GEBZE TECHNICAL UNIVERSITY

DATA STRUCTURE AND ALGORITHM

HOMEWORK -4 REPORT

NAME: SUHA BERK KUKUK

STUDENT ID: 225008003102

Time Complexity Analysis

1)

```
public void stackPush()

    {
        int size = getSize();
        for(int i=0; i<size ; i++)
        {
            stack.push(this.getUsername().charAt(i));
        }
    }
```

The time complexity of this method is O(n), where n is the length of the username.

The method first calls the getSize() method, which should have a time complexity of O(1) if it simply returns a stored variable. The for loop then iterates n times, where n is the length of the username. For each iteration, the charAt() method is called, which has a time complexity of O(1) for an individual character. Therefore, the time complexity of the entire for loop is O(n).

Inside the loop, the push() method is called, which has an amortized time complexity of O(1) for adding an element to the top of a stack. This is because the implementation of push() typically uses an underlying array or linked list, which allows for constant-time insertion at the top. Therefore, the time complexity of the push() method inside the loop is O(1).

Since the for loop has a time complexity of O(n), and the push() method inside the loop has a time complexity of O(1), the overall time complexity of the stackPush() method is O(n).

2)

```
public boolean checkIfValidUsernameRecursive(String username)
    {
        if(username.isEmpty())
            return false;
        else if(!username.matches("[a-zA-Z]+"))
            return false;
        else if(username.length() ==1)
            return true;
        else
            return 
checkIfValidUsernameRecursive(username.substring(1));

    }
```

The time complexity of this method is O(n), where n is the length of the input username.

The method first checks if the username is empty, which has a time complexity of O(1). If it is empty, the method immediately returns false.

If the username is not empty, the method then checks if it consists entirely of letters using a regular expression pattern matching operation, which has a time complexity of O(n) in the worst case, where n is the length of the input string.

If the username is valid so far, the method checks if the length of the username is 1, which has a time complexity of O(1). If it is, the method returns true.

If the length of the username is greater than 1, the method recursively calls itself with a substring of the username starting from the second character. The length of the substring is n-1, where n is the length of the original username. Therefore, the method will be called recursively n-1 times in the worst case, each time with a substring of length n-2, n-3, and so on, down to 1. The total time complexity of these recursive calls is the sum of 1 + 2 + 3 + ... + (n-1), which is equivalent to n(n-1)/2. In the worst case, the time complexity of these recursive calls is O(n^2).

Therefore, the overall time complexity of the checkIfValidUsernameRecursive() method is O(n^2), due to the worst-case scenario of the recursive calls.

3)

```
public boolean containUserNameSpirit( Password1 password)
    {

        //Create Stack Structure
        stackPush(); //Stack consist username of letter
        password.stackPush(password.stackPasword); //stack consist
password of letter

        while (!password.stackPasword.isEmpty()) {
            char c = password.stackPop(password.stackPasword);
            if (username.contains(String.valueOf(c))) {
                return true;
            }
        }
        return false;
    }
```

The while loop iterates over each character in the password stack and has a maximum of m iterations. Inside the loop, the stackPop() method of the Password1 class is called, which should have a time complexity of O(1) for an individual character. Then, the contains() method of the String class is called with a single character, which has a time complexity of O(n) in the worst case, where n is the length of the username. Therefore, the time complexity of the while loop is O(m*n).
The overall time complexity of the containUserNameSpirit() method is therefore O(n+m+nm), which can be simplified to O(nm), since n and m are usually comparable in size.

4)

```
protected boolean isBalancedPassword()
    {
        for (int i = 0; i < password.length(); i++) {
            char c = password.charAt(i);
            if (isOpenBracket(c)) {
                stackBalance.push(c);
            } else if (isCloseBracket(c)) {
                if (stackBalance.empty() ||
!isMatchingBracket(stackBalance.peek(), c)) {
                    return false;
                }
                stackBalance.pop();
            }
        }
        return stackBalance.empty();
    }
```

Inside the loop, the charAt() method is called to get the character at the current index, which has a time complexity of O(1) for an individual character.

