Time Complexities

Batuhan BASAK

1 checkIfValidUsername

```
private boolean checkIfValidUsername(String username){
    if (username.length() < 1){
        System.out.println("The username is invalid. " + "
            It should have at least 1 character.");
        return false;
    }
    if (!isOnlyAlphaChar(username)){
        System.out.println("The username is invalid. " + "
            It should have letters only.");
        return false;
    }
        return true;
}


private static boolean isOnlyAlphaChar(String username){
    if (username.length() == 0)
        return true;
    else if (!Character.isAlphabetic(username.charAt(username.length() - 1)))
        return false;
    else
        return isOnlyAlphaChar(username.substring(0, username.length() - 1));
}
```

Time complexity of method isOnlyAlphaChar is $O(n)$. Time complexity of first if condition in checkIfValidUsername $\theta(1)$. The second one takes $O(n)$.

$$T(n) = O(n) + \theta(1) = O(n)$$

Thus time complexity of checkIfValidUsername is $O(n)$.

2 containsUserNameSpirit

```
private boolean containsUserNameSpirit(String username, String password1){
    if (usernameChars.isEmpty())
        fillStack(username, usernameChars);
    if (password1Chars.isEmpty())
        fillStack(password1, password1Chars);

    char c;
    while (!usernameChars.isEmpty()){
        c = usernameChars.peek();
        while(!password1Chars.isEmpty()){
            if (c == password1Chars.pop()){
                usernameChars.clear();
                fillStack(username, usernameChars);
                password1Chars.clear();
                fillStack(password1, password1Chars);
                return true;
            }
        }
        fillStack(password1, password1Chars);
        usernameChars.pop();
    }
    // reset stack.
    fillStack(username, usernameChars);
    return false;
}


private void fillStack(String string, Stack<Character> stack){
    if (string.length() != 0){
        stack.push(string.charAt(string.length() - 1));
        fillStack(string.substring(0, string.length() - 1), stack);
    }
}
```

The time complexity of fillStack is $O(n)$. Assume the length of string username is n and length of string password1 is m. The time complexity of outer while loop is $O(n \cdot m)$. Hence $O(n \cdot m) + O(n) = O(n \cdot m)$, the time complexity of containsUserNameSpirit is $O(n \cdot m)$.

3 isBalancedPassword

```
private boolean isBalancedPassword(String password1){
    if (password1Chars.isEmpty())
        fillStack(password1, password1Chars);
    if (!checkBracketsMatching(password1)){
        return false;
    }
    return true;
}


private boolean checkBracketsMatching(String string){
    if (string.isEmpty()){
        return false;
    }

    Stack<Character> brackets = new Stack<>();
    char c;
    for (int i = 0; i < string.length(); i++){
        c = string.charAt(i);

        if (c == '(' || c == '[' || c == '{'){
            brackets.push(c);
        } else if (c == ')' || c == ']' || c == '}'){
            if (brackets.isEmpty())
                return false;
            if ((brackets.peek() == '(' && c != ')') ||
                    (brackets.peek() == '[' && c != ']') ||
                    (brackets.peek() == '{' && c != '}') )
                return false;
            else
                brackets.pop();
        }
    }
    return true;
}
```

The time complexity of checkBracketsMatching is $O(n)$. Thus time complexity of isBalancedPassword is $O(n)$.

4 isPalindromePossible

```
private boolean isPalindromePossible(String password1){
    nonRepeatedString = new String();
    getNonRepeatedString(password1);
    int oddCount = getCountOfOddChars(password1, nonRepeatedString);
    return oddCount <= 1;
}


private int getCountOfOddChars(String s, String charSet){
    if (charSet.isEmpty())
        return 0;
    int len = charSet.length();
    char c = charSet.charAt(len - 1);
    if (countChar(s, c) % 2 != 0)
        return 1 + getCountOfOddChars(s, charSet.substring(0, len - 1));
    return getCountOfOddChars(s, charSet.substring(0, len - 1));
}


private int countChar(String s, char c){
    if (s.isEmpty())
        return 0;
    int len = s.length();
    if (s.charAt(len - 1) == c)
        return 1 + countChar(s.substring(0, len - 1), c);
    else
        return countChar(s.substring(0, len - 1), c);
}


private void getNonRepeatedString(String s){
    if (s.isEmpty())
        return;
    int len = s.length();
    char c = s.charAt(len - 1);
    if (c != '(' && c != '[' && c != '{' &&
            c != ')' && c != ']' && c != '}' &&
            nonRepeatedString.indexOf(c) == -1)
        nonRepeatedString += c;
    getNonRepeatedString(s.substring(0, len - 1));
}
```

Time Complexity of getNonRepeatedString is $O(n)$. Time complexity of coutChar is $O(n)$. Because each if condition it uses method countChar the time complex-

ity of getCountOfOddChars is $O(n) \cdot O(n) + \theta(1) = O(n^2)$. The time complexity of the method isPalindromePossible is $\theta(1) + O(n) + O(n^2) + \theta(1) = O(n^2)$.

5 isExactDivision

```
private boolean isExactDivision(int password2){
    int [] denominations = new int[3];
    denominations[0] = 4;
    denominations[1] = 17;
    denominations[2] = 29;

    return isExactDivision(password2, denominations);
}


private boolean isExactDivision(int password2, int [] denominations){
    return multA(0, password2, denominations);
}


private boolean multA(int a, int password2, int [] denominations){
    if (a*denominations[0] > password2){
        return false;
    } else {
        return multB(a, 0, password2, denominations);
    }
}


private boolean multB(int a, int b, int password2, int [] denominations){
    if (b*denominations[1] > password2){
        return false;
    } else {
        return multC(a, b+1, 0, password2, denominations);
    }
}


private boolean multC(int a, int b, int c, int password2, int [] denominations){
    if (c*denominations[2] > password2 ){
        if (a * denominations[0] <= password2) {
            return multA(a+1, password2, denominations);
        }
        return false;
    } else if (a*denominations[0] + b*denominations[1] + c*denominations[2] == password2){
```

```
        return true;
    } else {
        return multC(a, b, c+1, password2, denominations);
    }
}
```

Both isExactDivision methods have same because first 4 line in first method doesn't effect the complexity. Assume

$$n = \frac{password2}{denominations[0]}, m = \frac{password2}{denominations[1]}, k = \frac{password2}{denominations[2]} \quad (1)$$

multA time complexity is $O(n \cdot m \cdot k)$.