GTU Department Of Computer Engineering

CSE 222 / 505 – Spring 2023

Homework 4

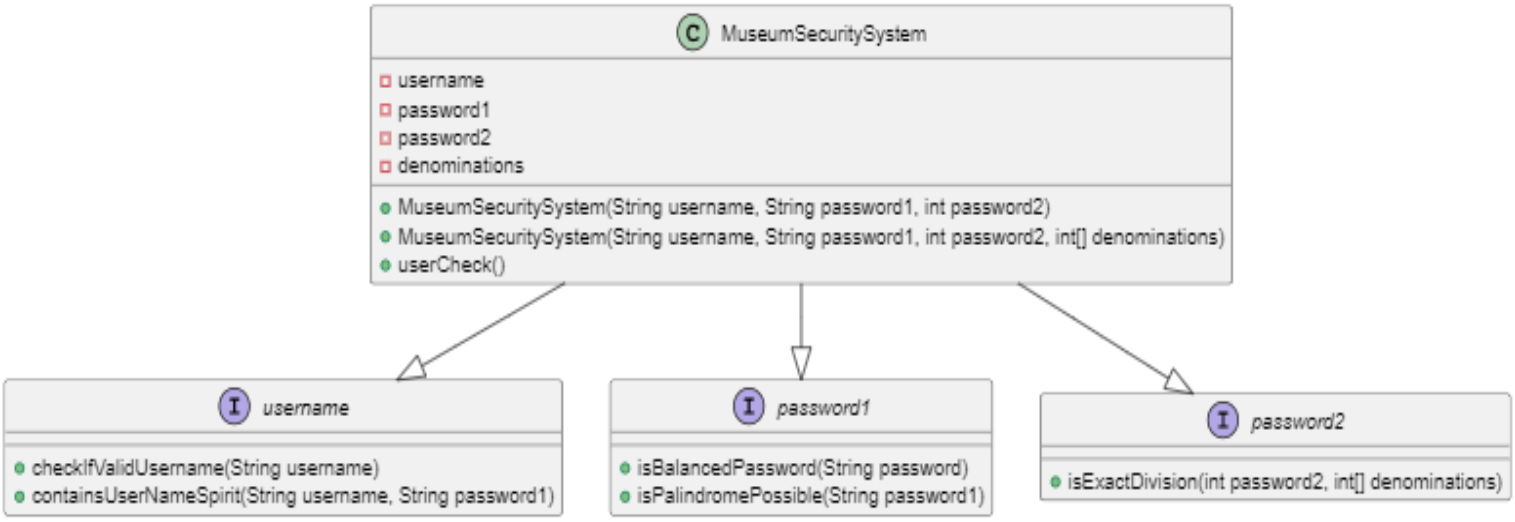
Ahmet CEBECİ

1901042708

INTRODUCTION

We introduce the Topkapı Museum Security System Java class, which is designed to authenticate usernames and passwords, allowing authorized users to securely access the system. We will delve into the details of the class's functioning, how username and password validation are performed, the criteria that need to be met, and the use of recursive algorithms. Our goal is to provide a comprehensive understanding of how the security system works and the purposes for which it can be used.

CLASS DIAGRAM



CODE FUNCTIONALITY

1 – boolean checkIfValidUsername(String username)

boolean checkIfValidUsernameRecursive(String username)

```java
protected boolean checkIfValidUsername(String username) {
    if (username.isEmpty()) {
        System.out.println(x:"The username is invalid. It should have at least 1 character.");
        return false;
    }
    return checkIfValidUsernameRecursive(username);
}
private boolean checkIfValidUsernameRecursive(String username) {
    if (username.isEmpty()) {
        return true;
    }
    if (!Character.isLetter(username.charAt(index:0))) {
        System.out.println(x:"The username is invalid. It should have letters only");
        return false;
    }
    return checkIfValidUsernameRecursive(username.substring(beginIndex:1));
}
```

The checkIfValidUsername function receives the username as a parameter. Initially, it checks if the username is empty. If so, the function returns false and displays an error message indicating that the username should have at least one character.

If the username is not empty, the checkIfValidUsernameRecursive function is called with the username as the argument. This function uses recursion to check each character of the username.

In the checkIfValidUsernameRecursive function, the base case is when the username is empty, which means that all the characters have been checked and the username is valid. In this case, the function returns true.

