GTU DEPARTMENT OF COMPUTER ENGINEERING

CSE – 222 SPRING 2023 HOMEWORK REPORT

ONUR ATASEVER
210104004087

```java
public boolean checkIfValidUsername(String username)
{
    if(username == null)
    {
        System.out.println("String is null");
        return false;
    }
    else if(username.isEmpty())
    {
        System.err.println("String is empty");
        return false;
    }
    //System.out.println("Gelen string: " + username + "\nFirst ir
    if(!Character.isLetter(username.charAt(0)))
    {
        System.out.println("NonLetter: " + username.charAt(0));
        return false;
    }
    if(username.length() != 1)
        return checkIfValidUsername(username.substring(1));
    return true;
}
```

There are four base case fort his recursive function. First one checks whether given string is null or not, second one checks if string is empty or not, third one checks whether first index of string is a letter. And fourth one checks length of string. If it is more than 1, it goes recursively until one of base case conditions satisfy.

TIME COMPLEXITY

CheckIsValid()

Assume length is n

Best Case

If first index is non-letter or string is null or empty.

$T_{best}(n) = \Theta(1)$

Worst Case

If it is not null and empty and if it does not encounter with non-letter, it goes until end of string.

$T_{worst}(n) = T_{worst}(n-1) + \Theta(1)$

$T_{worst}(n) = T_{worst}(n-2) + \Theta(1) + \Theta(1)$

$\vdots$

$T_{worst}(n) = T(n-(n-1)) + \underbrace{\Theta(1) + \ldots + \Theta(1)}_{(n-1)}$

$T_{worst}(n) = \Theta(n)$

So $T(n) = O(n)$

```
public boolean isBalancedPassword(String password1)
{
    Stack<Character> characters = new Stack<Character>();

    for(int i=0; i<password1.length(); i++)
    {
        if(isOpen(password1.charAt(i)))
        {
            characters.push(password1.charAt(i));
        }
        else if(isClosed(password1.charAt(i)))
        {
            if(characters.empty())
            {
                System.out.println("There is not enough open bracket");
                return false;
            }
            char compare = characters.pop();
            if(!isMatch(compare, password1.charAt(i)))
            {
                System.out.println("Brackets did not match");
                return false;
            }
        }
    }
    if(characters.empty())
        return true;
    else
    {
        System.out.println("There are unmatched open brackets");
        return false;
    }
}
```

IsOpen function checks whether index is one of ‘(’, ’[’, ’{’ IsClosed function checks whether index is one of ‘)’, ’]’, ’}’ and isMatch function checks whether given parameters match or not (For ex: ‘(’ and ’)’ matches, ‘(’ and ‘}’ does not match).

If it is open bracket it pushes character to the stack, if it is closed bracket it pop from stack and it controls whether they match. If they does not match it returns false, if there is an closed bracket but there is not any open bracket (I mean stack is empty) it returns false, if it is end of the string but stack is not empty it returns false (because it means there are unmatched open brackets).

If it is end of string and if stack is empty it means all brackets are matched. It returns true.

TIME COMPLEXITY

isBalanced()

Assume length of string is n

Best Case

It encounter with closed bracket directly. Since stack is empty it returns false.

$T_{best}(n) = \Theta(1)$

Worst Case

All brackets match. So it goes end of the string with for loop

$T_{worst}(n) = \Theta(n)$

So $T(n) = O(n)$

```
public boolean  isPalindromePossible(String  password1)
{
    //System.out.println("gelen pass:" + password1);
    String new_password = clean_password(password1);
    String str = "";

    if(isPolindromeCreateable(new_password, 0, 0, str))
    {
        //System.out.println("res: " + str);
        return true;
    }
    else
        return false;
}
```

clean_password function seperates letter and non letter characters and returns new string which contains only letters. It is T(n) = Θ(n)

Main recursive function for polindrome is isPolindromeCreateable.

