Decisions in Visual Basic I

Topics

- if statements
- testing the "randomness" of random numbers
- introduction to concept of string patterns

Preliminaries

If you haven't already done so, start by making a copy of today's lab folder (`Lab04_Decisions`) and saving it in your `M:\CS130\Labs` folder. Right-click on the folder you just copied and rename it `Lab04_YourLastName_YourFirstName` (but use your actual last and first names).

You may now work locally by opening the write-up from within the copied folder.

Flipping a coin

1. Launch the RAPTOR software and use it to create and save a new program called `FlippingACoin_Part1.rap` inside of your lab folder.

   Complete `FlippingACoin_Part1.rap` so that it generates a random number between 0 and 1, exclusive, and prints `"Heads"` when the number is less than `0.5` and `"Tails"` otherwise (i.e., it is greater than or equal to `0.5`).

When done, save and close file `FlippingACoin_Part1.rap`.

2. Launch VS Express 2013 and open the project `FlippingACoin_Part2` which can be found inside of your lab folder.

   Complete the project `FlippingACoin_Part2` so that it is equivalent to your RAPTOR program.

   Run this project 20 times and record below the number of *Heads* and number of *Tails*:

   - number of *Heads*: ______ out of 20 flips
   - number of *Tails*: ______ out of 20 flips

   Does it look like a fair coin (in the sense that the coin is just as likely to generate *Heads* as *Tails*)?

Your project should behave identically to the solution which can be run by double-clicking the file `FlippingACoin_Part2.exe` found in the `Executables` folder inside of your lab folder.

When done, save and close project `FlippinACoin_Part2`.

Checkpoint 1 (30/100): A successful `FlippingACoin_Part2` project:

- ☐ is based on a complete and correct `FlippingACoin_Part1.rap` program that displays Heads and Tails based on your user
- ☐ simulates a fair coin

3. Use **File Explorer** to navigate to your lab folder. Copy and paste the folder `FlippingACoin_Part2`, and then right-click on the copy to rename it as `FlippingACoin_Part3`. Launch VS Express and use it to open the project `FlippingACoin_Part3`.

   Make the necessary changes to the project `FlippingACoin_Part3` to simulate a **biased** coin instead of a fair one. When flipped, this biased coin is three times more likely to generate *Heads* than *Tails* (Hint: recall that you the value `0.5` from the previous part was chosen to split the range `[0,1)` perfectly in half).

   Run this project 20 times and record below the number of *Heads* and number of *Tails*:

   - number of *Heads*: ______ out of 20 flips
   - number of *Tails*: ______ out of 20 flips

   Does it look like a fair coin or a biased coin now?

Your project should behave identically to the solution which can be run by double-clicking the file `FlippingACoin_Part3.exe` found in the `Executables` folder inside of your lab folder.

When done, save and close project `FlippinACoin_Part3`.

Checkpoint 2 (65/100): A successful `FlippingACoin_Part3` project:

☐ simulates a biased coin that is three times more likely to generate *Heads* than *Tails*

Password crack calculator

Disclaimer. The program that you develop for this exercise should **NOT** be considered a measure of the strength of a particular password. An obvious example would be the password '*P@ssw0rd*', which, according to the program, would take 147.53338 centuries to crack, but in practice would be among the first passwords checked by any cracker worth their salt.

TL;DR

Implement a project to calculate the number of passwords that can be constructed from some alphabet, as well as the time it would take to check the entire search space assuming a cracking program capable of some number of guesses per second.

This exercise will give you a better understanding and appreciation for password composition.

Background

Have you ever wondered why many websites have password requirements like:

- be a minimum of 8
- cannot be based on a word or name
- must contain at least one
  - upper-case letter,
  - lower-case letter,
  - number, and
  - special character: ^ $ / % @ #

These are all attempts to prevent users from choosing passwords that are easy to guess. Keep in mind, that in this context, it is not a person guessing, but rather a machine running a *cracking* program, capable of thousands or millions of guesses per second. It turns out that when requirements like this are not enforced, people tend to pick easily guessable passwords. According to a variety of sources, some of the most popular passwords are:

- password
- 123456
- qwerty
- monkey
- password1

By requiring the inclusion of certain groups of characters, it prevents the use of passwords such as those listed above, at least in their most obvious form. If a cracking program is unable to quickly guess a password based on a dictionary of common passwords, such as those above, it may be necessary to use *brute-force* to guess the password, i.e., check all possible passwords that satisfy the requirements.

A useful exercise for any person interested in the security of their password protected identity is to determine how many possible passwords could be constructed from a given set of requirements, and how long it would take a computer cracking program to check all such passwords.

Your task is to create a Visual Basic project that does just that. Your project will take a password as input, then compute the number of possible passwords, called the *search space*, that use the same types of characters as the input password. Then your project will output that search space, along with the amount of time for a computer capable of checking a certain number of passwords per second to crack the input password.

To start, let's define the *password alphabet*. The *password alphabet* is the set of letters, numbers, and other characters that are allowed to be used to compose a password. So, for example, if only lower-case letters were allowed in a password, then the alphabet would be a-z and its size would be 26. However, if upper-case letters as well as numbers were allowed, then the alphabet would be a-z, A-Z and 0-9, and its size would be 62.

