GTU Department of Computer Engineering
CSE 222/505 - Spring 2023
Homework 4 Report

Mehmet Mete Şamlıoğlu
200104004093

1-) SYSTEM REQUIREMENTS

The system is a basic password encryption system that checks username, password1, and password2 according to the cases mentioned in PDF. Some of the methods are recursive and others are uses Stack data structure as a data container in it. The main purpose is to perform recursion and stack methods to solve the problems. In methods that use recursion, the main goal is to divide problems into sub-problems and after that solve each sub-problem. This way makes it easy to solve a complex problem. However, in question 5 it was not easy to develop a pure recursive method(no loop in it) to solve the problem. When it comes to stack, the stack is LIFO (Last in first out) data structure, it has 5 methods pop, peek, push, empty, and search and the only element that you can access is the last element that has been pushed to stack. When compared to recursive algorithms developing, performing the same algorithm with stack becomes relatively easy.

One of the other main goals of the homework was to analyze the time complexities of both strategies. Some of them become much more than I expected, but I did not find any other more efficient way to solve them (question 5). The detailed analysis of each function will be in the next stage.

2-) PROBLEM-SOLUTION APPROACH AND TIME COMPLEXITY ANALYSIS FOR EACH FUNCTION

In this stage, I will examine all the methods that have been asked one by one and after that, I will calculate the time complexities one by one.

1-) **[A Recursive Function] boolean checkIfValidUsername(String username):** a function which checks if it contains only letters, and the minimum length is 1

Problem solution; In the beginning I checked if the length of the string is less than 1, if it is not satisfied the function returns false to the main in the beginning. After that, I recursively call the helper recursive method (checkIfValidUsernameHelper) that is going to iterate all the strings recursively and check whether elements of the string are only letters or not. It performs it by checking the integer ASCII value of each character. I did count the uppercase letters too but an uppercase letter is a different character than a lowercase letter character so the username “Mete” and “mete” are not the same.

```
    */
    public boolean checkIfValidUsername(String username)
    {
        if(username.length( ) < 1)   O(1)
        {
            System.out.printf("The username is invalid, the minimum username length is 1. Try again.\n");   O(1)
            return false;   O(1)        T(n) = O(1)
        }

        return checkIfValidUsernameHelper(username);
    }

    /**
    * A recursive function which checks if it contains only letters, and the minimum length is 1.
    * @param username  A username of the employee that is going to be check whether it is valid or not
    * @return Return true if username satisfies the conditions, false otherwise.
    */
   public boolean checkIfValidUsernameHelper(String username)   → T(n)
   {
       /* Base case: Is string null or length < 1*/
       if(username == null || username.length( ) < 1)   — O(1)
           return false;
       /* Second base/end case : length == 1 */
       else if(username.length( ) == 1)   — O(1)
       {
           char lastLetter = username.charAt(0);
               /* Check if the ASCII code of letter is inbetween a-z or A-Z */
               if((lastLetter>=65 && lastLetter <= 90) || (lastLetter<=122 && lastLetter>=97))   — O(1)
                   return true;   O(1)
               else
               {
                   System.out.printf("The username is invalid, the username must contain only letters. Try again.\n");   O(1)
                   return false;   O(1)
               }
       }
       else
       {
           char checkLetter = username.charAt(0);   — O(1)
           /* Check if the ASCII code of letter is inbetween a-z or A-Z */
           if((checkLetter>=65 && checkLetter<= 90) || (checkLetter<=122 && checkLetter>=97))   → T(n-1)
               return checkIfValidUsername(username.substring(1));  /*Remove the first element and call the function again */
           else
           {
               System.out.printf("The username is invalid, the username must contain only letters. Try again.\n");
               return false;
           }                                       T(n) = T(n-1) + 1
       }
   }
```

The time complexity of the checkIfValidUsername(String username) method is T(n) = θ(1) if we exclude the return statement that recursively calls checkValidUsernameHelper.

checkifValidUsernameHelper is a helper function that recursively iterates the string and checks the elements of the string. When I analyzed the method, I ended up with the **T(n) = T(n-1) + 7** equation. There are many constants in the function but there is no difference between writing constantly as 7 or 1 to the recurrence equation so I will simplify that and analyze the recurrence relation of **T(n) = T(n – 1) + 1**

The solution to this recurrence relation is below.