Parameter new_password contains only letters of password1. Second parameter is target which means the index we'll research in the string, third parameter is index which is compared with target, and last parameter is res, it change string with polindrome if it is possible(this is for test)

```java
public boolean isPolindromeCreateable(String password, int target, int index, String res)
{
    if(target == password.length() && password.length() > 1)
    {
        System.out.println("String can not be polindrome");
        return false;
    }
    if(password.length() == 1)
    {
        StringBuilder sb = new StringBuilder(res);
        sb.append(password.charAt(0));
        res = sb.toString();
        //System.out.println("res: " + res);
        return true;
    }
    if(index == password.length()) //boolean koymaya gerek var mı isFound diye
    {
        return isPolindromeCreateable(password, target+1, 0,res);
    }

    if(target != index && password.charAt(index) == password.charAt(target))
    {
        if(password.length() == 2)
        {
            StringBuilder sb = new StringBuilder(res);
            sb.append(password.charAt(0));
            res = sb.toString();
            //System.out.println("res: " + res);
            return true;
        }
        password = swapIndex(password, target, 0);
        password = swapIndex(password, index, password.length() -1);
        StringBuilder sb = new StringBuilder(res);
        sb.append(password.charAt(0));
        res = sb.toString();
        return isPolindromeCreateable(password.substring(1, password.length()-1), 0, 0, res);
    }
    else
    {
        return isPolindromeCreateable(password, target, ++index, res);
    }

}
```

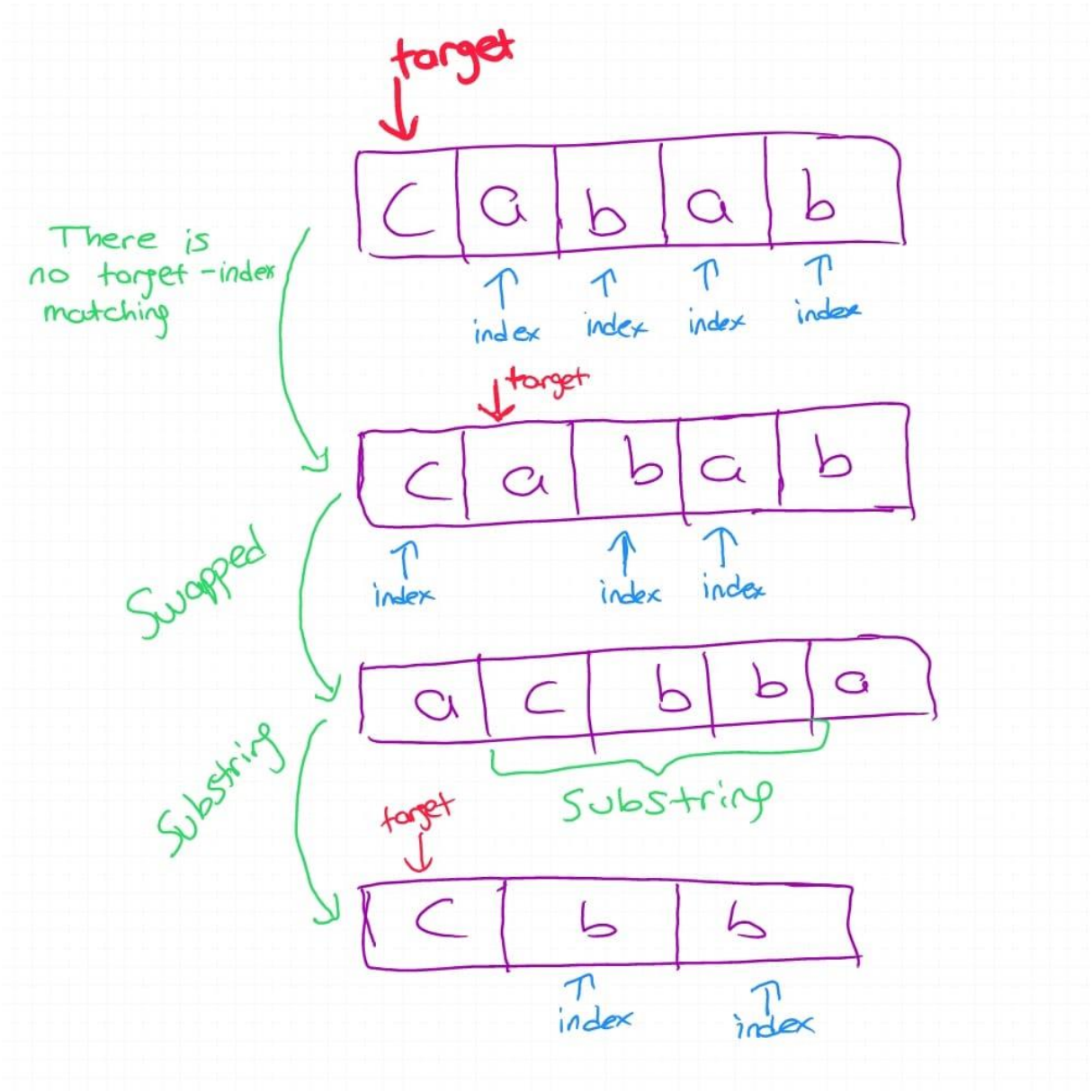
Main idea of this method is selecting an letter of string and finding same letter rest of string. If they match, swap functions swap them with end and beginning of string. And it send substring recursively (For ex: bacac → abcca → bcc → cbc → b)

