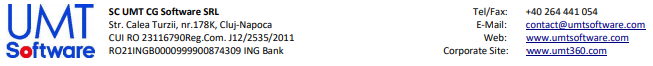
**Strong Password Checker – DOC**

1. **Text:**



A password is considered strong if below conditions are all met:

1. It has at least 6 characters and at most 20 characters.

2. It must contain at least one lowercase letter, at least one uppercase letter, and at least one digit.

3. It must NOT contain three repeating characters in a row ("...aaa..." is weak, but "...aa...a..." is strong, assuming other conditions are met).

Write an algorithm that takes a string s as input, and return the MINIMUM change required to make s a strong password. If s is already strong, return 0.

Insertion, deletion or replace of any one character are all considered as one change.

1. **Technologies used:**

Editing program: CLion by JetBrains

Language: C++ (CMAKE\_CXX\_STANDARD 14)

GitHub: https://github.com/draganmaria99/UMT\_problema4

1. **Package organization:**

The package named “Parola” contains all the relevant files.

1. **Assumptions made:**

* Console application is enough
* The string given by the user will never contain special characters: ,./[]()!@? Etc.
* The string MUST have between 6 and 20 characters. If not, the user is asked for a valid string, that is between 6 and 20.
* There must be a minimal UI, even if in the console.
* Tests are necessary.

1. **Running scenarios:**

**Scenario 1:**

App starts

User input is expected:

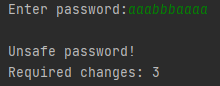


The user inputs a string:



The user presses enter.

The string is valid and the app writes the answer:



**Scenario 2:**

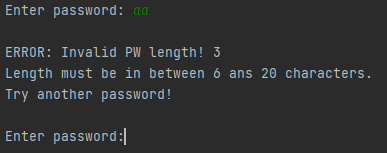
App starts

User input is expected:



The user inputs a string that is too short.

The app asks for another correct input.



Scenario 1 starts again.

1. **Specifications:**

Each function is specified in the header file it belongs to.

**hasDigit, hasUpper** and **hasLower** use functions from the “regex” library.

/\* Function that checks if a string "to\_check" has a digit in it.  
 \* in: string  
 \* out: 0 - if "to\_check" does not contain a digit  
 \* 1 - if "to\_check: does contains a digit  
 \*/  
int hasDigit(std::string to\_check);  
  
  
/\* Function that checks if a string "to\_check" has an upper case character in it.  
 \* in: string  
 \* out: 0 - if "to\_check" does not contain an upper case char  
 \* 1 - if "to\_check: does contains an upper case char  
 \*/  
int hasUpper(std::string to\_check);  
  
  
/\* Function that checks if a string "to\_check" has a lower case character in it.  
 \* in: string  
 \* out: 0 - if "to\_check" does not contain a lower upper case char  
 \* 1 - if "to\_check: does contains a lower case char  
 \*/  
int hasLower(std::string to\_check);

**hasRepeatedChar** uses logic that I made:

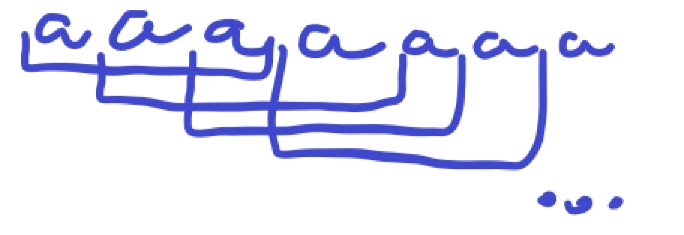
/\* Function that return the number of character pairs that are made of identical characters on consecutive indexes.  
 \* Example of pair: aaa  
 \* in: string  
 \* out: pair - int variable that stores the number of pairs with the required criteria \*/  
int hasRepeatedChar(std::string to\_check){

int repetition;  
 int pair;  
 repetition = 2;  
 pair = 0;  
  
 for(int i = 0; i<to\_check.length()-2; i++){  
 if(to\_check[i] == to\_check[i+1] and to\_check[i] == to\_check[i+2]){  
 repetition++;  
 }  
 else{  
 pair += repetition/3;  
 repetition = 2;  
 }  
 }  
 pair += repetition/3;  
 return pair;  
}

The algorithm counts the number of pairs

It checks groups of 3 consecutive elements. If they are equal, it increases the repetition counter.

Repetition starts at 2 because it counts the exact number of characters in a consecutive repetition stream.



Above is a pair sequencing example.

Every time a sequence of identical characters ends, we calculate the new number of pairs.

If there is a pair at the end, it is calculated outside of the loop.

A pair is the number of repetition for the current character divided by 3.

We can observe that every group of 3 identical elements we need to make a change.

So we return the number of pairs.

**pwStrengthChecker** is the main function that verifies how many criteria are met using the above functions.

/\* Function that takes a string "s" and return the MINIMUM number of changes that need to be made for it to be safe.  
 \* in: string  
 \* out: nr of changes required <int>  
 \*  
 \*/  
int pwStrengthChecker(std::string s);

For the 3 simple things (upper, lower and digit) it counts how many of them are missing.

Then it uses **hasRepeatedChar** to see how many unwanted pairs there are.

If the number of unwanted pairs is bigger than the counter for the other requirements, then the minimum number of changes is the number of unwanted pairs, that is returned.

If the number of unwanted pairs is smaller than the counter for the other requirements, then the number of the other requirements is returned.