CSE 222: Data Structures and Algorithms ()

Homework #4 Time Complexity Analysis

Instructor: Dr. Gökhan Kaya *Name:* Tugay Talha İçen *Student Id:* 210104004084
Assistant: Sibel Gülmez *210104004084*

Problem 1: checkIfValidUsername (? points)

```
    static boolean checkIfValidUsername(String username) {
        // checks string validation and lenght
a.      if(username == null || username.length() == 0) {
            System.out.println("the username is invalid, it should be at least 1 character
            return false;
        }
        // checks first char is letter or not
b.      if(isLetter(username.charAt(0))) {
            // if last letter valid string is valid
            if(username.length() == 1)
            return true;
            // checks other letters
            return checkIfValidUsername(username.substring(1));
        }
        System.out.println("the username is invalid, it shouldcontains only letters");
        return false;
    }
```

(a) {This part have basic compression and println, there is no loop all parts have O(1) time complexity }

(b) {There is no loop in this part isLetter(O(1), charat(O(1)) .lenght()(O(1)) and a recursive call with substring function that O(N).)}

(Conc.) { This method recursively checks the rest of the characters with creating substrings and have O(N) per step. The recursion continues until either the end of the string is reached that's why this make the method $O(N^2)timecomplexity$)}

(Solution)

$$checkIfValidUsername\ have\ O(N^2)timecomplexity(lenghtoftheusername)$$

Problem 2: isBalancedPassword *(? points)*

```
   static boolean isBalancedPassword(String password1) {
a.        Stack<Character> stack = new Stack<>();
       boolean isAnyBracket = false, isAnyLetter = false;

b.        for (char c : password1.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
                isAnyBracket = true;
            } else if (c == ')' || c == '}' || c == ']') {
                if (stack.isEmpty()) {
                    System.out.println("the password1 is inv
                    return false;
                }
                char top = stack.pop();
                if ((c == ')' && top != '(') || (c == '}' &&
                    System.out.println("the password1 is inv
                    return false;
                }
            } else if(isLetter(c)) {
                isAnyLetter = true;
            } else {
                System.out.println("the password1 is invalid
                return false;
            }


       }
c.        if (!isAnyBracket)
            System.out.println("the password1 is invalid, it
       if (!isAnyLetter)
            System.out.println("the password1 is invalid, it
       if(!stack.isEmpty())
            System.out.println("the password1 is invalid, it

       return (stack.isEmpty() && isAnyBracket && isAnyLett
    }
```

(a) {*Initialization of Stack and boolean variables has a time complexity of O(1).*
}

(b) { *for loop iterates over each character of the password1.toCharArray(), so*

the time complexity of the loop is O(N), where N is the length of the password. and toCharArray method have O(N) time complexity that calls once }

(b.inloop) {*In the loop there are push, pop, isLetter, isEmpty and println methods all of these methods have O(1) time complexity in worst case* }

(c) {*There is only compressions and println function that have O(1) time complexity* }

(Conc.) { *Therefore, the time complexity of the isBalancedPassword function is O(N), where N is the length of the password.)*}

(Solution)

isBalancedPassword function is O(N)(length of the password.)

Problem 3: containsUserNameSpirit *(? points)*

```
   static boolean containsUserNameSpirit(String username, Str
a.          Stack<Character> stack = new Stack<Character>();

        if(password1.length() < 8) {
            System.out.println("the password1 is invalid, it

        }

        // Push each character of the password onto the stac
b.        for (char c : password1.toCharArray()) {
          stack.push(lowerCase(c));
        }

 c.       for(char c : username.toCharArray()) {
            if (stack.contains(lowerCase(c)))
                return true;
        }

 d.       System.out.println("the password1 is invalid, it s
        return false;
    }
```

(a) {*Initialization of stack, compression in if and println have a time complexity of O(1).* }

(b) { *for loop iterates over each character of the password1.toCharArray() and pushes to stack, so the time complexity of the loop is O(N). toCharArray method have O(N) time complexity that calls once, and push function have O(1). At the end for loop has O(N) time complexity, where N is the length of password1.* }

(c) { *for loop iterates over each character of the password1.toCharArray() and checks stacks for it's lowercase version, so the time complexity of the loop is O(N*M). toCharArray method have O(M) time complexity that calls once, lowercase function have O(1), and contains method has O(N) . At the end for loop has O(N*M) time complexity, where N is the length of password1 and M is length of username.* }