If the username is not empty, the function checks the first character of the username. If it is not a letter (i.e., not an uppercase or lowercase alphabetical character), the function returns false and displays an error message stating that the username should consist of letters only.

If the first character is a letter, the function continues to check the rest of the username by calling itself recursively with the substring of the username starting from the second character.

The checkIfValidUsernameRecursive function, while checking each character in the username, moves on to the next character of the username in each step and reduces the string by one character. This means that the function will be called at most n times and will check each character only once. Therefore, the time complexity increases linearly and can be expressed as O(n).

2 - boolean containsUserNameSpirit(String username, String password1)

```
protected boolean containsUserNameSpirit(String username, String password1) {
    Stack<Character> stack = new Stack<>();
    for (int i = 0; i < password1.length(); i++) {
        stack.push(password1.charAt(i));
        if (username.indexOf(password1.charAt(i)) != -1) {
            return true;
        }
    }
    System.out.println(x:"The password1 is invalid. It should have at least 1 character from the username.");
    return false;
}
```

The function takes the username and password1 as parameters.

It initializes a new stack of characters called stack.

The function iterates through each character in password1 using a for loop.

In each iteration, it gets the character at the current index i from password1 and assigns it to the variable passwordChar.

The passwordChar is then pushed onto the stack.

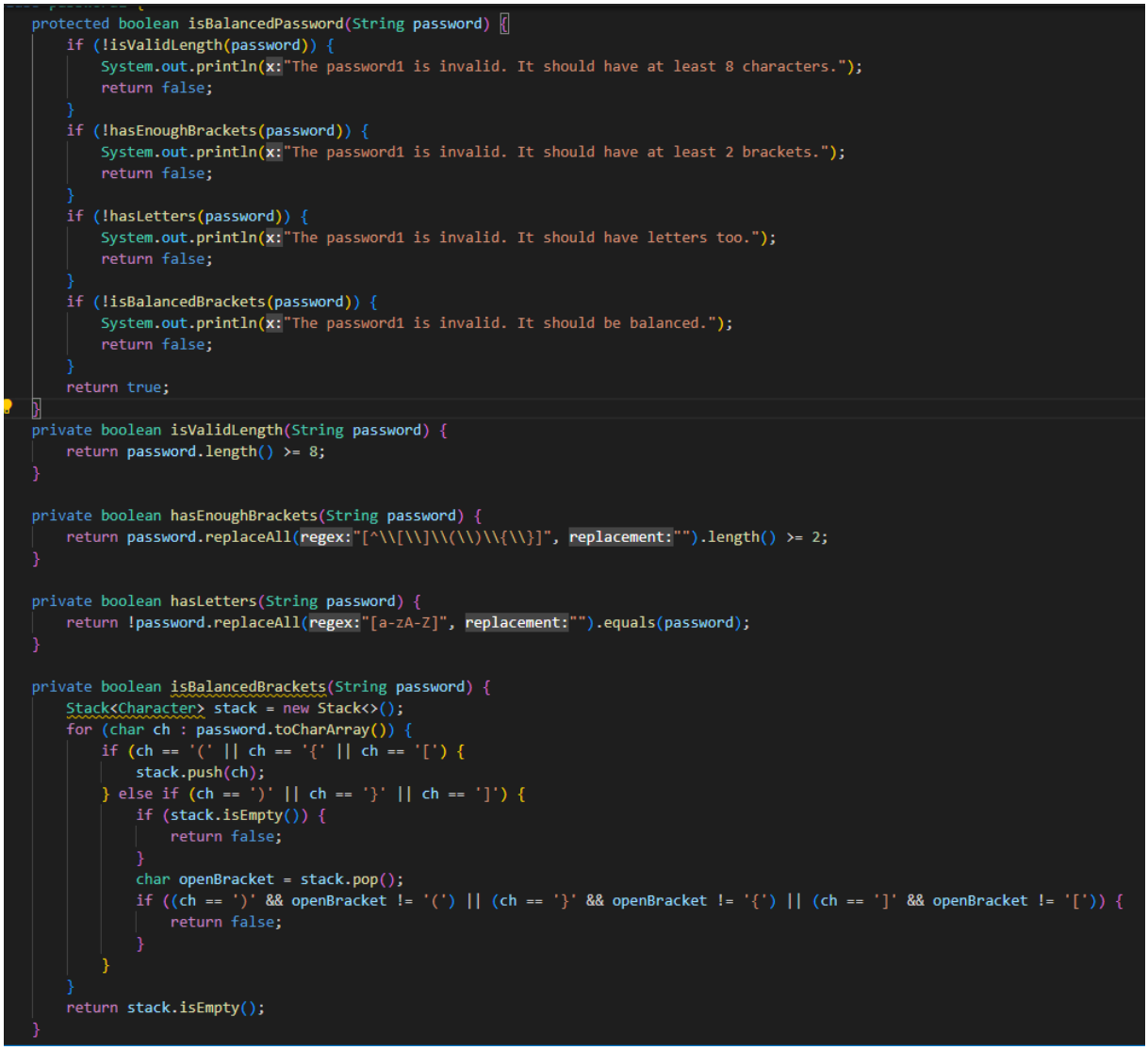
The function checks if the current passwordChar exists in the username using the indexOf method. If the character is found in the username (i.e., indexOf returns an index different from -1), the function returns true, indicating that the password contains at least one character from the username.