Given this definition, then the total number of passwords of length L that can be created from an alphabet of size C can be expressed as:

$$C^L$$

Since a cracking program does not know a priori how many characters are in the password it is trying to crack, it will have to check not only those passwords of length L, but also those passwords whose length is less than L. Thus, using the previous equation, the *search space* for a password of length L can be expressed as:

$$(C^1 + C^2 + \cdots + C^L) = C(C^L - 1)/(C - 1)$$

Given a cracking program capable of checking G passwords per second, the time it would take (in seconds) to check all passwords in a search space of that size can be expressed as:

$$(C(C^L - 1)/(C - 1))/G$$

Keep in mind that this is not an indication of how long it would take to crack a particular password, only of how long to check the entire search space. It is conceivable that any given password will be found in less time than that calculated. Still, it allows one to compare the effect of alphabet size and password length on the search space and time.

Instructions

1. Launch VS Express 2013 and open the project `PasswordCracking_Part1` which can be found inside of your lab folder.

   You are to complete this project so that it accomplishes the tasks described in the previous section.

   However, before you can even think about computing the size of the search space and how long it would take to check, you must determine the size of the alphabet. For this project, the alphabet will be dictated by the password entered by the user. For example, if the user entered the password `PASSWORD`, then the alphabet size would be 26 because `PASSWORD` only contains upper-case letters, of which there are 26. However, if the user had entered `PaSSwORD`, then the alphabet size would be 52 because `PaSSwORD` contains both upper- and lower-case letters.

Any valid password entered by the user can only contain characters belonging to one of the following character types:

- lower-case letters [a-z],
- upper-case letters [A-Z],
- digits [0-9], and
- special characters [^$/%@#].

To compute the total size of the alphabet based on the user input password, determine which character types are included in the user's password and add an amount appropriate for that character type to a variable called `alphabetSize`. To find out if a `String` contains any one of a set of characters, you can use the `Like` operator. For example, to check if the `String` variable `passwd` contains any lower-case letters and add the appropriate amount to the variable `alphabetSize` if it does, you would use the following Visual Basic code:

```
If passwd Like "*[a-z]*" Then
    alphabetSize = alphabetSize + 26
End If
```

The correct pattern for each of the different characters types is as follows:

- lower-case letters `"*[a-z]*"`,
- upper-case letters `"*[A-Z]*"`,
- digits `"*[0-9]*"`, or
- special characters `"*[^$/%@#]*"`.

Complete the project `PasswordCracking_Part1` so that it takes a user input password and computes and outputs the alphabet size based on the user's input.

One final note. Up to this point, our means of output has been limited to a *Rich Text Box*, typically named `outResults`. However, in the project that you have been given, you will be outputting to *Text boxes*, and specifically for this part, a text box named `txtAlphabetSize`. Fortunately, in Visual Basic, outputting to text boxes is done exactly like outputting to rich text boxes, so the Visual Basic statement

```
txtExample.Text = "This is an example of text box output."
```

would put the sentence `"This is an example of text box output."` in the text box named `txtExample`.

After you get your project working, fill out `PasswordCracking_TestTable1.docx`, making sure that your expected output and actual output are consistent.

Your project should behave identically to the solution which can be run by double-clicking the file `PasswordCracking_Part1.exe` found in the `Executables` folder inside of your lab folder.

When done, save and close project `PasswordCracking_Part1`.

Checkpoint 3 (70/100): A successful `PasswordCracking_Part1` project:

- ☐ produces correct output for all test cases in Test Table 1
- ☐ must also have successfully completed Checkpoint 2

2. Use **File Explorer** to navigate to your lab folder. Copy and paste the folder `PasswordCracking_Part1`, and then right-click on the copy to rename it as `PasswordCracking_Part2`. Launch VS Express and use it to open the project `PasswordCracking_Part2`.

   Enhance your project so that instead of outputting just the alphabet size, it outputs the search space size as well. To do this, first recall the equation for computing the search space size,

$$C(C^L - 1)/(C - 1),$$

   where C is the alphabet size and L is the length of the password. Since you computed the alphabet size in Part 1, you are just missing the length L. Luckily this information is easily ascertained from the user input password with the following line of Visual Basic, assuming you are using variables, `passwd` and `passwdLength` to store the input password and its length respectively:

```
passwdLength = passwd.Length
```

   After you get your project working, fill out `PasswordCracking_TestTable2.docx`, making sure that your expected output and actual output are consistent.

   You SHOULD get an error when you input the password in the last row. Getting your project to produce correct results for this input will be the subject of our next lab meeting. For now, it is enough that you consider why this might be happening.

Your project should behave identically to the solution which can be run by double-clicking the file `PasswordCracking_Part2.exe` found in the `Executables` folder inside of your lab folder.

When done, save and close project `PasswordCracking_Part2`.

Checkpoint 4 (75/100): A successful `PasswordCracking_Part2` project:

☐ produces correct output for all test cases in Test Table 2 except `"aA1@@@"`
☐ must also have successfully completed Checkpoint 2

Submission Instructions

During our next lab meeting, you will be asked to expand your solutions to Part 2. Prior to the meeting, you are expected to

- finish ALL parts of this lab,
- study (again) any material you struggled with in this lab, and
- study new material needed for the next lab

You will submit your work for this lab and the next one together at the end of our next lab meeting.