First base case: If length of string is 1, it is true.

Second base case: If length is more than one, and target is equal to length of string it is false. Because if target is equal to length of string it means it could not find same letter before.

Third base case: If there is match and length of string is 2, it returns true.

This is an example of algorithm.



TIME COMPLEXİTY

isPolindromeCreateable()

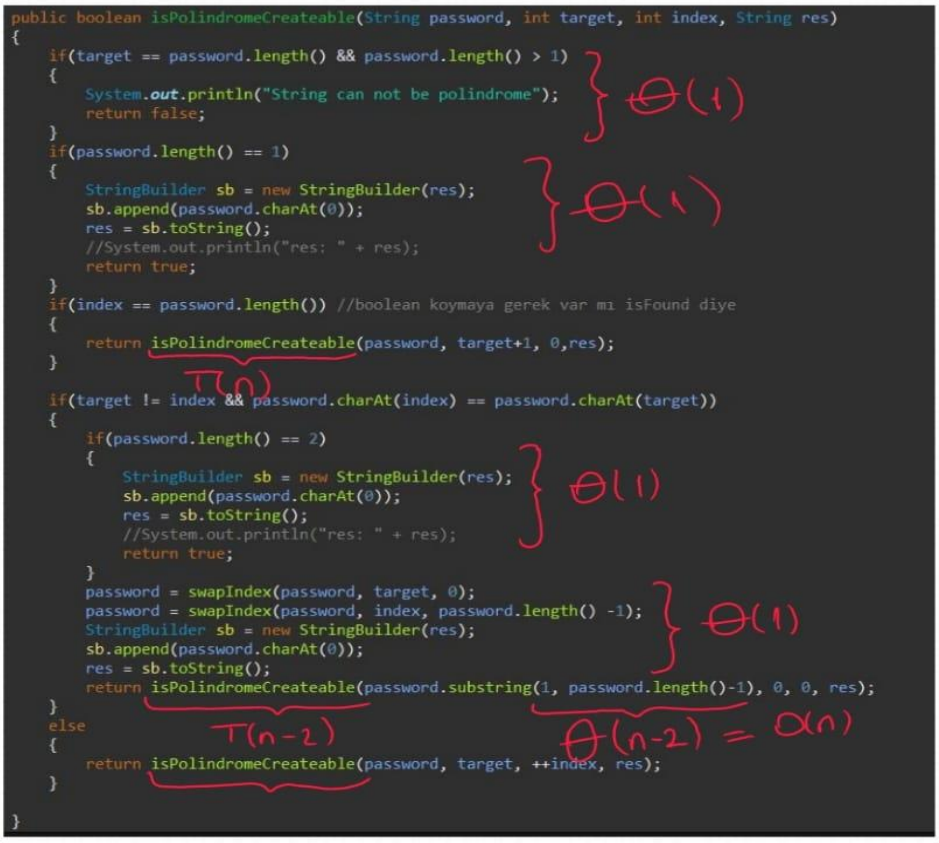
Assume length of password n

Best Case

If n = 1 it is directly polindrom

$T_{best}(n) = \Theta(1)$

Worst Case



```
public boolean isPolindromeCreateable(String password, int target, int index, String res)
{
    if(target == password.length() && password.length() > 1)
    {
        System.out.println("String can not be polindrome");
        return false;
    }
    if(password.length() == 1)
    {
        StringBuilder sb = new StringBuilder(res);
        sb.append(password.charAt(0));
        res = sb.toString();
        //System.out.println("res: " + res);
        return true;
    }
    if(index == password.length()) //boolean koymaya gerek var mı isFound diye
    {
        return isPolindromeCreateable(password, target+1, 0,res);
    }

    if(target != index && password.charAt(index) == password.charAt(target))
    {
        if(password.length() == 2)
        {
            StringBuilder sb = new StringBuilder(res);
            sb.append(password.charAt(0));
            res = sb.toString();
            //System.out.println("res: " + res);
            return true;
        }
        password = swapIndex(password, target, 0);
        password = swapIndex(password, index, password.length() -1);
