

**GTU Department of Computer Engineering**  
**CSE 222 - Spring 2023**  
**Homework 4- Report**

**Doğukan Taştan-1901042627**

# Time Complexity Analysis

## 1. CheckIfValidUsername

```
/** function which checks if it contains only letters, and the minimum length is 1. */
public boolean checkIfValidUsername(String username) {
    if (username == null || username.length() == 0) {
        System.out.println("The username is invalid.It should have at least 1 character.");
        return false;
    }

    /** call helper */
    return checkIfValidUsernameHelper(username, 0);
}
private boolean checkIfValidUsernameHelper(String username, int index) {

    if (index == username.length()) {
        return true;
    }

    char currentChar = username.charAt(index);
    if (!Character.isLetter(currentChar)) {
        System.out.println("The username is invalid.It should have letters only. ");
        return false;
    }

    return checkIfValidUsernameHelper(username, index + 1);
}
```

The function processes each character of the username string one by one.

The function checks the next character by incrementing index by 1 on each recursive call. If index is equal to the length of the username string, the function returns true. If the current character is not a letter, the function returns false.

Let  $T(n)$  represent the time it takes the checkIfValidUsername function to process a username of length  $n$ . In this case, each call of the function will occur by decrementing the length of the username by one character. Therefore, the recursive equation is as follows:

$$T(n) = T(n-1) + c$$

$$= T(n-2) + c + c$$

$$= T(n-3) + c + c + c$$

$$= \dots$$

$$= T(0) + c * n$$

This leads us to the following conclusion:

$$T(n) = c' + c * n$$

(Here  $c'$  represents a constant value for fixed-time operations that occur when the length of the username string is 0.)

This expression shows that the function  $T(n)$  is directly proportional to  $n$ , so the complexity can be considered  **$O(n)$** .

## 2.containsUserNameSpirit

```
public boolean containsUserNameSpirit(String username, String password1) {  
  
    /**  
     * @param stack contains the username char by char.  
     */  
    Stack<Character> stack = new Stack<>();  
  
    for (int i = 0; i < username.length(); i++) {  
        /** push the stack char by char */  
        stack.push(username.charAt(i));  
    }  
  
    while (!stack.isEmpty()) {  
        char currentChar = stack.pop();  
        /** contains is string method */  
        if (password1.contains(Character.toString(currentChar))) {  
            return true;  
        }  
    }  
  
    /** print error message */  
    System.out.println("The password1 is invalid.It should have at least 1 character from the username");  
    return false;  
}  
//*****
```

processes each character of the string username and adds it to a stack. Processes until the stack is empty. Removes a character from the stack. Checks if this character is present in the password1 string. If a matching character is found during processing, true is returned, otherwise false is returned.

The containsUserNameSpirit function processes each character of username at once and fills the stack. Let the length of username be  $n$ . In this case, the first loop has  **$O(n)$**  complexity.

The second loop processes until the stack is empty. In the worst case, there are  $n$  elements on the stack and for each element the string password1 is searched. Let the length of password1 be  $m$ . In this case, the complexity of String.contains() can be considered  $O(m)$  and the second loop has a complexity of  **$O(nm)$** .

The total complexity is the sum of the complexities of these two loops:  **$O(n) + O(nm) = O(n + nm)$** . This shows that the complexity of the function is directly proportional to the input dimensions  $n$  and  $m$ .

### 3. isBalancedPassword

```
public boolean isBalancedPassword(String password1) {  
    /**  
     * @param stack contains the password1 char by char.  
     */  
    Stack<Character> stack = new Stack<>();  
  
    for (int i = 0; i < password1.length(); i++) {  
        char currentChar = password1.charAt(i);  
  
        if (isOpenParenthesis(currentChar)) {  
            stack.push(currentChar);  
        }  
        else if (isCloseParenthesis(currentChar)) {  
            if (stack.isEmpty() || !isMatchingParenthesis(stack.pop(), currentChar)) {  
                System.out.println("Unbalanced parentheses found.");  
                return false;  
            }  
        }  
    }  
  
    if (!stack.isEmpty()) {  
        System.out.println("Password1 is invalid. It should be balanced");  
        return false;  
    }  
    return true;  
}
```

Processes each character of the password1 string and adds the open parentheses to the stack. When a closed parenthesis is found, it extracts an open parenthesis from the stack and checks if it matches. If the stack is empty after processing all characters, true is returned, otherwise false is returned.