If the loop finishes and no character from the username is found in the password1, the function prints an error message stating that the password is invalid because it should have at least one character from the username, and it returns false.

The containsUserNameSpirit function ensures that the password contains at least one character from the username, which is a requirement for a valid password in this specific implementation.

When analyzing the time complexity of the function, the for loop performs m steps depending on the length of password 1. In each step, the indexOf method performs n steps in the worst case, depending on the length of username. However, these steps occur in parallel, and the loop iterates only over password1. Therefore, the total time complexity of this function will be O(m).

3 - boolean isBalancedPassword(String password1)



```
protected boolean isBalancedPassword(String password) {
    if (!isValidLength(password)) {
        System.out.println(x:"The password1 is invalid. It should have at least 8 characters.");
        return false;
    }
    if (!hasEnoughBrackets(password)) {
        System.out.println(x:"The password1 is invalid. It should have at least 2 brackets.");
        return false;
    }
    if (!hasLetters(password)) {
        System.out.println(x:"The password1 is invalid. It should have letters too.");
        return false;
    }
    if (!isBalancedBrackets(password)) {
        System.out.println(x:"The password1 is invalid. It should be balanced.");
        return false;
    }
    return true;
}
private boolean isValidLength(String password) {
    return password.length() >= 8;
}

private boolean hasEnoughBrackets(String password) {
    return password.replaceAll(regex:"[^\\[\\]\\(\\)\\{\\}]", replacement:"").length() >= 2;
}

private boolean hasLetters(String password) {
    return !password.replaceAll(regex:"[a-zA-Z]", replacement:"").equals(password);
}

private boolean isBalancedBrackets(String password) {
    Stack<Character> stack = new Stack<>();
    for (char ch : password.toCharArray()) {
        if (ch == '(' || ch == '{' || ch == '[') {
            stack.push(ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (stack.isEmpty()) {
                return false;
            }
            char openBracket = stack.pop();
            if ((ch == ')' && openBracket != '(') || (ch == '}' && openBracket != '{') || (ch == ']' && openBracket != '[')) {
                return false;
            }
        }
    }
    return stack.isEmpty();
}
```

Firstly, it checks if the password has a valid length. If the password is less than 8 characters, the method returns false and prints an error message.

Next, it checks if the password contains at least 2 brackets. If not, the method returns false and prints an error message.

Then, it checks if the password contains letters. A password consisting only of numbers or symbols without letters is also considered invalid. If the password doesn't contain any letters, the method returns false and prints an error message.

Finally, it checks if the brackets in the password are balanced. If the brackets are not balanced, the method returns false and prints an error message.

If all of these checks are successful, the method returns true, indicating that the password is valid.

The code also includes some helper methods like "isValidLength", "hasEnoughBrackets", "hasLetters", and "isBalancedBrackets" to perform the relevant checks. For example, "isBalancedBrackets" method checks if the brackets in the password are balanced or not, using a stack data structure.The replaceAll method is called three times, each with a time complexity of O(m), where m is the length of password1. In total, these operations have a time complexity of O(m).

The toCharArray method is called on passwordWithoutLetters, which has a time complexity of O(m) since it iterates through the entire string to create the character array.

