GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 222 - SPRING 2023

HOMEWORK 4 REPORT

ETHEM MENGAASLAN

200104004016

THE DESIGN:

All methods are implemented as member methods of a class called SecurityControlMethods.

I have used wrapper methods in some cases :

> isPalindromePossible(): A wrapper for the recursive isPalindromePossibleRec() method, to simplify the parameters for the end user of the method and to remove the brackets from the string before checking it for palindrome.
>
> isExactDivision(): A wrapper for the recursive isExactDivision() method, to simplfiy the parameters, also checks the password for being in between the range 10-10000 using password2Checker method at the beginning.

I also have implemented some extra methods to keep the code clean such as:

> removeBrackets(): Removes the brackets from the input string, to provide a bracket-free string for some methods like palindrome method.
>
> password2Checker(): I didn’t want to write an if statement to check the password2 for being in the correct range, so I implemented this method and used it in necessary methods.

All the methods are called from a main method called login(), which takes inputs in the format as stated in the assignment pdf.

The login() method does the necessary controls, and I tried to keep the control order from the lowest complexity to the highest complexity in order to make the program more efficient. (For example, the isExactDivision method with the complexity of O(2^n) is checked only if all of the other controls are passed.)

Here is the hierarchy of the controls from the first control to last control:

1) Username is checked for
    1.1) Length
    1.2) Consisting of only letters
2) Password1 is checked for
    2.1) Length
    2.2) Number of brackets
    2.3) Being balanced / unbalanced
    2.4) Containing letters from the username
    2.5) Being able to form a palindrome
3) Password2 is checked for
    3.1) Being in between the 10-10000 range
    3.2) Being compatible with the denominations

SAMPLE INPUTS & OUTPUTS:

```
myObj.login("sibelgulmez","[rac()ecar]",74);
```
The username and the passwords are valid. The door is opening, please wait...

```
myObj.login("", "[rac()ecar]", 74);
```
The username is invalid due to being shorter than 1 character.

```
myObj.login("sibel1", "[rac()ecar]", 74);
```
The username is invalid due to including other characters other than letters.

```
myObj.login("sibel", "pass[]", 74);
```
The password1 is invalid due to being shorter than 8 characters.

```
myObj.login("sibel", "abcdabcd", 74);
```
The password1 is invalid due to having less than 2 brackets.

```
myObj.login("sibel", "[[[[]]]]", 74);
```
The password1 is invalid due to not containing a letter(s) from the username.

```
myObj.login("sibel", "[no](no)", 74);
```
The password1 is invalid due to not containing a letter(s) from the username.

```
myObj.login("sibel", "[rac()ecar]]", 74);
```
The password1 is invalid due to the password being unbalanced.

```
myObj.login("sibel", "[rac()ecars]", 74);
```
The password1 is invalid due to not being able to become a Palindrome.

```
myObj.login("sibel", "[rac()ecar]", 5);
```
The password2 is invalid due to not being between 10 and 10000

```
myObj.login("sibel", "[rac()ecar]", 35);
```
The password2 is invalid due to not being compatible with the denominations

THE TIME COMPLEXITY ANALYSIS OF THE METHODS:

```java
protected boolean checkIfValidUsername(String username){

    if (username.length() == 0) {
        // If the username length is 0, it is invalid.
        System.out.println("The username is invalid due to being shorter
than 1 character.");
        return false;
    }

    if ( Character.isLetter(username.charAt(0)) == false ){
        System.out.println("The username is invalid due to including other
characters other than letters.");
        return false;
    }
    // If the username passed the if statement above, and consists of only
1 characters. It means it is valid.
    if ( username.length() == 1 ){
        return true;
    }

    return checkIfValidUsername(username.substring(1)); // The method is
called again with the first letter shifted.

}
```

The algorithm checks the first character of the username and returns false if it is not a letter. If the first character is a letter, it checks if the length of the username is 1, if it is; it returns true. Otherwise, it recursively calls itself with the first character of the username removed.