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

$$T(n) = \begin{cases} 1, & n = 0 \\ T(n-1) + 1, & n > 0 \end{cases}$$

$$*\,T(n) = T(n-1) + \underbrace{1}_{1}$$

$$T(n-1) = T(n-2) + 1$$

$$*\,T(n) = [T(n-2) + \underbrace{1] + 1}_{2}$$

$$T(n-2) = T(n-3) + 1$$

$$*\,T(n) = [T(n-3) + \underbrace{1] + 1 + 1}_{3}$$

(*)

(*)
continue for k times
Assume n=k at k. step

$$T(n) = T(n-k) + k$$

$$T(n) = \underbrace{T(0)}_{1} + n$$

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

$$O(n)$$

As you can see the time complexity above **the result of recurrence is T(n) = n + 1** and the time complexity of checkifValidUsernameHelper is **O(n)**

2) [A Stack Function] boolean containsUserNameSpirit(String username, String password1): a function which checks if the string password contains at least one letter of the username

Problem solution; In the beginning, I created a stack and pushed all the letters in the stack in it by iterating the index. After that, I created another for loop, and I checked each element of the string again and compared the element of the string with all the elements in Stack by using the search method of the stack. It was not an efficient solution but I was not able to find a better solution than **O(n^2)**

```java
/**
 * A function which checks if the string password contains at least one letter of the username.
 * @param username  A string that contains the username of employee
 * @param password A string password that contains brackets and letters
 * @return True if the string password contains at least one letter of the username, false otherwise
 * */
public  boolean containsUserNameSpirit(String username, String password1)
{
    Stack<Character> myStack = new Stack<Character>( );
    char ch;
    boolean isExist = false;          O(1)

    for(int i = 0 ; i < password1.length( ); i++)          O(m)
    {
        ch = password1.charAt(i);
        if(ch != '(' || ch != ')' || ch != '[' || ch != ']' || ch != '{'  || ch != '}')
        myStack.push(ch);          O(1)
    }
    for(int index = 0 ; index < username.length( ); index++)          O(n)
    {
        ch = username.charAt(index);          O(1)
        if( myStack.search(ch) > 0)          O(n)
        {
            isExist = true;          O(1)
            break;
        }
    }                                                T(n) = O(n^2)
    /* Error print */
    if(isExist == false)          O(1)
        System.out.printf("The username is invalid, username must include at least one letter of username. Try again.\n");

    return isExist;
}
```

As I said, I used a search( ) method in the second loop which has a O(n) time complexity. Therefore the **time complexity of this method is O(n^2)**

3) [A Stack Function] boolean isBalancedPassword(String password1): In the given string sequence, the function considers two brackets to be matching if the first bracket is an open bracket, (ex: (, {, or [), and the next bracket is a closed bracket of the same type. The string cannot start with a closed bracket. There can be letters between any two brackets

Problem solution; We have solved exactly the same question in our course, it is from our coursebook. First, we need to check the length of the password and also the number of bracket occurrences in the string. There must be at least 2 brackets and the length must be 8 at least. Secondly, I removed letters and left only brackets in the string. After that, I started iterating the element of password1 and whenever I encounter an open parenthesis I push it onto the Stack, and whenever I encounter a close parenthesis I pop the stack and compare the brackets. If after I pop the stack the element at the top is the same type of bracket as the one I encounter. So If they are matching I continue iterating, if not I terminated the loop and return false immediately. So if an iteration of password1 is completed successfully it means that the parenthesis is balanced.

```java
{
    Stack<Character> myStack = new Stack<Character>( );          θ(1)
    String password = ""; /* A string that contains no character other than {([])} */
    char ch, topElement;
    boolean isBalanced = true;          θ(1)
    int brackets_count = 0;

    /* Control length of the password */          θ(1)
    if(password1.length( ) < 8)
    {
        System.out.printf("The password1 is invalid, the minimum length of password1 string must be 8. Try again.\n");
        return false;
    }
    for(int i = 0 ; i < password1.length( ); i++)          θ(n)
    {
        ch = password1.charAt(i);
        if(ch == '(' || ch == ')' || ch == '[' || ch == ']' || ch== '{'  || ch == '}')          θ(1)
        {
            password = password + ch;          O(1)
            brackets_count++;
        }
    }

    if(brackets_count < 2) /* Brackets can not be less than 2 */          θ(1)
    {
        System.out.printf("The password1 is invalid, it must include at least 2 brackets. Try again.\n");
        return false;
    }

    for(int index = 0; index < password.length( ) && isBalanced != false; index ++)          θ(n)
    {
        ch = password.charAt(index);
        if(ch == '{' || ch == '(' || ch == '[')          θ(1)
            myStack.push(ch);
        else if(ch == '}' || ch == ')' || ch == ']')
        {
            if(myStack.empty( ) == false)
                topElement = myStack.pop( );          θ(1)
            else
            {
                isBalanced = false;          O(1)
                break;
            }
            switch(topElement)
            {
                case '{':
                    if(ch != '}')
                        isBalanced = false;          θ(1)
                    break;
                case '[':
                    if(ch != ']')
                        isBalanced = false;          O(1)
                    break;
                case '(':
                     if(ch != ')')
                        isBalanced = false;          θ(1)
                    break;
                default:
                        isBalanced = false;          θ(1)
                        break;
            }
        }
    }                                                  T(n) = θ(n)

    if(isBalanced == false)          θ(1)

        System.out.printf("The password1 is invalid, the brackets are not balanced. Try again.\n");

    return isBalanced;
}
/**
```