The isBalancedPassword function processes every character of password1 at once. Let the length of password1 be  $n$ . In this case, the for loop has  $O(n)$  complexity.

The operations inside the for loop (character checking, stack operations) are in constant time ( $O(1)$ ). In this case, the complexity of the function can be evaluated as  **$O(n)$** . This means that the complexity of the function is directly proportional to the input size  $n$ .

## 4.isPalindromePossible

```
/** *****  
public boolean isPalindromePossible(String password1) {  
  
    return isPalindromePossible(password1, 0, extractLetters(password1));  
}  
  
/** extracts other elements since only letters will be processed */  
private String extractLetters(String input) {  
  
    StringBuilder letters = new StringBuilder();  
    for (int i = 0; i < input.length(); i++) {  
        char c = input.charAt(i);  
        if (Character.isLetter(c)) {  
            letters.append(c);  
        }  
    }  
    return letters.toString();  
}  
  
/** overriding function, This function will be called when extractLetters returns only a string of letters */  
private boolean isPalindromePossible(String password1, int oddCount, String letters) {  
  
    if (letters.length() == 0) {  
        if (oddCount <= 1)  
            return true;  
        else {  
            System.out.println("The password1 is invalid. It should be possible to obtain a palindrome from the pass  
            return false;  
        }  
    }  
    char firstChar = letters.charAt(0);  
  
    /** I'm checking for a match for the rest. */  
    String remainingLetters = letters.replace(Character.toString(firstChar), "");  
  
    int firstCharCount = letters.length() - remainingLetters.length();  
  
    /** counter increases if there is no match */  
    if (firstCharCount % 2 == 1) {  
        oddCount++;  
    }  
    return isPalindromePossible(password1, oddCount, remainingLetters);  
}  
}
```

The extractLetters function extracts only the letters from the password1 string. This operation has at most  $O(n)$  complexity (n: length of password1).

The isPalindromePossible function is called with a string full of letters and starts the process.

Recursively, at each step the first character of the string is taken and removed from the string. This operation has  $O(m)$  complexity in the worst case (m: number of letters).

At each step a recursive call is made and the remainingLetters string is updated. his recursive process continues until the string is empty.

We have seen that the complexity of the extractLetters function is  $O(n)$ . Let's examine the worst case for the complexity of the recursive isPalindromePossible function. If the length of the string decreases by 1 for each recursive call, there are m recursive calls and each call has complexity  $O(m)$ .

In this case, the total complexity of the recursive function can be evaluated as  $O(m^2)$ . In general, the complexity of the isPalindromePossible function is the sum of  $O(n)$  (first step) and  $O(m^2)$  (recursive calls):  $O(n) + O(m^2)$ . This shows that the complexity of the function is directly proportional to the input dimensions n and m.

## 5. isExactDivision

```
public boolean isExactDivision(int password2, int[] denominations) {
    /** Integer value checking */
    if (password2 < 10 || password2 > 10000) {
        System.out.println("The integer equivalent of password2 should be between 10 and 10,000.");
        return false;
    }
    return isExactDivision(password2, denominations, 0, 0);
}

private boolean isExactDivision(int password2, int[] denominations, int currentSum, int index) {
    if (currentSum == password2) {
        return true;
    }

    if (index == denominations.length || currentSum > password2) {
        return false;
    }

    boolean includeDenomination = isExactDivision(password2, denominations, currentSum + denominations[index], index);
    boolean excludeDenomination = isExactDivision(password2, denominations, currentSum, index + 1);

    return includeDenomination || excludeDenomination;
}
```

Check if password2 is between 10 and 10,000. This operation has  $O(1)$  complexity.