The for loop iterates through the passwordWithoutLetters character array, which has a length of at most m. For each character, it performs constant-time operations (pushing or popping from the stack and some basic comparisons). So, the for loop has a time complexity of O(m).

Considering all these operations, the overall time complexity of the isBalancedPassword function is O(m), where m is the length of the input password1.

4 - boolean isPalindromePossible(String password1)

boolean isPalindromePossibleRecursive(String password, int[] charCounter, int index, int oddNumberCounter)

```
protected boolean isPalindromePossible(String password1) {
    if(password1.isEmpty()) {
        return false;
    }
    String passwordWithoutBrackets = password1.replaceAll(regex:"[\\[\\]\\(\\)\\{\\}]", replacement:"");
    int[] charCounter = new int[60];
    return isPalindromePossibleRecursive(passwordWithoutBrackets, charCounter, index:0, oddNumberCounter:0);
}

private boolean isPalindromePossibleRecursive(String password, int[] charCounter, int index, int oddNumberCounter) {
    if (index >= password.length()) {
        if (oddNumberCounter > 1) {
            System.out.println(x:"The password1 is invalid. It should be possible to obtain a palindrome from the password1.");
            return false;
        }
        return oddNumberCounter <= 1;
    }
    int charIndex = password.charAt(index) - 65;
    charCounter[charIndex]++;
    int updatedoddNumberCounter;
    if (charCounter[charIndex] % 2 == 0) {
        updatedoddNumberCounter = oddNumberCounter - 1;
    }
    else {
        updatedoddNumberCounter = oddNumberCounter + 1;
    }
    return isPalindromePossibleRecursive(password, charCounter, index + 1, updatedoddNumberCounter);
}
```

First, the code checks if the password is empty. If it is, it returns false.

Next, it removes brackets and other specified characters ("[", "]", "(", ")", "{", "}") from the password and saves the remaining characters in a string.

Then, it defines a character counter array ("charCounter") to keep track of the number of occurrences of each character in the password.

Finally, it calls a special helper method named "isPalindromePossibleRecursive". This method counts the occurrences of the characters in the password and determines whether the password can be a palindrome.

The "isPalindromePossibleRecursive" method iterates through each character in the password, incrementing the corresponding count in the "charCounter" array. It also updates an "oddNumberCounter" variable by checking if the count of each character is odd or even.

After iterating through all the characters, the method determines whether the password can be a palindrome based on the value of the "oddNumberCounter" variable. If there are more than one characters with odd count, it returns false and prints an error message. Otherwise, it returns true, indicating that the password can be a palindrome.

In summary, the "isPalindromePossible" method determines whether a password can be a palindrome by counting the occurrences of the characters and returning true or false based on the result.

The isPalindromePossible function time complexity:

Creating the passwordWithoutBrackets string takes O(m) time, where m is the length of password1.

Initializing the charCounter array takes O(1) time, since the size of the array is constant (50).

The isPalindromePossibleRecursive function processes each character in the passwordWithoutBrackets string once. Since there are m characters in the passwordWithoutBrackets string, the time complexity of the recursive function is O(m).

Taking both parts into account, the total time complexity of the function is O(m) + O(1) + O(m) = O(2m) + O(1). The dominant term here is O(2m), which can be simplified to O(m). Therefore, the overall time complexity of this function is O(m), where m is the length of the password1 string.

5 - boolean isExactDivision(int password2, int[] denominations)

boolean isExactDivisionRecursive(int remaining, int[] denominations, int currentIndex)

```java
protected boolean isExactDivision(int password2, int[] denominations) {
    if(password2 < 10 || password2 > 10000) {
        System.out.println(x:"The password2 is invalid. It should be between 10 and 10000.");
        return false;
    }

    boolean result = isExactDivisionRecursive(password2, denominations, currentIndex:0);

    if(!result) {
        System.out.println(x:"The password2 is invalid. It is not compatible with the denominations.");
    }

    return result;
}

private boolean isExactDivisionRecursive(int remaining, int[] denominations, int currentIndex) {
    if (remaining == 0) {
        return true;
    }
    if (remaining < 0 || currentIndex >= denominations.length) {
        return false;
    }

    // Case 1: Include the current denomination in the sum
    boolean useCurrentDenomination = isExactDivisionRecursive(remaining - denominations[currentIndex], denominations, currentIndex);

    // Case 2: Skip the current denomination and move to the next one
    boolean skipCurrentDenomination = isExactDivisionRecursive(remaining, denominations, currentIndex + 1);

    return useCurrentDenomination || skipCurrentDenomination;
}
```

This code checks if an integer password2 can be exactly divided by some denominations. First, it checks if password2 is valid, i.e., if it is between 10 and 10000. If not, it returns false. Then, it calls the recursive function isExactDivisionRecursive, which takes three arguments: the remaining value to be divided, an array of denominations, and the current index in the denominations array.

