# Lab04

## Part 1

Use Raptor to write a simple program that simulates a single flip of a coin by asking the user to enter a 0 or 1 and printing “Heads” when user inputs a 1 and “Tails” when user inputs a 0.

a. Modify the program to print an error message in case user enters anything other than a 0 or 1.

   NOTE: Conditions cannot be compound.

b. Convert your program to Visual Basic (VB).

c. Change your VB program to randomly generate an integer between 0 and 1 instead of asking the user for input. Run this program 20 times and record the sequence of “Heads” and “Tails”. Does it look like a fair coin?

d. Create another version of program that simulates a biased coin, which when flipped is 3 times more likely to generate heads than tails. Run this program 20 times and record the sequence of “Heads” and “Tails”. Does it like a fair coin?

   HINT: Remember than you can control the range of generated numbers and what they mean.

---

## Part 2

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

### Learning objectives

This exercise is designed to help you learn about (and assess whether you have learned about):

- how to understand a problem.
- how to write an algorithm for solving a problem.
- how to write a small program in Visual Basic using variables, conditions, user input, text output.
- how to create test cases that can be used to verify the correctness of an algorithm.
- ??? compound conditions ???

As a practical matter, this exercise will help you become comfortable with the various policies (including submission policies and grading policies) that you must comply with while working on them and the various tools you will use while working on them.

Lastly, 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 and a maximum of 16 characters
- 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 many 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, it is may be 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 cracking program to check all such passwords. The total number of passwords of length $L$ that can be created from an alphabet of size $C$ can be expressed as:

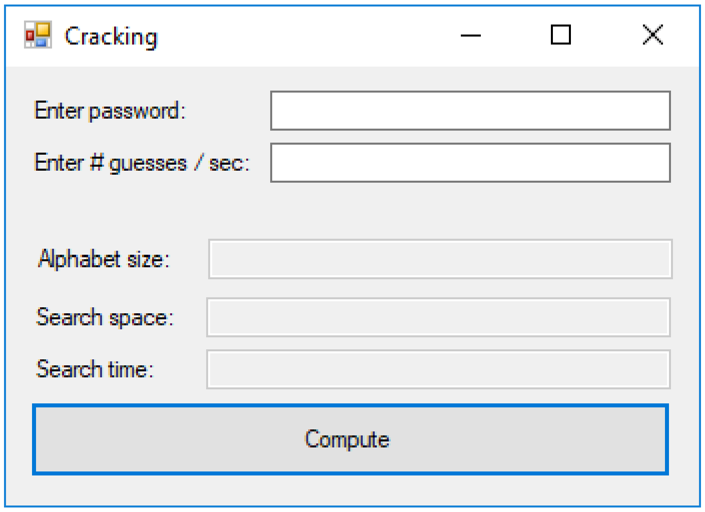
[1] $C^L$.

Using equation [1], the time it would take for a cracking program, capable of $G$ guesses per second, to check all passwords of minimum length $N$ and maximum length $M$ can be expressed as:

[2] $(C^N+C^{N+1}+\cdots+C^M)/G = C(C^{M\text{-}1})/(C\text{-}1)$.

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 search the entire password 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



You are to write a program, called `Craking`, using the example above as a starting point. For this problem, the alphabet size will be dictated by the password entered by the users and should contain a string of characters belonging to only the following character lists:

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

It will be your job to compute the total size of the alphabet based on the value of the user input password by increasing the alphabet size by an appropriate amount based on which classes it contains. In the example below, the alphabet size would be 26 because `PROTOTYPE` only contains lower-case letters, of which there are 26. For simplicity, we will assume that the minimum password length is always one and that the maximum length is the length of `PROTOTYPE`.

To find out if a `String` contains any one of a list of characters, you can use the `Like` operator. In this case, the `pattern` should be one of the following, depending on the character list your are checking for:

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

The characters `*` before and after the brackets enclosing the character set will include matches found anywhere in the `String` object that the method is being called on.

Your program should output **EXACTLY** two lines to the console. The first line should report the size of the search space, i.e., total number of passwords that can be constructed, and the second should report the time to search the entire space. The output format of your program should **EXACTLY** match the following example:

```
Search space size: 1558868884
Time to search:    2.57749 weeks
```

which can be produced by the following two lines of Java code:

```
txtAlphabetSize.Text = alphabetSize
txtSearchSpace.Text = searchSpace
txtSearchTime.Text = searchTime
```

where `size`, `time`, `unit` are three variables that represent the search space size, the time to search, and the units for the time respectively.

To make your program more user friendly, it should present the time to search using the most appropriate unit of measurement from the following list:

- seconds,
- minutes,
- hours,
- days,
- weeks,
- years, or
- centuries.

For example, if the number of seconds to search the space is less than 60, it should use the unit *seconds*, however if it is greater than 60, but less than 60×60=3600, then it should use the unit *minutes* and so on, all the way up to *centuries*. To keep things consistent, we will define a year as 52 weeks.

## Testing

One of the most important tasks when developing a piece of software is to test it to ensure its correctness. To this end, you should create a set of test cases to test your code under different conditions. For this exercise, the test cases will be different values of the variable `PROTOTYPE`. You should manually compute the search space size and time to search for each of the test cases, and then check your results against those of your program. Some examples of test cases might be:

- `"aaaa"`
- `"AAAA"`
- `"0000"`
- `"!!!!"`
- `"aA0!"`

Be judicious about your choice of test cases, so that you are not repeating calculations that will ultimately produce the same results. For example, it is probably not necessary to check both `"aaaa"` and `"bbbb"` since they define the same alphabet and have the same maximum length.

If you are satisfied with your set of test cases and your program is producing correct results, then your set of test cases is inadequate. If you have found a test case for which your program produces incorrect output, or you cannot come up with any such cases after thinking about it for some time, then you should download and run this testing program.

After downloading the testing program to the directory where you developed your own program, make sure that your program is compiled, and click the `Test` button in DrJava. This will run your program using a pre-defined set of test cases and report whether or not your program produced the correct output. The testing program should report two failures (test8 and test9). If it reports more, then you should address each of them until only these two remain.

## Improvements

---

D level

- get input from user
- output a simple message using input

---

---

C level

- add calculations and output for:
  - alphabet size

    - search space
    - search time
- deal with limitations of Integer and introduce Single data type for computing searchTime
- evaluate a set of test conditions to check correctness

---

---

B level

- add units to output
- deal with more limitations of Integer and introduce Long data type for computing searchSpace
- format output to 3 decimal places [lookup Math.Round online]
- identify and evaluate a set of test conditions to check correctness

---

---

A level

- ??? deal with limitations of Long for computing searchSpace [lookup BigInteger data type online]
- ??? blend if statements into a single regular expression
- do error checking to see if password contains any invalid characters [lookup more regular expressions online]
    - print error message and exit [lookup Close() online]
- identify and evaluate a set of test conditions to check correctness
- ??? rewrite unit if-ladder without compound conditions ???

---