Question 1

Write a method that takes 2 parameters. The first parameter is an array of integers that has already been sorted from smallest to largest. The second parameter is an integer. The method should return the number of times the second parameter appears in the first parameter.

Example:

If the first parameter is [1, 2, 3, 3, 4, 5] and the second parameter is 3. The method should return 2.

Part 1: Please explain your algorithm.

Less optimal approach:

We go through the entire array linearly, checking each element to determine if it matches the number we are searching for. If there is a match, we increment a counter. This will return the number of matches found in the array for the number being compared.

This approach is the least optimal since it has a time complexity of O(n), as we are checking each element individually, which is not efficient for large arrays.

package lessOptimalApproach;

public class Response{

public static int occurrences(int[] arr, int number) {

int count = 0;

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

if (arr[i] == number) {

count++;

}

}

return count;

}

public static void main(String[] args){

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

int number = 5;

System.out.println(occurrences(arr,number));

}

}

Most Optimal Response:

The occurrences function is responsible for calculating how many times a specific number appears in a sorted array. To do this, it calls the binarySearch method twice: once to find the first occurrence of the number and once to find the last occurrence. In this way, it calculates the difference between both indices to determine how many times the number is repeated.

The binarySearch method performs a binary search in the array. It uses an additional parameter called findFirst, which indicates whether the first or last occurrence of the number is searched. If findFirst is true, the search focuses on the left half of the array after finding the target, ensuring that the first occurrence is found. If findFirst is false, the search continues on the right half to find the last occurrence. The time complexity of this approach is O(log n), which makes this solution very efficient even for large arrays.

package mostOptimalResponse;

public class Response {

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

int firstIndex = binarySearch(arr, target, true);

int lastIndex = binarySearch(arr, target, false);

if (firstIndex == -1) {

return 0;

}

return lastIndex - firstIndex + 1;

}

private static int binarySearch(int[] arr, int target, boolean findFirst) {

int left = 0, right = arr.length - 1;

int result = -1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

result = mid;

if (findFirst) {

right = mid - 1;

} else {

left = mid + 1;

}

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return result;

}

public static void main(String[] args) {

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

int target = 3;

int result = occurrences(arr, target);

System.out.println("The number " + target + " appears " + result + " times.");

}

}

public class ResponseTest {

public static void main(String[] args) {

lessOptimalApproach.Response lessOptimalResponse = new lessOptimalApproach.Response();

int[] arr1 = {1, 2, 3, 3, 4, 5};

int result1 = lessOptimalResponse.occurrences(arr1, 3);

System.out.println(result1 == 2 ? "Less Optimal Test 1 Passed." : "Test 1 Failed: Expected 2, but got " + result1);

int[] arr2 = {1, 2, 3, 3, 4, 5};

int result2 = lessOptimalResponse.occurrences(arr2, 1);

System.out.println(result2 == 1 ? "Less Optimal Test 2 Passed." : "Test 2 Failed: Expected 1, but got " + result2);

int[] arr3 = {1, 2, 3, 3, 4, 5};

int result3 = lessOptimalResponse.occurrences(arr3, 6);

System.out.println(result3 == 0 ? "Less Optimal Test 3 Passed." : "Test 3 Failed: Expected 0, but got " + result3);

int[] arr4 = {1, 2, 3, 3, 4, 5, 5, 5, 5};

int result4 = lessOptimalResponse.occurrences(arr4, 5);

System.out.println(result4 == 4 ? "Less Optimal Test 4 Passed." : "Test 4 Failed: Expected 4, but got " + result4);

mostOptimalResponse.Response mostOptimalResponse = new mostOptimalResponse.Response();

int result5 = mostOptimalResponse.occurrences(arr1, 3);

System.out.println(result5 == 2 ? "Most Optimal Test 1 Passed." : "Test 1 Failed: Expected 2, but got " + result5);

int result6 = mostOptimalResponse.occurrences(arr2, 1);

System.out.println(result6 == 1 ? "Most Optimal Test 2 Passed." : "Test 2 Failed: Expected 1, but got " + result6);

int result7 = mostOptimalResponse.occurrences(arr3, 6);

System.out.println(result7 == 0 ? "Most Optimal Test 3 Passed." : "Test 3 Failed: Expected 0, but got " + result7);

int result8 = mostOptimalResponse.occurrences(arr4, 5);

