representing the bus number to search.

**Output format :**

If the bus number is found in the list, print "Bus No. found at index " followed by the index(0-indexed).

Otherwise, print "Bus No. Not found".

**Refer to the sample output for the exact format.**

**Code constraints :**

In this scenario, the test cases fall under the following constraints:

1 ≤ n ≤ 15

1 ≤ bus numbers ≤ 100

1 ≤ x ≤ 100

**Sample test cases :**

**Input 1 :**

5

43 67 89 90 96

90

**Output 1 :**

Bus No. found at index 3

**Input 2 :**

5

43 67 89 90 96

91

**Output 2 :**

Bus No. Not found

import java.util.Scanner;

class Main {

    // Method to perform binary search on a sorted array

    public static int binarySearch(int[] array, int x) {

        int low = 0; // Initialize the starting index of the search range

        int high = array.length - 1; // Initialize the ending index of the search range

        // Continue searching while the range is valid

        while (low <= high) {

            int mid = low + (high - low) / 2; // Calculate the middle index

            if (array[mid] == x) {

                return mid; // Target found at index mid

            } else if (array[mid] < x) {

                low = mid + 1; // Search in the right half

            } else {

                high = mid - 1; // Search in the left half

            }

        }

        return -1; // Target not found in the array

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Create a Scanner object to read input

        int n = scanner.nextInt(); // Read the number of bus numbers

        int[] busNumbers = new int[n]; // Create an array to store the bus numbers

        // Read the bus numbers into the array

        for (int i = 0; i < n; i++) {

            busNumbers[i] = scanner.nextInt();

        }

        int x = scanner.nextInt(); // Read the bus number to search for

        // Perform binary search to find the index of the bus number

        int resultIndex = binarySearch(busNumbers, x);

        // Print the result based on whether the bus number was found

        if (resultIndex != -1) {

            System.out.println("Bus No. found at index " + resultIndex);

        } else {

            System.out.println("Bus No. Not found");

        }

        scanner.close(); // Close the scanner

    }

}

**Explanation of the Code**

1. **Binary Search Method (binarySearch)**:
   * **Purpose**: To find the index of a specific bus number (x) in a sorted array (array).
   * **Parameters**:
     + array: The sorted array of bus numbers.
     + x: The bus number to search for.
   * **Algorithm**:
     + Initialize low to 0 and high to the last index of the array.
     + Calculate the middle index (mid).
     + Compare the value at mid with x:
       - If equal, return mid (index where the bus number is found).
       - If x is greater, adjust low to mid + 1 to search in the right half.
       - If x is smaller, adjust high to mid - 1 to search in the left half.
     + If low exceeds high, x is not in the array, so return -1.
2. **Main Method**:
   * **Input Reading**:
     + Use Scanner to read input from the user.
     + First, read the number of bus numbers (n).
     + Populate the busNumbers array with n integers from the user.
     + Read the target bus number (x) to search for.
   * **Binary Search**:
     + Call binarySearch to find the index of x in busNumbers.
   * **Output**:
     + Print the index if the bus number is found.
     + Print "Bus No. Not found" if the bus number is not in the array.
3. **Scanner Close**:
   * Close the Scanner object to free up resources.

**Key Points**

* **Binary Search**: Efficiently finds an element in a sorted array with a time complexity of O(log n).
* **Sorted Array**: Binary search requires the array to be sorted; otherwise, results will be incorrect.
* **Handling Output**: The program correctly distinguishes between a found and not found scenario, providing appropriate feedback to the user.