(d) {*There is only println function that have O(1) time complexity* }

(Conc.) { *In the worst case, if both the length of password1 and the length of the username are equal to the maximum possible length, then the time complexity*

will be O(N^2*)*, $where N is the maximum length. Otherwise O(N * M) time complexity, where N is the length of password1 and M is length of username.$}

(Solution)

*containsUserNameSpirit function is O(N*M)(length of the password and username)*

Problem 4: isPalindromePossible *(? points)*

```
    static boolean isPalindromePossible(String password1) {
a.          StringBuilder output = new StringBuilder();
        String noBPass;

        for (char c : password1.toCharArray()) {
            if (c != '(' && c != ')' && c != '[' && c != ']'
                output.append(c);
            }
        }

b.          Boolean[] table = new Boolean[output.length() + 1]
        for(int i = 0; i < output.length(); ++i) {
            table[i] = false;
        }

        table[output.length()] = (output.length() % 2 == 0)

        noBPass = output.toString();

c.          if(!recursivePolindrome(noBPass, table, 0)) {
              System.out.println("the password1 is invalid, it
              return false;
        }
        return true;
    }

    private static boolean recursivePolindrome(String passwo
d.          if(index == password1.length())
        return true;
        if(table[index] == true)
            return recursivePolindrome(password1, table, ind

        char c = password1.charAt(index);

e.          for(int i = index + 1; i < password1.length(); ++i
              if (c == password1.charAt(i) && !table[i]) {
                  table[i] = true;
                  return recursivePolindrome(password1, table,
              }
        }
```

```
f.          if(table[password1.length()]) {
                table[password1.length()] = false;
                return recursivePolindrome(password1, table, ind
            }

            return false;
        }
```

(a) {*string builder initialization have O(N)*}

(b) { *As we analyze in previous questions password1.toCharArray() have O(N) call one,Array initialization has O(N)(default value assigment) or O(1) depend on JVM, for have O(N), inside of for there is only assgiment that is O(1).* }

(c) { *A recursive call that we will be analyez in the down, and println that have O(1)*}

(d) {*compression O(1) and continue recursion call* }

(e) {*have a loop iterates until find same character and do oen more recursion call that call have* $O(N^2)inworstcasetimecomplexity, andO(N)bestcasetimecomlexity, whereNislenghtof$

(f){$Itispossibletoworkonetimeforeachcheckandcontinuestherecursivevallifwork$}

(Conc.) { *In the worst case, have* $O(N^2)timecomplexity, whereNisthelengthofpassword1andinbestcasehaveO(N)timeco$

(Solution)

isPalindromePossible function is $O(N^2)worstcase, O(N)bestcase(lengthofthepassword)$

Problem 5: isExactDivision *(? points)*

```
    static boolean isExactDivision(int password2, int
[] denominations) {
        // Check if password between 10 and 10000
 a.       if(password2 < 10 || password2 > 10000) {
            System.out.println("the password2 is invalid, it
            return false;
        }

        // Checks if password2 obtainable by summing denomin
  b.       if (!isExactDivisionRecursive(password2, denominat
            System.out.println("the password1 is invalid, it
            return false;
        }

        return true;
    }

   private static boolean isExactDivisionRecursive(int
password2, int [] denominations, int currentsum, int inde
 c.       if(index != -1)
            currentsum += denominations[index];
        if (currentsum == password2)
            return true;
        if (currentsum > password2)
            return false;

 d.       for (int i = 0; i < denominations.length; ++i) {
            if(isExactDivisionRecursive(password2, denominat
                return true;
        }
        return false;
    }
```

(a) {*Basic compression and print have O(1)*}

(b) { *A recursive call that we will be analyze in the down below, and println that have O(1)*}

(c) { *Basic compressions sum and assigment all of them have O(1)*}

(d) {*Each re-*

cursive call results in another loop that also executes n times, leading to a total of $n^n iterations. This is because each recursive call iterates through the entire denominations ar$

(Conc.) { *In the worst case, have* $O(N^N) time complexity, where N is the length of demonitaions and in best case have O(1) time$

(Solution)

isPalindromePossible function is $O(N^N) worst case, O(1) best case (length of the denominations)$