Then, the isOpenBracket() method is called, which has a time complexity of O(1) since it simply checks if the input character is an open bracket character.

If the input character is an open bracket, the push() method of the stackBalance stack is called, which has an amortized time complexity of O(1) for adding an element to the top of a stack.

If the input character is a close bracket, the method calls the isCloseBracket() method, which has a time complexity of O(1) since it simply checks if the input character is a close bracket character.

If the input character is a close bracket, the method then checks if the stackBalance stack is empty or if the top element of the stack does not match the current character using the isMatchingBracket() method. These operations have a time complexity of O(1).

If the top element of the stackBalance stack matches the current character, the pop() method of the stackBalance stack is called, which has an amortized time complexity of O(1) for removing an element from the top of a stack.

After the loop completes, the method returns true if the stackBalance stack is empty, which has a time complexity of O(1) using the empty() method.

Therefore, the overall time complexity of the isBalancedPassword() method is O(n).

5)

```
protected boolean isPalindromePossible(Stack<Character> stack)
    {

        int sizePassword = stack.capacity();;
        stackPush(stack);
        char x = stackPop(stack);
        for(int i=1; i<sizePassword; i++)
        {
            if(x==stackPeek(stack))
            {
                stackPop(stack);
                this.palindrome += 2;
                return isPalindromePossible(stack);
            }

        }
        if (this.palindrome %2 <=1)
            return true;
        else
            return false;
    }
```

The stackPeek() method is called inside the loop, which has a time complexity of O(1) for accessing the top element of the stack without removing it.

The loop iterates over the stack and compares each element with the top element. If the elements match, it pops the top element from the stack and increments the palindrome variable by 2, and then calls the isPalindromePossible() method recursively. The number of recursive calls is at most equal to the size of the stack, since the method pops one element from the stack in each iteration of the loop. Therefore, the time complexity of the recursive calls is O(n).

Finally, the method checks if the value of palindrome is even or odd, and returns true if it is even and false if it is odd. This operation has a time complexity of O(1).

Therefore, the overall time complexity of the isPalindromePossible() method is O(n), where n is the size of the input stack

6)

```
private boolean exactDivisionHelper(int remainder, int[] denominations,
int index) {
        // Base case: the remainder is zero, so the function

  have found a valid combination of denominations
        if (remainder == 0) {
            return true;
        }

        // Recursive case: try subtracting each denomination from the
remainder
        for (int i = index; i < denominations.length; i++) {
            int denomination = denominations[i];
            if (denomination <= remainder) {
                boolean result = exactDivisionHelper(remainder -
denomination, denominations, i);
                if (result) {
                    return true;
                }
            }
        }

        // If the function reach this point, the function  have tried
all possible combinations without success

        return false;
    }
```

The method uses a recursive approach to find a valid combination of denominations that add up to the given remainder. At each step, the method subtracts a denomination from the remainder and recursively calls itself with the updated remainder and the same set of denominations. The method continues this process until the remainder becomes zero, at which point it returns true to indicate that a valid combination has been found. If none of the combinations results in a remainder of zero, the method returns false to indicate failure.