As you can see above at the end our $T(n) = \theta(2n) + \theta(1)$, When we simplify it we will get $T(n) = \theta(n)$. So the time complexity of the isBalancedPassword function is **O(n)**

4-) **[A Recursive Function] boolean isPalindromePossible(String password1):** In the given string sequence, the function considers if it is possible to obtain a palindrome by rearranging the letters in the string. The function ignores the brackets in the string while computing the function. While converting the string to palindrome; you cannot add/remove letters but you can rearrange them in the string.

Problem solution; In this question, I just checked the frequency of odd letter occurrences in the String password1. If the length of the string is even, it can not have any odd number occurrences letter in the String. However, if the length of the string is odd, it can have only one odd number of occurrence letters in the string. This was the only sub-problem that I had to solve. After that, it was really easy to solve it. I created an empty array that contains the number of occurrences of 52 characters. These characters are A-Z and A-Z. So I started putting the occurrences value of each letter starting from index 0 to 52. In the array a-z is in between 0 to 26, and A-Z is in between 26-52. Whenever I encounter I uppercase letter I subtracted 65 from its integer ASCII value and use it as an index. Thanks to that I counted the occurrences properly. When I reached the end of the string. I iterated the integer array and checked how many odd numbers of occurrences letters exist, and after the checking I got the result.

```
boolean isPalindromePossible(String password1)
{
    /* Remove brackets from string */ → O(1)
    char ch;
    String removeBrackets = "";
    int letters[] = new int[52]; /* An array for the letters a-z  A-Z , from 0 to 25 it will containt a-z after that A-Z */

    for(int i = 0; i < 52 ; i++) /* Initialize array with all zeros */ → O(52)
        letters[i] = 0;

    for(int i = 0 ; i < password1.length( ); i++ ) → O(n)                T(n) = O(N)
    {
        ch = password1.charAt(i);
        if( ch != '{' && ch !='(' && ch != '[' && ch != ')' && ch !=']' && ch != '}') → O(1)
            removeBrackets = removeBrackets + ch;
    }


    return  isPalindromePossible(removeBrackets, 0, removeBrackets.length( ), letters);
}
/**
 * A recursive function that checks the frequency of the occurence in strig.
 * @param password1 a password string that is going to be checked
 * @param length length of the string
 * @param lettersCount an array that will store the frequency of each letter(from a-z  A-Z ) in string
 * @return Returns true, if a string can be rearranged to be palindrome, false otherwise
 * */
boolean isPalindromePossible(String password1, int index, int length, int [] lettersCount) → T(n)
{
    /* Base case: All the elements in string has been searched,and number of letter occurences in string are added to lettersCount array*/
    if(index == length) → O(1)
    {
        int OddOccurence = 0;
        boolean isPalindrome = true; → O(1)
        for(int i = 0; i < 52 ; i++) /* A loop that will check the freqeuncy of letters in string */ → O(52)
        {
            if( lettersCount[i]!=0 && lettersCount[i] % 2 != 0) → O(1)
                OddOccurence++;
            if( (password1.length( ) % 2 == 0)  && OddOccurence > 0) /* If length of the string is even, odd occurence can not be exist */
            {
                System.out.printf("The password1 is invalid due to the palindrome not possible. Try again.\n"); → O(1)
                isPalindrome = false;
                break;
            }
            else if((password1.length( ) % 2 != 0) && OddOccurence > 1) /* If length of the string is even, only one odd occurence can be exist */
            {
                System.out.printf("The password1 is invalid due to the palindrome not possible. Try again.\n"); → O(1)
                isPalindrome = false;
                break;
            }
        }

        return isPalindrome; → O(1)
    }
    else
    {
        char ch = password1.charAt(index); → O(1)
        int index_of_letter;

        if(ch > 90) /* If the letter inbetween a-z */ → O(1)
            index_of_letter = (int)ch - 97;
        else /* A-Z  */
            index_of_letter = (int)(ch - 65) + 26; /* The uppercase letters will start right after where lower case letters are ended in the array */

        lettersCount[index_of_letter] = lettersCount[index_of_letter] + 1; /* Increase the number of occurences for that letter in string */
        return isPalindromePossible(password1, index + 1, length, lettersCount); → T(n-1)
    }
}
```