The special function isExactDivision is called recursively and performs the following operations :

- a. If currentSum and password2 are equal, true is returned.

- b. If the end of the array is reached or  $\text{currentSum} > \text{password2}$ , false is returned.

- c. Recursive calls are made to choose between including and excluding the current element.

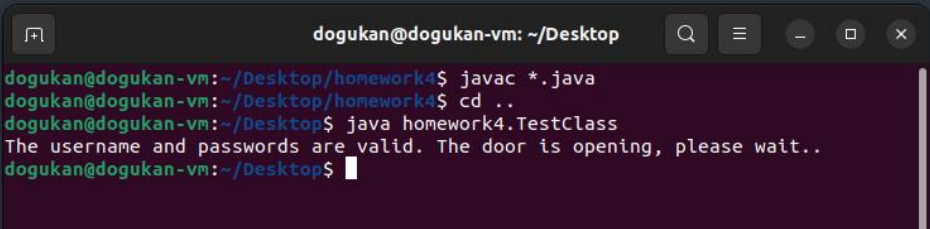
The isExactDivision function continues the recursion by dividing into two subproblems at each step, and at each level it builds a binary tree structure depending on the size of the denominations array ( $n$  levels). Therefore, the maximum number of nodes of the recursive tree is  $2^n$ .

The operations performed at each node are constant time ( $O(1)$ ). Thus, the total complexity can be considered as  **$O(2^n)$** .

## Output Results

The state where all inputs are valid:

```
public class TestClass {  
    public static void main(String[] args) {  
        int[] denominations = {4, 17, 29};  
  
        Username username = new Username();  
        Password1 pass1 = new Password1();  
        Password2 pass2 = new Password2();  
        if(username.checkIfValidUsername("dogukan")  
            && pass1.containsUserNameSpirit("dogukan", "cc22a")  
            && pass1.isBalancedPassword("abcd(cb)a")  
            && pass1.isPalindromePossible("aba")  
            && pass2.isExactDivision(75,denominations)) {  
  
            System.out.println("The username and passwords are valid. The door is opening, please wait..");  
        }  
    }  
}
```

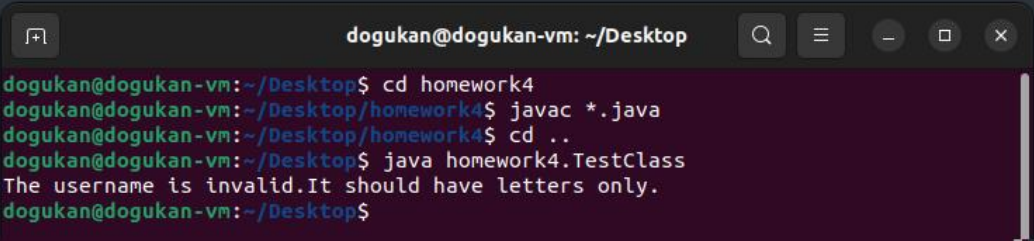


The terminal window shows the following commands and output:

```
dogukan@dogukan-vm: ~/Desktop  
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java  
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..  
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass  
The username and passwords are valid. The door is opening, please wait..  
dogukan@dogukan-vm:~/Desktop$
```

The case where the Username is wrong:

```
public class TestClass {  
    public static void main(String[] args) {  
        int[] denominations = {4, 17, 29};  
  
        Username username = new Username();  
        Password1 pass1 = new Password1();  
        Password2 pass2 = new Password2();  
        if(username.checkIfValidUsername("dogukan29")  
            && pass1.containsUserNameSpirit("dogukan", "cc22a")  
            && pass1.isBalancedPassword("abcd(cb)a")  
            && pass1.isPalindromePossible("aba")  
            && pass2.isExactDivision(75,denominations)) {  
  
            System.out.println("The username and passwords are valid. The door is opening, please wait..");  
        }  
    }  
}
```