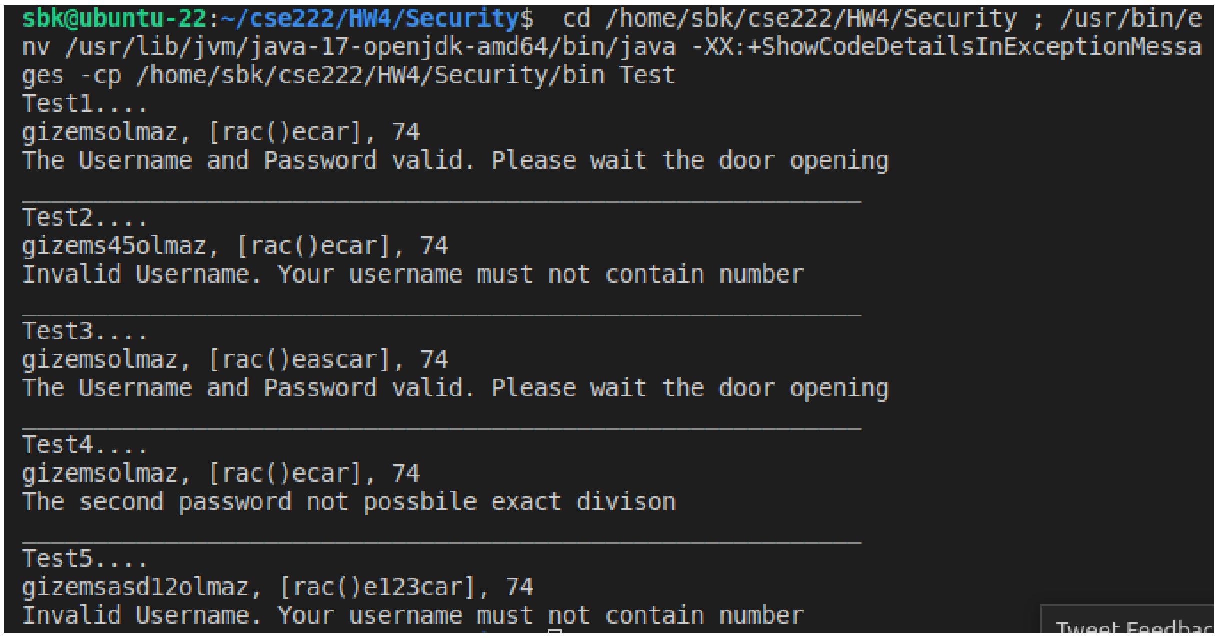
The worst-case time complexity of this method is O(k^n), where k is the maximum value of a denomination and n is the number of denominations. This occurs when the method has to explore all possible combinations of denominations to find a valid one.

Overall, the time complexity of this method can be expressed as O(k^n), where k is the maximum value of a denomination and n is the number of denominations.

Test Result And Appraoch

```
    //Create username, password1, password2
        Username user1 = new Username("gizemsolmaz");
        Password1 pass1 = new Password1("[rac()ecar]");
        Password2 pass2 = new Password2(74);

        //test params
        System.out.println("Test1....");
        System.out.println(user1.getUsername() + ", " +
pass1.getpassword() + ", " + pass2.getpassword() );
        test(user1, pass1, pass2);


System.out.println("___________________________________________________
________");

        //Create username, password1, password2
        Username user2 = new Username("gizems45olmaz");
        Password1 pass12 = new Password1("[rac()ecar]");
        Password2 pass22 = new Password2(74);

        //test params
        System.out.println("Test2....");
        System.out.println(user2.getUsername() + ", " +
pass12.getpassword() + ", " + pass22.getpassword() );
        test(user2, pass12, pass22);


System.out.println("___________________________________________________
________");
        //Create username, password1, password2
        Username user3 = new Username("gizemsolmaz");
        Password1 pass13 = new Password1("[rac()eascar]");
        Password2 pass23 = new Password2(74);

        //test params
        System.out.println("Test3....");
        System.out.println(user3.getUsername() + ", " +
pass13.getpassword() + ", " + pass23.getpassword() );
        test(user3, pass13, pass23);


System.out.println("___________________________________________________
________");
        //Create username, password1, password2
        Username user4 = new Username("gizemsolmaz");
        Password1 pass14 = new Password1("[rac()ecar]");
        Password2 pass24 = new Password2(10);
        //test params
        System.out.println("Test4....");
        System.out.println(user4.getUsername() + ", " +
pass1.getpassword() + ", " + pass2.getpassword() );
        test(user4, pass14, pass24);



System.out.println("___________________________________________________________
________");

        //Create username, password1, password2
        Username user5 = new Username("gizemsasd12olmaz");
        Password1 pass15 = new Password1("[rac()e123car]");
        Password2 pass25 = new Password2(74);

        //test params
        System.out.println("Test5....");
        System.out.println(user5.getUsername() + ", " +
pass15.getpassword() + ", " + pass25.getpassword() );
        test(user5, pass15, pass25);
```