So the recurrence relation I had to solve is $T(n) = T(n) + 1$ (it is not $T(n) = T(n) + n$ because it is a fixed-size array and it iterates 52 times in the end case). I already solved the same equation above in question 1. So $T(n) = n + 1$ and the time complexity of isPalindromePossible is **O(n)**

5-) [A Recursive Function] boolean isExactDivision(int password2, int [] denominations): Considering the given list of the denominations, the function determines if it is possible to obtain the password by the summation of denominations along with arbitrary coefficients, which are non-negative integers

Problem solution; This method was the hardest one amongst others. I knew that algorithm from an internet site called "leet code". I solved a similar question before and I know that the calling function recursively consecutively helps us to create all the possibilities. So if you have a small set of values {x, y, z}, to produce all the combinations we can use two recursive calls consecutively, by doing this we will do a 2^n search, and if we chose the base cases properly, it will be easy to solve afterward. Since it was complicated I tried to keep it as simple as I can and made my base – cases attentively. I basically get the first denominator at the end and check whether it is possible to multiply the first denominator with some coefficient and get the password2. This was the most basic case, I subtracted the first denominator from password one and called the function recursively by passing the result of subtraction. I have three base cases, one of them checks if the search is succeeded, and the other one checks if the search is fail for that particular case. So I checked these two base cases for my very first recursive call, so after that for each particular case that fails another recursive call started from the other recursive call. The other recursive call continues searching with different denominators by increasing the index by one. In the end, I made a 2^n search and checked each combination. You can find a more detailed way of searching below.