In the worst case, the algorithm will need to iterate through the entire username string before returning its validity. Therefore, the time complexity is O(n).

```java
protected boolean containsUserNameSpirit(String username, String password){

    // Used a object wrapper for char (Character).
    Stack<Character> myStack = new Stack<>();
    boolean containsUname = false;

    for(int i=0;i<password.length();i++){
        myStack.push(password.charAt(i));
    }

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

        if( myStack.contains(username.charAt(i)) ){
            containsUname = true;
            return containsUname;
        }

    }

    System.out.println("The password1 is invalid due to not containing a
letter(s) from the username.");
    return containsUname;

}
```

The first for loop has a time complexity of O(n), as it iterates through the password and pushes each character into the stack.

The second loop iterates through each character in the username string and checks whether it is present in the myStack object using the contains() method. The time complexity of the contains() method is O(n) in the worst case, where n is the number of elements in the myStack object. In this case, the maximum number of elements in the myStack object is n, so the time complexity of the contains() method becomes O(n) in the worst case. Since this operation is performed the length of the username (let's call it "m".) times, the time complexity of this loop is O(n * m)

So, the overall time complexity of this code block is O(n * m).

```
protected boolean isBalancedPassword( String password1 ){

    if(password1.length() < 8){
        System.out.println("The password1 is invalid due to being shorter
than 8 characters.");
        return false;
    }

    Stack<Character> myStack = new Stack<>();

    String opens = "{[(";
    String closes = "}])";

    int bracketCounter = 0;

    for(int i=0;i<password1.length();i++){
        if(opens.indexOf(password1.charAt(i)) != -1 ||
closes.indexOf(password1.charAt(i)) != -1 ){
            bracketCounter++;
        }
    }

    if(bracketCounter < 2 ){
        System.out.println("The password1 is invalid due to having less
than 2 brackets.");
        return false;
    }

    for(int i=0;i<password1.length();i++){
        if( opens.indexOf( password1.charAt(i) ) >= 0 ){
            myStack.push(password1.charAt(i));
            //System.out.println("open found index: " + i);
        }else if( closes.indexOf(password1.charAt(i)) >= 0 ){
            //System.out.println("close found index: " + i);
            if(myStack.empty() == false &&  opens.indexOf(myStack.peek())
== closes.indexOf(password1.charAt(i))){
                myStack.pop();
            }else{
                System.out.println("The password1 is invalid due to the
password being unbalanced.");
                return false;
            }
        }
    }

    if(myStack.size() == 0){
        //System.out.println("Statement balanced!");
        return true;
    }else{
        System.out.println("The password1 is invalid due to the password
being unbalanced.");
        return false;
        }

}
```

The first if statement takes constant time O(1) to check if the length of the password is less than 8, so it can be ignored.

The for loop that iterates through the characters of the password to count the number of brackets takes O(n) time.

The second if statement also takes constant time to check if the bracket count is less than 2, so it is O(1).

The second for loop that iterates through the characters of the password to check if the brackets are balanced takes O(n) time in the worst case. Each iteration of the loop takes constant time to check if the character is an opening bracket, closing bracket or something else, and performs operations on the stack. Since each operation on the stack takes constant time, the overall time complexity of the loop is O(n).

Therefore, the time complexity of the entire algorithm is O(n).

```
private boolean isPalindromePossibleRec(String password1) {
    // Base case: if the string consists of only one letter, it's a
palindrome
    if (password1.length() == 1 || password1.length() == 0) {
        return true;
    }

    // Recursive case: find a character that has a matching pair
    for (int i = 0; i < password1.length(); i++) {
        char c = password1.charAt(i);
        int pairIndex = password1.indexOf(c, i + 1);

        if (pairIndex != -1) {
            // If a pair character is found, remove the 2 occurences of the
pair characters from the string and call the method again with the modified
string.
            String newString =
password1.replaceFirst(String.valueOf(c),"");
            newString = newString.replaceFirst(String.valueOf(c),"");
            return isPalindromePossibleRec(newString);
        }
    }

    // If no matching pair is found, the string can only be a palindrome if
it has a single character
    //return password1.length() == 1;
    return false;
}
```

The time complexity of this algorithm depends on the input string. Let’s assume that n be the length of the input string.