Result :



```
sbk@ubuntu-22:~/cse222/HW4/Security$  cd /home/sbk/cse222/HW4/Security ; /usr/bin/e
nv /usr/lib/jvm/java-17-openjdk-amd64/bin/java -XX:+ShowCodeDetailsInExceptionMessa
ges -cp /home/sbk/cse222/HW4/Security/bin Test
Test1....
gizemsolmaz, [rac()ecar], 74
The Username and Password valid. Please wait the door opening
____________________________________________________________
Test2....
gizems45olmaz, [rac()ecar], 74
Invalid Username. Your username must not contain number
____________________________________________________________
Test3....
gizemsolmaz, [rac()eascar], 74
The Username and Password valid. Please wait the door opening
____________________________________________________________
Test4....
gizemsolmaz, [rac()ecar], 74
The second password not possbile exact divison
____________________________________________________________
Test5....
gizemsasd12olmaz, [rac()e123car], 74
Invalid Username. Your username must not contain number
```

i) checkIfValidUsername

I checked username is empty, if it empty the program return false. The username skip this condition i check there are any number in username if it exist so i will return false. After that i checked lengt of username if it is 1, it will return and these condition not provide any result i return base function again but with creating new substring from password. These steps continue until become substring lenght is 1.

ii) containsUserNameSpirit

The function creates a new stack to hold the characters of the password.I iterate over each character of the password, starting from the end, and push it onto the stack.I pop characters off the stack one by one and check if the username contains the character. If it does, I return true because I have found a match.If I have checked all characters of the password and found no match, I return false.

iii) isBalancedPassword

I iterate over each character in the password string.If the character is an open bracket (i.e., '(', '{', or '['), I push it onto the stack. If the character is a close bracket (i.e., ')', '}', or ']'), I check if the stack is empty or if the top of the stack contains a matching open bracket. If the stack is empty or the top of the stack doesn't contain a matching open bracket, I return false because the password is not balanced. Otherwise, I pop the top of the stack because the open bracket and the close bracket form a matching pair. After iterating over all the characters, if the stack is empty, it means that all open brackets have a matching close bracket, so the password is balanced. Otherwise, the password is not balanced because there are open brackets without a matching close bracket.

iv) isPalindromePossible

I first remove the brackets from the password string to ignore them while computing the function. I create a new string that only contains letters I count the frequency of each letter in the password string by iterating over each character and updating a map that stores the frequency of each letter.

I count the number of odd-frequency letters in the password string by iterating over the values of the map and checking if they are odd. If a value is odd, it means that the corresponding letter appears an odd number of times in the password string. A palindrome is possible if and only if there are at most 1 odd-frequency letters. This is because a palindrome can have at most 1 letter with an odd frequency (which will be the middle letter), while all other letters must appear an even number of times to form a symmetric string.

v) isExactDivision

I start by calling the exactDivisionHelper function with the given password and list of denominations. I also pass an index of 0 to indicate that I haven't used any denominations yet. The exactDivisionHelper function checks if the remainder is zero, which means I have found a valid combination of denominations that adds up to the password. If the remainder is zero, I return true. If the remainder is not zero, I try subtracting each denomination from the remainder recursively. I start from the current index to avoid using the same denomination multiple times. If a recursive call returns true, it means that I have found a valid combination of denominations, so I return true immediately. If I have tried all possible combinations without success, the program will return false