        StringBuilder sb = new StringBuilder(res);
        sb.append(password.charAt(0));
        res = sb.toString();
        return isPolindromeCreateable(password.substring(1, password.length()-1), 0, 0, res);
    }
    else
    {
        return isPolindromeCreateable(password, target, ++index, res);
    }

}
```

For each target we must check all pairs.

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

$$T(n) = T(n-4) + O(n) + O(n)$$

$$\vdots$$

$$T(n) = \underbrace{T(n-(n-1))}_{T(1)} + \underbrace{O(n) + \dots + O(n)}_{(n-1)/2 \text{ times}}$$

$$T(n) = \Theta(1) + O\left(\frac{n \cdot n-1}{2}\right)$$

$$T(n) = O(n^2)$$

So T(n) = O(n^2)

```java
public boolean isExactDivision(int   password2,   int   []   denominations)
{
    if(checkPossibilities(password2, denominations, 0))
        return true;
    else
    {
        System.out.println("Password can not create with this denominations");
        return false;
    }
}
```

This function calls recursive function and sends parameter 0 as sum.

```java
public boolean checkPossibilities(int password, int[] dominations, int sum)
{
    if (sum > password)
    {
        return false;
    }

    boolean bool;

    if(sum == password)
    {
        return true;
    }

    /*return (checkPossibilities(password, dominations, sum + dominations[0]) ||
            checkPossibilities(password, dominations, sum + dominations[1]) ||
            checkPossibilities(password, dominations, sum + dominations[2]));
*/
    for(int i=0; i<dominations.length; i++)
    {
        bool = checkPossibilities(password, dominations, sum + dominations[i]);
        if(bool)
            return true;
    }
    return false;
}
```