System.out.println(result8 == 4 ? "Most Optimal Test 4 Passed." : "Test 4 Failed: Expected 4, but got " + result8);

int[] arr5 = {1, 1, 1, 1, 1};

int result9 = mostOptimalResponse.occurrences(arr5, 1);

System.out.println(result9 == 5 ? "Most Optimal Test 5 Passed." : "Test 5 Failed: Expected 5, but got " + result9);

int[] arr6 = {0, 0, 0, 0, 0};

int result10 = mostOptimalResponse.occurrences(arr6, 0);

System.out.println(result10 == 5 ? "Most Optimal Test 6 Passed." : "Test 6 Failed: Expected 5, but got " + result10);

int[] arr7 = {10, 20, 30, 40, 50};

int result11 = mostOptimalResponse.occurrences(arr7, 100);

System.out.println(result11 == 0 ? "Most Optimal Test 7 Passed." : "Test 7 Failed: Expected 0, but got " + result11);

}

}

Question 2

Assume are climbing a staircase of 10 steps. Also, assume that you can only take 1 or 2 steps each time you move up the stairs. Below is an example of two different ways you can climb the staircase.

1, 1, 2, 2, 1, 2, 1

2, 2, 2, 2, 2

Write an algorithm to calculate the total number of unique ways we can climb the staircase.

public class Response {

public static int climb(int n) {

if (n == 0) return 1;

if (n == 1) return 1;

int prev2 = 1, prev1 = 2;

for (int i = 3; i <= n; i++) {

int current = prev1 + prev2;

prev2 = prev1;

prev1 = current;

}

return prev1;

}

public static void main(String[] args) {

int n = 10;

System.out.println(climb(n));

}

}

public class ResponseTest {

public static void main(String[] args) {

test(0, 1);

test(1, 1);

test(2, 2);

test(3, 3);

test(4, 5);

test(5, 8);

test(10, 89);

}

private static void test(int n, int expected) {

int result = Response.climb(n);

System.out.println("waysToClimb(" + n + ") = " + result + " | Expected: " + expected

+ " | " + (result == expected ? "PASS" : "FAIL"));

}

}

Part 1: If you do a google search on this problem, you will find that one of the solutions uses the Fibonacci sequence. Explain how this solution works. You do not need to write any code.

**Explanation to solve the problem using the Fibonacci sequence**

To solve this problem, imagine you're climbing a staircase with 10 steps, and you can only take 1 or 2 steps at a time. The question is: how many different ways can you climb the staircase? To answer this, we need to think about how you can reach each step. For the first step, there is only one way to get there: take 1 step. For the second step, you can either take two 1-steps or one 2-step. These are the first two ways to climb.

From the third step onwards, you can reach it by either taking 1 step from the previous step or 2 steps from two steps below. This means the number of ways to get to each step is the sum of the ways to reach the two previous steps. This is similar to the Fibonacci sequence, where each number is the sum of the two numbers before it. Using this pattern, we can find the total number of ways to reach the 10th step.

Part 2: If we do not use Fibonacci sequence solution, how could we solve this problem? Please explain your algorithm.

The process begins by initializing the variables with f(1) = 1 and f(2) = 2, which represent the unique ways to reach the first and second steps respectively. From the third step or rung onwards, the number of unique ways that can be deduced is calculated using the formula f(n) = f(n-1) + f(n-2), which adds the ways from the previous two rungs. This pattern continues until the tenth rung is reached.

Now, if we do it step by step, the results are as follows: for n = 1, there is 1 way; for n = 2, there are 2 ways; for n = 3, there are 3 ways; and for n = 4, 5 ways are found. As you go further, the number of ways keeps growing: n = 5 has 8 ways, n = 6 has 13 ways, n = 7 has 21 ways, n = 8 reaches 34 ways, and n = 9 goes up to 55 ways. Finally, for n = 10, you get 89 unique ways to reach the top.

This pattern is very similar to the Fibonacci sequence, where each number is the sum of the previous two and so we can solve this problem.

Question 3

Assume that someone is giving you the following requirements for developing a REST API with 2 endpoints.