The function starts by checking if the remaining value is zero, in which case it returns true since the division is exact. If the remaining value is negative or the current index is out of range, the function returns false since the division is not exact.

Then, it proceeds to two cases: either include the current denomination in the sum, or skip the current denomination and move to the next one. It recursively calls the function for both cases and returns true if either of them returns true, meaning that an exact

division is possible with the given denominations. If none of the cases return true, the function returns false.

The isExactDivisionRecursive function recursively explores all possible combinations of denominations until it either finds a combination that results in the exact division of remaining or all combinations have been exhausted. Since the function explores all possible combinations, its time complexity is O(2^n), where n is the number of denominations.

INPUT & OUTPUT

```java
class Test {
    Run | Debug
    public static void main(String[] args) {

        MuseumSecuritySystem test1 = new MuseumSecuritySystem(username:"sibelgulmez", password1:"[rac()ecar]", password2:74);
        System.out.println(x:"Test1:");
        test1.test();

        MuseumSecuritySystem test2 = new MuseumSecuritySystem(username:"", password1:"[rac()ecar]", password2:74);
        System.out.println(x:"Test2:");
        test2.test();

        MuseumSecuritySystem test3 = new MuseumSecuritySystem(username:"sibel1", password1:"[rac()ecar]", password2:74);
        System.out.println(x:"Test3:");
        test3.test();

        MuseumSecuritySystem test4 = new MuseumSecuritySystem(username:"sibel", password1:"pass[]", password2:74);
        System.out.println(x:"Test4:");
        test4.test();

        MuseumSecuritySystem test5 = new MuseumSecuritySystem(username:"sibel", password1:"abcdabcd", password2:74);
        System.out.println(x:"Test5:");
        test5.test();

        MuseumSecuritySystem test6 = new MuseumSecuritySystem(username:"sibel", password1:"[[[[]]]]", password2:74);
        System.out.println(x:"Test6:");
        test6.test();

        MuseumSecuritySystem test7 = new MuseumSecuritySystem(username:"sibel", password1:"[no](no)", password2:74);
        System.out.println(x:"Test7:");
        test7.test();

        MuseumSecuritySystem test8 = new MuseumSecuritySystem(username:"sibel", password1:"[rac()ecar]]", password2:74);
        System.out.println(x:"Test8:");
        test8.test();

        MuseumSecuritySystem test9 = new MuseumSecuritySystem(username:"sibel", password1:"[rac()ecars]", password2:74);
        System.out.println(x:"Test9:");
        test9.test();

        MuseumSecuritySystem test10 = new MuseumSecuritySystem(username:"sibel", password1:"[rac()ecar]", password2:5);
        System.out.println(x:"Test10:");
        test10.test();

        MuseumSecuritySystem test11 = new MuseumSecuritySystem(username:"sibel", password1:"[rac()ecar]", password2:35);
        System.out.println(x:"Test11:");
        test11.test();
    }
}
```

```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework$ cd Homework4/
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/Homework4$ javac *.java
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/Homework4$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework$ java Homework4.Test
Test1:
The username and passwords are valid. The door is opening, please wait..
Welcome sibelgulmez!
Test2:
The username is invalid. It should have at least 1 character.
Test3:
The username is invalid. It should have letters only
Test4:
The password1 is invalid. It should have at least 8 characters.
Test5:
The password1 is invalid. It should have at least 2 brackets.
Test6:
The password1 is invalid. It should have letters too.
Test7:
The password1 is invalid. It should have at least 1 character from the username.
Test8:
The password1 is invalid. It should be balanced.
Test9:
The password1 is invalid. It should be possible to obtain a palindrome from the password1.
Test10:
The password2 is invalid. It should be between 10 and 10000.
Test11:
The password2 is invalid. It is not compatible with the denominations.
```