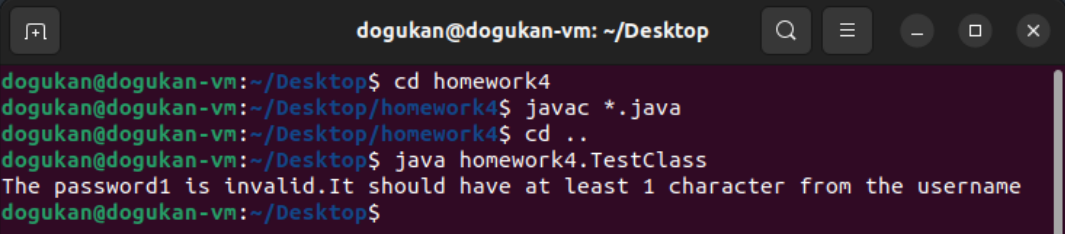


The terminal window shows the following commands and output:

```
dogukan@dogukan-vm: ~/Desktop  
dogukan@dogukan-vm:~/Desktop$ cd homework4  
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java  
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..  
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass  
The username is invalid.It should have letters only.  
dogukan@dogukan-vm:~/Desktop$
```

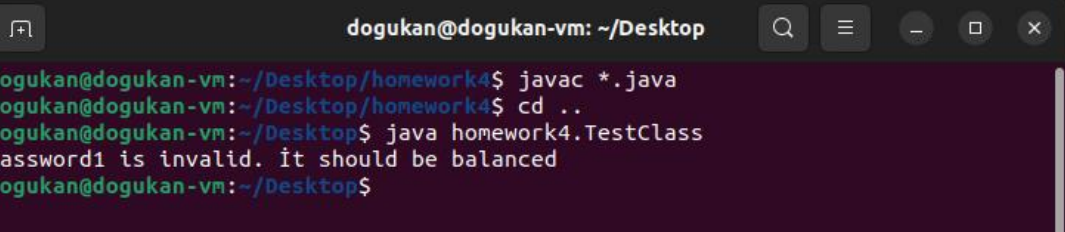
The case where the password1 is wrong:

```
public class TestClass {  
    public static void main(String[] args) {  
        int[] denominations = {4, 17, 29};  
  
        Username username = new Username();  
        Password1 pass1 = new Password1();  
        Password2 pass2 = new Password2();  
        if(username.checkIfValidUsername("dogukan")  
            && pass1.containsUserNameSpirit("dogukan", "cc22")  
            && pass1.isBalancedPassword("abcd(cb)a")  
            && pass1.isPalindromePossible("aba")  
            && pass2.isExactDivision(75,denominations)) {  
  
            System.out.println("The username and passwords are valid. The door is opening, please wait..");  
        }  
    }  
}
```



```
dogukan@dogukan-vm: ~/Desktop  
dogukan@dogukan-vm:~/Desktop$ cd homework4  
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java  
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..  
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass  
The password1 is invalid.It should have at least 1 character from the username  
dogukan@dogukan-vm:~/Desktop$
```

```
public class TestClass {  
    public static void main(String[] args) {  
        int[] denominations = {4, 17, 29};  
  
        Username username = new Username();  
        Password1 pass1 = new Password1();  
        Password2 pass2 = new Password2();  
        if(username.checkIfValidUsername("dogukan")  
            && pass1.containsUserNameSpirit("dogukan", "cc22a")  
            && pass1.isBalancedPassword("abcd(cb)a(")  
            && pass1.isPalindromePossible("aba")  
            && pass2.isExactDivision(75,denominations)) {  
  
            System.out.println("The username and passwords are valid. The door is opening, please wait..");  
        }  
    }  
}
```



```
dogukan@dogukan-vm: ~/Desktop  
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java  
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..  
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass  
Password1 is invalid. It should be balanced  
dogukan@dogukan-vm:~/Desktop$
```



```

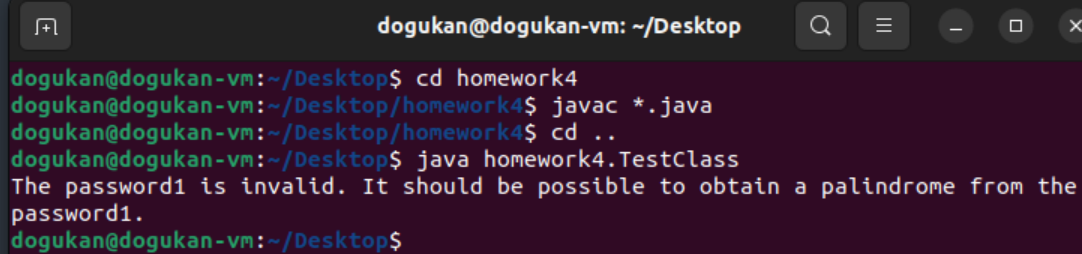
public static void main(String[] args) {

    int[] denominations = {4, 17, 29};

    Username username = new Username();
    Password1 pass1 = new Password1();
    Password2 pass2 = new Password2();
    if(username.checkIfValidUsername("dogukan")
        && pass1.containsUserNameSpirit("dogukan", "cc22a")
        && pass1.isBalancedPassword("abcd(cb)a")
        && pass1.isPalindromePossible("abas")
        && pass2.isExactDivision(75,denominations)) {

        System.out.println("The username and passwords are valid. The door is opening, please wait..");
    }
}