Endpoint 1

type: POST

url: /cars

description: create car entry

Request:

{

"model": "Honda Civic",

"year": 2020

}

Response:

{

"id": 1

}

Endpoint 2

type: GET

url: /cars/{id}

description: retrieve car details with id (the id is the same one from the response of the post)

Response:

{

"model": "Honda Civic",

"year": 2020

}

API Implementation Details:

The microservice will store the car data in a database. The id of each car entry is the primary key of the record in the table.

Part 1: What additional information do you need before you begin coding this?

To define the database and schema, it is critical to determine whether PostgreSQL, MySQL, SQLite, or stored procedures will be used. Also, it must be decided whether the id field will be auto-incremented with @GeneratedValue(strategy = GenerationType.IDENTITY) or whether a UUID will be used for added security and scalability. In addition to the basic fields (id, model, year), additional attributes such as color, price, or status may be required. Finally, it is important to set constraints, such as ensuring that the year is not less than 1886, the year of the first automobile.

Regarding the project configuration, one must choose between Spring Data JPA with Hibernate or manual SQL queries for data management. It is also advisable to integrate Swagger/OpenAPI to document endpoints and facilitate integration with other applications. To ensure future compatibility, it is good practice to define an API versioning strategy, for example, structuring paths as /api/v1/cars.

Validation and error handling are key to a robust system. Annotations like @NotBlank, @Size, @Min, @Max can be applied on the Car entity attributes to validate the input data. If an ID does not exist in the database, the system should respond with a 404 Not Found. Centralized exception handling will help provide consistent responses to clients.

Finally, it is essential to define the authentication and authorization system. It should be determined whether API access will be restricted to authenticated users only and which method will be used (JWT, OAuth2, API Keys). Also, whether there will be different user roles, such as admin to create cars and regular users only to query them. If OAuth2 is chosen, it must be decided which provider to integrate (Google, Okta, Auth0, Keycloak). It is also advisable to implement rate limiting or IP restrictions to avoid abuses in API consumption.

Part 2: Make some assumptions for the questions that you came up with in part 1. Create a list of test scenarios for the requirements listed above. Be sure to indicate what the input and expected output should be for each test.

Create a car entry with valid data

Input:

{

"model": "Honda Civic",

"year": 2020

}

Expected Output:

{

"id": 1

}

Retrieve car details with valid ID

Input: GET /cars/1

Expected Output:

{

"model": "Honda Civic",

"year": 2020

}

Retrieve car details with invalid ID (non-existent car)

Input: GET /cars/999

Expected Output: HTTP Status: 404 Not Found

Create car entry with invalid data (missing model)

Input:

{

"year": 2020

}

Expected Output: HTTP Status: 400 Bad Request

Create car entry with invalid data (year before 1886)

Input:

{

"model": "Old Timer",

"year": 1800

}

Expected Output: HTTP Status: 400 Bad Request (Year must not be less than 1886)

Question 4

Imagine you are doing a code review of a pull request containing the following code. Would you approve or reject this pull request? What feedback would you offer?

public boolean checkCredendials() {

UserCredentials credentials = get\_login\_form\_input();

User user = fetchUserProfileFromDB(credentials.username);

if (user == null)

throw new LoginFailedException(“invalid username”);

if (credendials.password != user.password) {

throw new LoginFailedException(“Invalid Password”);

}

LOGGER.info(“User {} logged in using password {}”,

credentials.username,

credentials.password);

return true;

}

I would reject this pull request due to multiple security and code quality issues. First, there is a major security risk in logging sensitive data, as the code logs both the username and password in plain text. This should be removed entirely, and only the username should be logged if necessary. Additionally, the password is being compared using !=, which is incorrect since passwords should be securely hashed and compared using a function like BCrypt.checkpw().

From a code quality perspective, there are spelling errors in method and variable names, such as checkCredendials() instead of checkCredentials(), which can reduce code readability. Moreover, there is a potential NullPointerException if get\_login\_form\_input() returns null, which should be handled properly before accessing properties. Exception handling is also inconsistent, as error messages for invalid credentials expose whether the username exists or not, which could be exploited by attackers. A more secure approach would be to return a generic error message like "Invalid username or password".

To address these issues, the PR should be rejected until proper security measures are implemented, such as using BCrypt for password storage, fixing exception handling, and improving method and variable naming. Sensitive logging should be removed entirely to protect user data.

Summary:

The request should be rejected due to security risks and code quality issues.

I would recommend fixes for logging, password handling, exception handling, and typos.