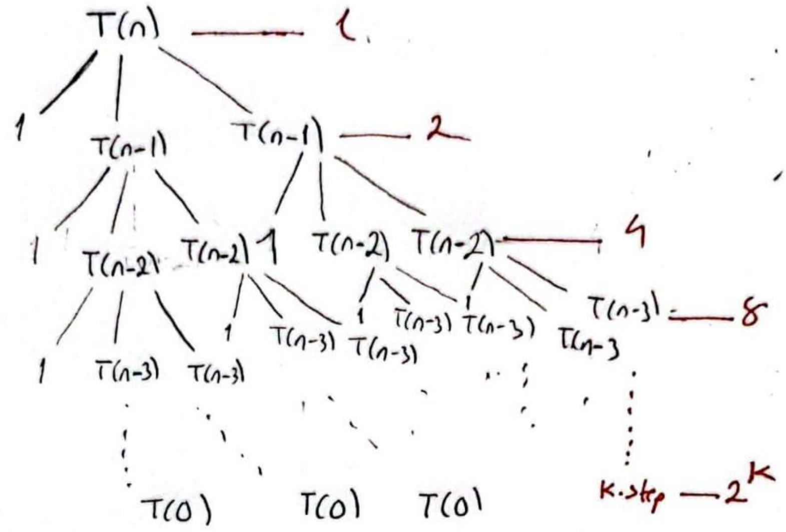
```java
/**
 * This method checks if password2 satisfies the range rule first, after that provekes the overloaded method
 * and checks if we can obtain a password2 by summing and multiplying coeffs with arbitary integers
 *@param password2 A integer password which is inbetween [10, 1000]
 *@param denominations A set of integer which is going to multiply with arbitary coeffs.
 *@return This function makes a recursive call to overloaded recursive isExactDivision function, and it returns the result
 */
public boolean isExactDivision(int password2, int[] denominations)
{
    if( !(password2 > 10 && password2 < 10000))   → θ(1)
    {
        System.out.printf("The password2 is invalid, password2 must be in between (10, 10000). Try again\n");   → θ(1)
        return false;
    }
    else
    {
        boolean isDividible = isExactDivision(password2, denominations, 0);   ?

        if(isDividible == false)   θ(1)
            System.out.printf("The password2 is invalid, it is not possible to obtain the password2 by the summation of denominations(with arbitrary coeffs). Try again\n");

        return isDividible;   θ(1)
    }
}

/**
 * The function determines if it is possible to obtain
 * the password by the summation of denominations along with arbitrary coefficients, which are non-negative integers
 * @param password2 An integer password that is inbetween [10, 1000]
 * @param denominations An integer array that contains denominations for given password2.
 * @return true, if we can get the integer password2 by multiplying denominatons with different coeffs and summing afterwards, false otherwise.
 */
public boolean isExactDivision(int password2, int[] denominations, int index)   T(n)
{

    boolean first_recursion_result;   θ(1)
    /* Base case : If the sum of multiplication denominations with arbitary coeffs equals to password this if block will be activated and retunr true*/
    if (password2 == 0)   θ(1)
    {
        return true;
    }
    /* Base case: It means that search is not successfull for particular denominator*/

    if (password2 < 0 || index >= denominations.length )   θ(1)
    {

        return false;  /* Solution is not exist */   θ(1)

    }
    /* The very first case is to check, if we can find the solution subtracting the very first denomination in array */

    int substracted_password = password2 - denominations[index];   θ(1)

    first_recursion_result = isExactDivision(substracted_password, denominations, index);   T(n-1)

    if (first_recursion_result == true) /* If the first recursion returns true which means that by subtracting the coeff we found the solution */
    {   θ(1)
        return true;
    }

    return isExactDivision(password2, denominations, index + 1);   T(n-1)
}
```

T(n) = 2T(n-1) + θ(1)

So the recurrence relation that I need to solve is T(n) = 2T(n-1) + θ(1), the detailed solution is below. As you can see below, first I created every combination of these denominations, and after that, I tried to perform the operation they are supposed to do. The very last thing was declaring end cases. The end case T(0) is not a general end case when two recursion calls involve. Each particular search must reach T(0), by doing this in the end all the leaves (you can see in the below graph) will be reached.

$T(n) = 2T(n-1)+1$

$$T(n) = \begin{cases} 1 & , n=0 \\ 2T(n-1)+1 & , n>0 \end{cases}$$



So basically that is how we controlled each combination in isExactDivision method. Lets calculate the time complexity with reccurrence relation

$$T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1) & n>0 \end{cases}$$

(1) * $T(n) = 2T(n-1)+1$

$T(n-1) = 2T(n-2)+1$

(2) * $T(n) = 2[2T(n-2)+1]+1 = 2^2T(n-2)+2+1$

$T(n-2) = 2T(n-3)+1$

(3) * $T(n) = 2[2[2T(n-3)+1]+1]+1$

$T(n) = 2^3T(n-3)+2^2+2^1+2^0$

⋮

(*)