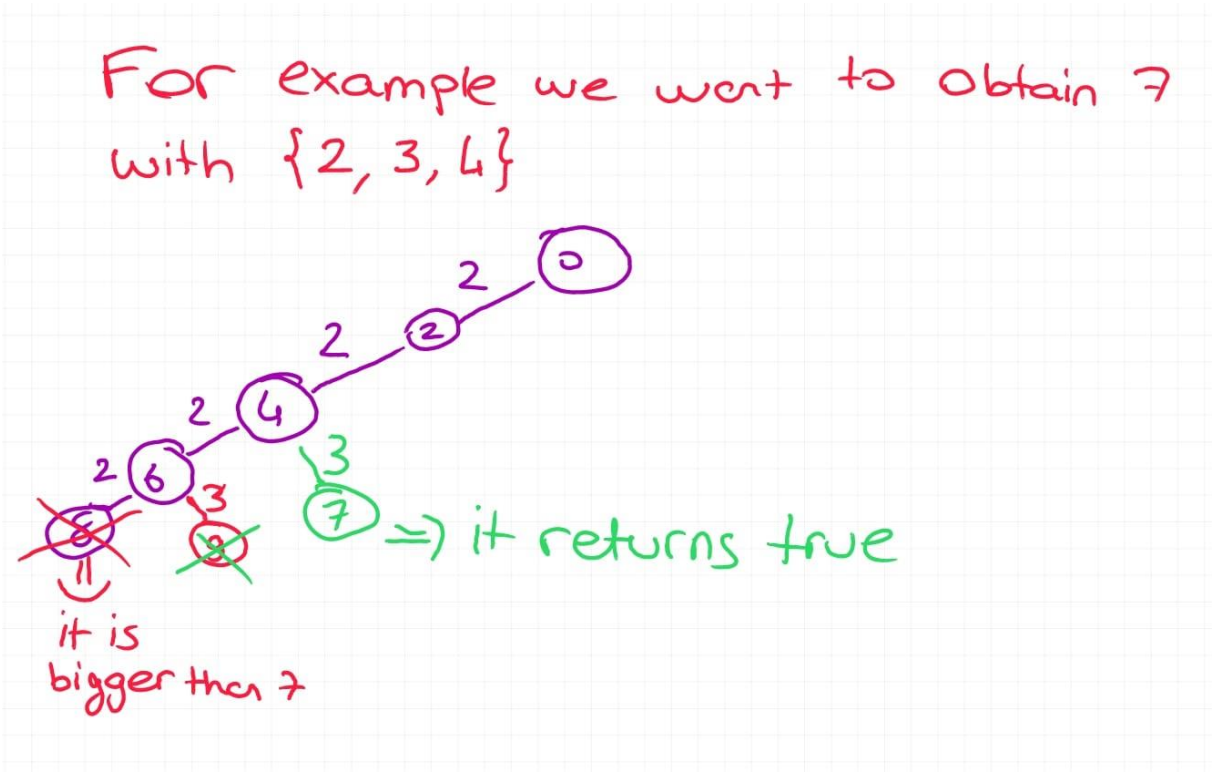
This method tries all possibilities one by one.

First case: If sum is bigger than password it means it could not equal to sum. So it does not need to continue. It return false.

Second case: If sum is equal to password it returns true.

Third case: If sum is still smaller than password it continue to sum.

This is algorithm of my code



TIME COMPLEXITY

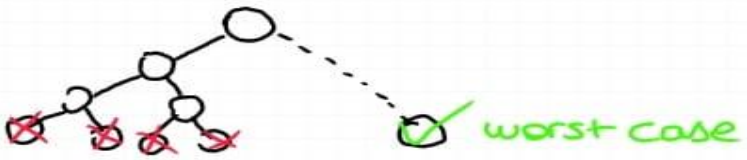
Best Case

In the best-case first denominator is equal to password. So it calls one recursive call.

$T_{best}(n) = \Theta(1)$

Worst Case

Worst Case depends on elements of denominations array and password.



So in the worst case we should Obtain password with last element Of denominator. It means we can not obtain password by using elements except last element.

