**CS/COE 1501 Project 1**

**Goal:**

To demonstrate knowledge of both exhaustive search of a problem space and lookup search through the implementation of a very basic brute-force password cracker.

**Background:**

Let's assume that you are writing a password cracker that targets the following policy:

* Passwords have to be exactly 5 characters long (no shorter, no longer)
  + 1-3 of which must be letters (lowercase "a"-"z", no capitals)
  + 1-2 of which must be numbers
  + 1-2 of which must be symbols (specifically "!", "@", "$", "^", "\_", or "\*")
* Passwords must not contain any of the 500 most used English words (found in dictionary.txt), or any of these words with one or more numbers/symbols substituted for letters (specifically, "7" for "t", "4" for "a", "0" for "o", "3" for "e", "1" for "i", "1" for "l", or "$" for "s") Your program will have two parts. First, it will enumerate all valid passwords (i.e., those that abide by the above policy), timing how long it takes to get to each one. Second, it will allow the user to enter passwords and report how long your cracker took to guess that password. If the entered password is not valid, your program will select 10 alternative valid passwords and report their times.

**Specifications:**

* You must implement a De La Briandais (DLB) trie data structure (as described in lecture) to use in your project.
* Your main program should be called from the command line with one of two command line arguments, either "-find" or "-check" (e.g., "java pw\_check -find" or "java pw\_check -check"). Your program should first be run with the "-find" argument before being run with the "-check" argument.
* When called with the "-find" argument, your program should generate the list of all passwords and record the time required to reach each one (in milliseconds).
  + You are provided a list of dictionary words to check in dictionary.txt. Use this file to populate a DLB trie with strings that cannot be contained within user passwords.
  + Use exhaustive search and pruning rules to find all valid passwords. Be sure to carefully choose your pruning rules! Your search needs to be as efficient as you can make it.
  + Time the running of your program to see how long it takes your program to generate each possible valid password.
  + You must write the list of valid passwords and the time required to find each one out to a text file called "all\_passwords.txt".
    - Each line of this file should contain the password and then the time in ms separated by a comma.
    - E.g., q!23z,2.34567
* When called with the "-check" argument, your program should prompt the user to enter passwords until they wish to stop. You should check that a list of all valid passwords has already been generated (i.e., your program has already been run with the "-find" option and "all\_passwords.txt" exists).
  + For each entered password, there are two options: it is either a valid password, or it is not.
    - If it is a valid password, inform the user how long it took your cracker to guess that password.
    - If it is not a valid password, find 10 valid passwords that share the longest prefix with the entered password and inform the user how long it took your cracker to guess each of these passwords.
  + To facilitate these checks, you should create a symbol table that stores the key/value (password/time) pairs from "all\_passwords.txt" in such a way that it makes it most efficient for the user to continually enter passwords and get timing reports back.
    - Create a new file called "ST\_approach.txt" that both describes your approach to implementing this symbol table and justifies your decision to take this approach. Note that this file does not need to be extensive, just a few lines so the TA is aware of what to look for in your code and why you chose this approach.

**Submission Guidelines:**

* **DO NOT** add "all\_passwords.txt" to your git repository. This must be generated by calling your program with the "-find" command line option.
* **DO NOT SUBMIT** any IDE package files.
* You must name your main program file pw\_check.java.
* You must be able to compile your program by running "javac pw\_check.java".
* You must be able to run your program by running "java pw\_check [ -find | -check ]".
* You must fill out info\_sheet.txt.
* Be sure to remember to push the latest copy of your code back to your GitHub repository before the the assignment is due. At the deadline, the repositories will automatically be copied for grading. Whatever is present in your GitHub repository at that time will be considered your submission for this assignment.

**Additional Notes:**

* Since passwords should only contain lower case letters, your program should run in a case insensitive manner. Any capitalized letters in the dictionary or user input should be converted to lower case letters.
* Generating all valid passwords may take some time. Be sure to give yourself plenty of time to debug your program with this in mind.
* The list of all valid passwords/times will be rather large, be sure to account for this in devising a solution to implement your symbol table.
* To increase the efficiency of your search for valid passwords, be sure not to perform unnecessary dictionary checks.
  + Also keep in mind that valid passwords cannot contain dictionary words anywhere, e.g., "!and5" is a bad password because it contains the word "and".
* You only need to keep in mind the number/symbol letter replacements listed above when looking for dictionary words: "7" for "t", "4" for "a", "0" for "o", "3" for "e", "1" for "i", "1" for "l", or "$" for "s". You do not need to consider any other substitutions
* Your DLB implementation must be all your own code, you cannot build the trie around Java builtin data structures such as the ArrayList or LinkedList classes.
* You can use Java's System.nanoTime() (<https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime-->) to calculate the runtime of your program.

**Grading Rubric:**

* DLB trie implemented as described in class: 20
* Valid passwords are correctly enumerated via exhaustive search and pruning: 20
* Timing is recorded appropriately: 5
* all\_passwords.txt is properly formatted: 5
* The exhaustive search is sufficiently efficient: 15
* Symbol table implementation is sufficiently efficient: 10
* Program responds correctly to user-input bad passwords: 15
* Program runs appropriately and command line arguments are properly handled: 5
* Assignment info sheet/submission: 5