k.step

$T(n) = 2^k T(n-k) + \underbrace{2^{k-1}+2^{k-2}+\dots+2^1+2^0}_{2^n-1}$

Assume $n=k$

$T(n) = 2^n \underbrace{T(0)}_{1} + 2^n-1$

$T(n) = 2^n \cdot 1 + 2^n - 1$

$T(n) = 2^{n+1}-1$

$T(n) = \underline{\underline{O(2^n)}}$

3-) RUNNING AND RESULTS

Result of the TestClass function

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java homework4.TestClass
The username and passwords are valid. The door is opening, please wait...
The username is invalid, the minimum username length is 1. Try again.
The password2 is invalid, it is not possible to obtain the password2 by the summation of denominations(with arbitrary coeffs). Try again
The username is invalid, the username must contain only letters. Try again.
The password1 is invalid, the minimum length of password1 string must be 8. Try again.
The password1 is invalid, it must include at least 2 brackets. Try again.
The username is invalid, username must include at least one letter of username. Try again.
The username is invalid, username must include at least one letter of username. Try again.
The password1 is invalid, the brackets are not balanced. Try again.
The password1 is invalid due to the palindrome not possible. Try again.
The password2 is invalid, password2 must be in between (10, 10000). Try again
The password2 is invalid, it is not possible to obtain the password2 by the summation of denominations(with arbitrary coeffs). Try again
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

Results of the Test isPalindromePossible method

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
String: {[(ecarcar)]}, isPalindromePossible: true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
String: {ab[bac]aaba}, isPalindromePossible: false
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
String: {(abba)cac}, isPalindromePossible: true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
String: {kayak}, isPalindromePossible: true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
Is password1 '{[(ecarcar)]}' valid : true
The password1 is invalid due to the palindrome not possible. Try again.
Is password1 '{ab[bac]aaba}' valid : false
Is password1 '{(abba)cac}' valid : true
Is password1 '{[kayak]]' valid : true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

Results of the isBalancedPassword method

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
Is password1 '{[(abacaba)]}' valid : true
Is password1 '{ab[bac]caba}' valid : true
The password1 is invalid, the brackets are not balanced. Try again.
Is password1 ')abc(cba' valid : false
The password1 is invalid, the brackets are not balanced. Try again.
Is password1 'a]bcd(cb)a' valid : false
Is password1 '[a]bcd(cb)a' valid : true
Special cases
 --------------------
The password1 is invalid, the minimum length of password1 string must be 8. Try again.
Is password1 '{mete}' valid : false
The password1 is invalid, it must include at least 2 brackets. Try again.
Is password1 '{meterose' valid : false
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

Results of the checkUsername method

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
Is username 'Mete' valid : true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
The username is invalid, the minimum username length is 1. Try again.
Is username '' valid : false
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

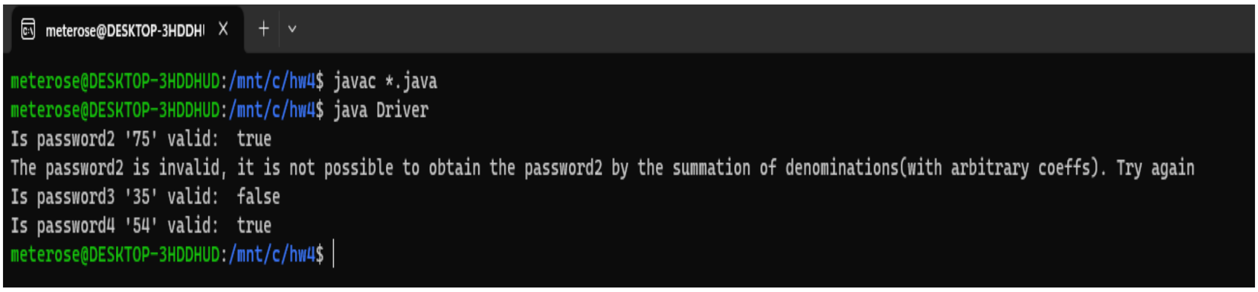
```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
The username is invalid, the username must contain only letters. Try again.
Is username 'mete24' valid : false
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
Is username 'mete' valid : true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

Results of the containsUsernameSpirit

```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java homework4.TestClass
The username is invalid, username must include at least one letter of username. Try again.
Is username 'gizem' contains any letter from {[(abacaba)]}: false
Is username 'gokhan' contains any letter from {[(abacaba)]}: true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

Results of the isExactDivision method



```
meterose@DESKTOP-3HDDHI X
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$ java Driver
Is password2 '75' valid:  true
The password2 is invalid, it is not possible to obtain the password2 by the summation of denominations(with arbitrary coeffs). Try again
Is password3 '35' valid:  false
Is password4 '54' valid:  true
meterose@DESKTOP-3HDDHUD:/mnt/c/hw4$
```

All the required test has been made, and as you can see above it has passed each one of them…

Mehmet Mete Şamlıoğlu

200104004093