Complexit depends on number of recursive call. And number of recursive calls until we obtain password with other elements depends on password and elements. If password gets bigger and elements get smaller, number of recursive calls increases.

$$T(sum) = T(sum - denom[0]) + T(sum - denom[1]) + \vdots + T(sum - denom[n]) + \Theta(1)$$

I don't know how to solve this problem

```java
public boolean containsUsernameSpirit(String username, String password1)
{
    Stack<Character> username_stack = new Stack<Character>();
    Stack<Character> password_stack = new Stack<Character>();
    Stack<Character> temp_stack = new Stack<Character>();

    int username_len=0, password_len=0;
    for(char c: username.toCharArray())
    {
        if(Character.isLetter(c))
        {
            //System.out.println("Usernameden pushlanan: " + c);
            username_stack.push(c);
            username_len++;
        }
    }

    for(char c: password1.toCharArray())
    {
        if(Character.isLetter(c))
        {
            //System.out.println("Passwordden pushlanan: " + c);
            password_stack.push(c);
            password_len++;
        }
    }

    return checkUsernamePassword(username_len, password_len, username_stack, password_stack, temp_stack);
}
```

It pushes elements which are letter to stack, and then it send this stacks to checkUsernamePassword which is recursive function

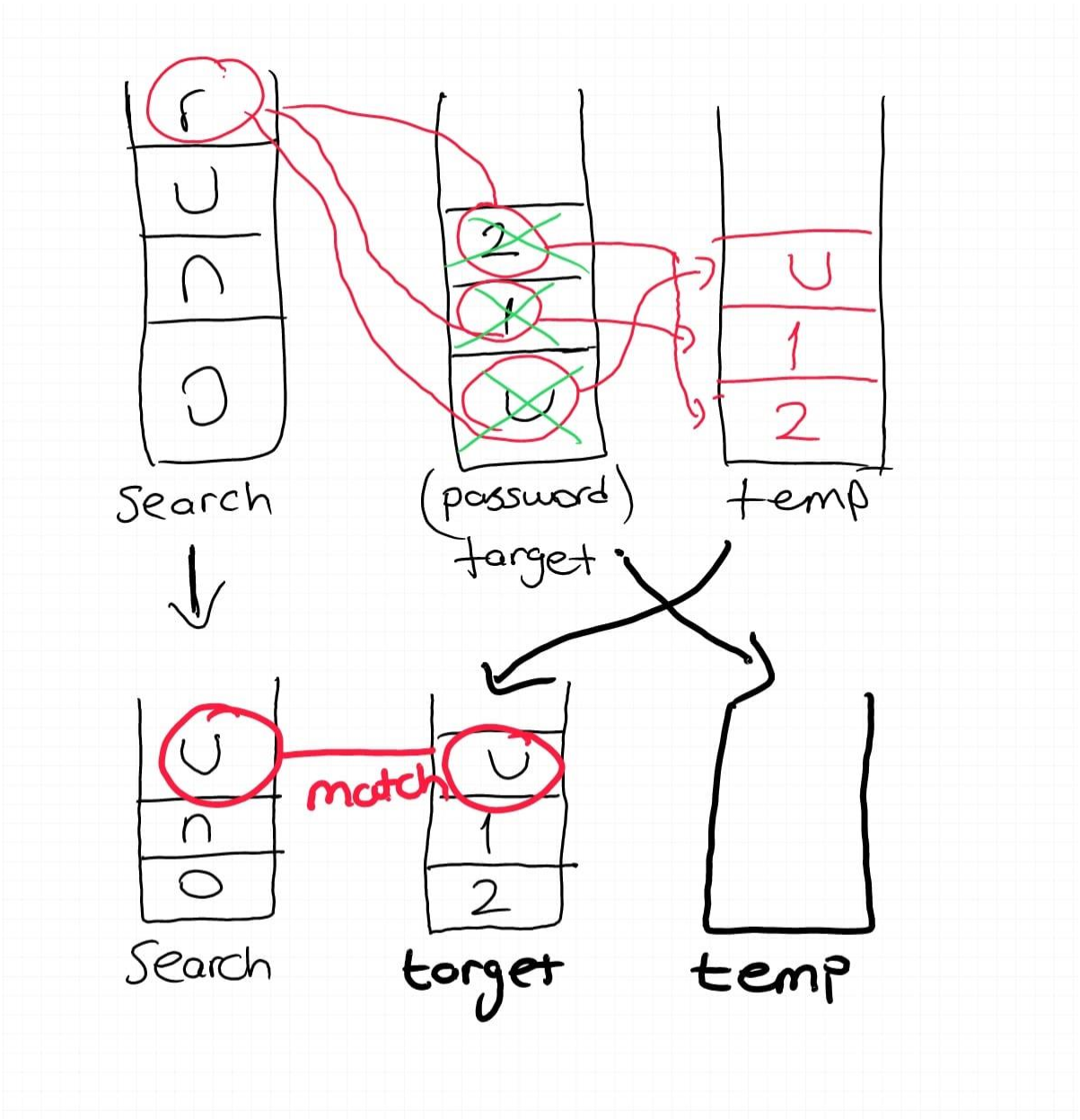
```java
private boolean checkUsernamePassword(int len_username, int len_password, Stack<Character> search,
        Stack<Character> target, Stack<Character> temp)
{
    if(len_username == 0)
    {
        System.out.println("Password does not contain any letter of username");
        return false;
    }
    for(int i=0; i<len_password; i++)
    {
        char control = target.pop();
        if(control == search.peek())
            return true;
        else
            temp.push(control);
    }
    search.pop();
    return checkUsernamePassword(len_username - 1, len_password, search, temp, target);
}
```