In the worst case scenario, where there are no matching pairs in the string, the algorithm will iterate through the string once with a for loop, which has a time complexity of O(n). This would be the case if the input string has distinct characters, for example "abcde".

In the best case scenario, the algorithm would find a matching pair at the first iteration of the for loop. In this case, the time complexity would be O(1).

In the average case scenario, the time complexity can be estimated as O(n/2), since the for loop only iterates through the first half of the string on average.

Since the time complexity can vary depending on the input string, we can express the time complexity as O(f(n)), where f(n) depends on the input string.

```java
public boolean isPalindromePossible(String password1){

        String passwordWithoutBrackets = removeBrackets(password1);

        if(isPalindromePossibleRec(passwordWithoutBrackets) == false){
            System.out.println("The password1 is invalid due to not being able to become a Palindrome.");
            return false;
        }

        return true;


}
```

First of all, the isPalindromePossible method is a wrapper method for the recursive isPalindromePossibleRec method. It removes the brackets from the input string with the removeBrackets method, which has a time complexity of O(n).

The isPalindromePossible method first calls the removeBrackets method, which takes O(n) time complexity. Then, it calls the isPalindromePossibleRec method, whose time complexity depends on the input string, as explained above.

Overall, the time complexity of the isPalindromePossible method can be expressed as O(f(n)) + O(n), where f(n) depends on the input string and can vary from O(1) to O(n), and this leads us to the time complexity of O(n) for the whole method.

```
private boolean isExactDivisionRec(int remaining, int[] denominations, int
index) {
    if (remaining == 0) {
        // If the remaining number is zero, it means that we have found
valid coefficents to give the goal password.
        return true;
    }
    if (index == denominations.length) {
        // If we have used all denominations, and the remainder is still
not zero, there is no valid coeficient combination
        return false;
    }
    for (int i = 0; i <= remaining / denominations[index]; i++) {
        // For each possible coefficient for the current denomination,
recursively check if the remainder can be further subtracted.
        if (isExactDivisionRec(remaining - i * denominations[index],
denominations, index + 1)) {
            return true;
        }
    }
    return false;
}
```

The time complexity of this algorithm is exponential, O(2^n), where n is the length of the denominations array.

The reason for this is because the algorithm uses a recursive approach that branches out into multiple subproblems. At each level of recursion, the algorithm tries every possible coefficient for the current denomination by looping from 0 up to the maximum possible coefficient (remaining / denominations[index]). Each coefficient choice creates a new subproblem with a smaller remaining value to solve recursively, which results in a branching factor of (remaining / denominations[index]) at each level of the recursion tree.

Therefore, the number of recursive calls grows exponentially with the size of the input array denominations, and the algorithm may explore a large number of subproblems before reaching a solution or determining that one does not exist. In the worst case, where none of the coefficients work for any of the denominations, the algorithm will explore all possible subproblems, resulting in a time complexity of O(2^n).

```java
private String removeBrackets(String password1){

    password1 = password1.replace("(","");
    password1 = password1.replace(")","");
    password1 = password1.replace("[","");
    password1 = password1.replace("]","");
    password1 = password1.replace("{","");
    password1 = password1.replace("}","");

    return password1;

}
```

The time complexity of the removeBrackets function is O(n), where n is the length of the input string password1. This is because the replace method iterates through the string each time it Is called, and it is called 6 times. 6 * O(n) leads us to O(n).

```java
private boolean password2Checker(int password2){
    if( password2 >= 10 && password2 <= 10000){
        return true;
    }
    System.out.println("The password2 is invalid due to not being between
10 and 10000");
    return false;
}
```

The time complexity of the password2Checker method is O(1), which means it has a constant time complexity.

This is because the method only performs a simple comparison of the input password2 with two constant values, and then returns a boolean value based on the result of the comparison. The method does not have any loops or recursive calls, and the amount of time it takes to execute does not depend on the size of the input password2.