```



```

dogukan@dogukan-vm: ~/Desktop
dogukan@dogukan-vm:~/Desktop$ cd homework4
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass
The password1 is invalid. It should be possible to obtain a palindrome from the
password1.
dogukan@dogukan-vm:~/Desktop$

```

The case where the password2 is wrong:

```

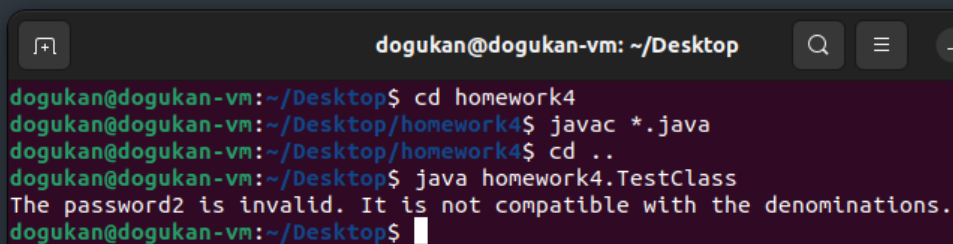
public static void main(String[] args) {

    int[] denominations = {4, 17, 29};

    Username username = new Username();
    Password1 pass1 = new Password1();
    Password2 pass2 = new Password2();
    if(username.checkIfValidUsername("dogukan")
        && pass1.containsUserNameSpirit("dogukan", "cc22a")
        && pass1.isBalancedPassword("abcd(cb)a")
        && pass1.isPalindromePossible("aba")
        && pass2.isExactDivision(35,denominations)) {

        System.out.println("The username and passwords are valid. The door is opening, please wait..");
    }
}

```



```

dogukan@dogukan-vm: ~/Desktop
dogukan@dogukan-vm:~/Desktop$ cd homework4
dogukan@dogukan-vm:~/Desktop/homework4$ javac *.java
dogukan@dogukan-vm:~/Desktop/homework4$ cd ..
dogukan@dogukan-vm:~/Desktop$ java homework4.TestClass
The password2 is invalid. It is not compatible with the denominations.
dogukan@dogukan-vm:~/Desktop$

```