Purpose is to control whether password contains at least one letter from username.

So I have to compare stack elements. I compare peek elements both username and password stacks. If they does not match, I pop from password stack and I push it temp stack. So I can compare all password elements with peek element of username. If password stack is empty, it means any element of password stack did not match with the peek element of username stack. So i pop username stack (because it did not match with any element) . Now, I should compare new peek element with password elements. All passwords elements

were in the temp stack, so when i call function recursively, I should change places of temp_stack and password_stack. And at each call, username_stack length should decrease 1.

If it is zero it means there is no any matches, it returns false.



TIME COMPLEXITY

Best Case

Element matches at first call.

Assume m is length of password and n is length of username

$$T(n) = \Theta(1)$$

Worst Case

Stacks element never can match.

for loop $= \Theta(m)$

Other operations $= \Theta(1)$

$$T(n) = T(n-1) + \Theta(m)$$

$$T(n-1) = T(n-2) + \Theta(m)$$

$$T(n) = T(n-2) + \Theta(m)$$

$$\vdots$$

$$T(n) = \underbrace{T(0)}_{\Theta(1)} + \underbrace{\Theta(m) + \dots + \Theta(m)}_{n \text{ times}}$$

$$T(n) = Q(m \cdot n)$$

So T(n, m) = O(m*n)

INPUTS

```
public static void main(String[] args) {
    tester("gizem", "{(abba)cac}", 75);

    tester("ahmet", "{(abba)cac}", 75);

    tester("ahmet", "{(abba)cac}", 35);

    tester("onur", "{(abbo)cac}", 54);

    tester("onur", "[a]bco(cb)a", 54);

    tester("onur", "{(abbo)cac", 54);

    tester("onur", "{(abbocac}", 54);

    tester("onur", "{(abbo)ca}", 54);

    tester("", "{(abbo)cac}", 54);

    tester("onur", "{(oo)}", 54);

    tester("onur", "abbocac", 54);

}
```

OUTPUTS

```
---------------
Username: gizem
Password does not contain any letter of username
---------------
Username: ahmet
The username and passwords are valid. The door is opening, please wait...
---------------
Username: ahmet
Password can not create with this denominations
---------------
Username: onur
The username and passwords are valid. The door is opening, please wait...
---------------
Username: onur
The username and passwords are valid. The door is opening, please wait...
---------------
Username: onur
There are unmatched open brackets
---------------
Username: onur
Brackets did not match
---------------
Username: onur
String can not be polindrome
---------------
Username:
String is empty
---------------
Username: onur
Size of password should be at least 8
---------------
Username: onur
There is not enough bracket
```