P09 Password Cracking

Overview

Password cracking is a process that can be used to recover passwords. It could be used to recover a lost password, but its use in most cases is for nefarious actors to gain unauthorized access to a system or account.

The *simplest* way to do this would be to brute-force all possible combinations of characters. This gets out of control quickly, as a password that is six characters long and uses uppercase letters, lowercase letters, and numbers has over 56 billion possibilities. We are going to take a more effective approach knowing that (1) lists of passwords are **leaked** from data breaches (2) some passwords are **stronger** than others and (3) some passwords are used extremely **frequently**.

Using this information we are going create and store passwords into a binary search tree (BST) and choose one of **three different** criteria to order the tree by based on how we want to attack the system:

- the number of times the password has been used
- the strength of the password
- the SHA-1 hash of the password (a way to obscure the actual password)

Warning: Do NOT use this password cracking on an actual system, you will get in trouble. This is an exercise for educational purposes.

Learning Objectives

The goals of this assignment include:

- **Implement** common binary search tree (BST) operations
- **Design** test scenarios and **detect** errors in BST implementations.
- **Organize** code using recursion.
- (BONUS!) **Explain** the concept of "strength" with respect to passwords and how it relates to computer security.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P09 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 04/23/2023**.
15 points	**Immediate Automated Tests:** accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. For P09 these tests will primarily focus on the robustness of your **tester methods**. Passing all immediate automated tests does NOT guarantee full credit for the assignment.
20 points	**Additional Automated Tests:** these will also run on submission to Gradescope, but you will not receive feedback from these tests until after scores are published.
10 points	**Manual Grading Feedback:** TAs or graders will manually review your code, focusing on commenting, style, and **recursive** implementation.

Assignment Requirements and Notes

(Please read carefully!)

- Pair programming is **NOT ALLOWED** for this assignment. You must complete and submit P09 individually.
- **The ONLY external libraries** you may use in your submitted files are:

```
import java.util.NoSuchElementException (Tester & PasswordStorage)
import java.math.BigInteger (Password)
import java.security.MessageDigest (Password)
import java.security.NoSuchAlgorithmException (Password)
```

- The only class that can contain a `main()` method is `PasswordStorageTester.java`.
- No additional data fields (static or non-static) or public methods, as per usual.
- Local variables and private helper method are okay, like always.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- As always, your assignment should follow the CS300 Course Style Guide and you must follow Academic Conduct Expectations and Advice.

1 Getting started

- To get started, let's first create a new **Java17** project within Eclipse. You can name this project whatever you like, but "**P09 Password Cracking**" is a good choice. As usual, make sure that you are using **Java 17**, don't add a module, and that you use the default package.
- Then, download and add following starter files to your project (links found on the assignment page on Canvas):
  - `Attribute.java` (an enum)
  - `Password.java`
  - `PasswordNode.java` (BST node)
  - `PasswordStorage.java` (BST)
  - `PasswordCrackingTester.java` (Tester)
- Take a moment to read through these provided classes before you continue. You will NOT need to create any additional files for this project.

2 Complete and test the Password class

At each node of our BST we will store a password string and some other information that we will use to decide how to structure our tree to fit our password cracking attack.

Almost all of it has been implemented, **but** there is one (1) remaining method YOU must implement: `compareTo()`.

Keep in mind:

- The **strength** rating is determined based on the number of different types of characters used and the length of the password using the following formula:

  $$charSetsUsed * 1.75 + length$$

  Ex. A password with only lowercase letters of length 8 will have a weaker (lower) strength rating than a password with both lower and upper case letters of length 8. "password" < "Password"

  Ex. If two passwords both use *lowercase letters*, *uppercase letters*, *numbers*, and *special characters*, but one is of length 9 and the other has length 11, the one with length 11 will have a higher strength rating.

- This `compareTo()` method works **differently** from `Comparable`'s: it requires an Attribute to compare on. Consider the following two passwords:

Password1:
password(5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8): 1000 [9.75]

Password2:
StronkPass12#(a279a9218ecca0dda2a99ddd5437ac0a6525e16e): 23 [20.0]

The `compareTo()` results for these two Passwords vary based on the Attribute given as a parameter:

  - OCCURRENCE: Use the number of occurrences of each password. In the above example, "password" > "StronkPass12#" since 1000 > 23.
  - STRENGTH_RATING: Use the strength rating of each password. In the above example, "password" < "StronkPass12#" since 9.75 < 20.0.
  - HASHED_PASSWORD: Use the hashed version of the password String comparing based on lexigraphical (alphabetical) order, as determined by String's `compareTo()`. In the above example, "password" < "StronkPass12#" since "5" < "*a*".

- `Password` objects are considered equal in **two (2)** different scenarios:
  - If ALL data fields match (from the `equals()` method). Use this in your tester methods to check that the correct Password object is returned.
  - If the Attribute in question matches (from the `compareTo()` method). Use this when implementing the BST.
- You can calculate an SHA-1 hash of a String at this site to help with your testers.
- If you are stuck coming up with some passwords for your tester, feel free to look up a list of the most commonly used passwords. *Do NOT use any of your own passwords.*

You will also need to implement the `testPasswordCompareTo()` tester method. (You are NOT required to verify the methods we implemented, though you *may* do so to verify your understanding.)

IMPORTANT – CONFIRM that your `compareTo()` works *before* you continue. If it does not, when you get to the more complex BST stuff you will be very sad and frustrated trying to debug it :(

3 Complete and test PasswordNode

The provided `PasswordNode` class will help organize data within the BST. This class specifically stores ONLY `Password` objects. Some of the basic methods (constructors, setters, getters) have already been implemented for you. There are **FOUR (4)** additional methods you will need to implement, that will be very helpful for completing the BST and its operations – see the provided file for details.

You must also implement the `testNodeStatusMethods()` tester method to verify the behavior of these four methods. (Again, you are not required to verify the provided methods.)

IMPORTANT – CONFIRM that your methods work *before* you continue. If they do not, when you get to the more complex BST stuff you will be very sad and frustrated trying to debug them :(

4 Complete and test PasswordStorage (the BST!)

Now to the the primary component of this assignment, the binary search tree implementation!

The ordering and operations of the BST will depend on its comparison criterion (an Attribute; see section 2 for details). Additionally, be aware that this is NOT a "balanced" BST; the structure of the tree will depend entirely on the order in which passwords are added (when using the `addPassword()` method).

However, testing a BST does NOT require the `addPassword()` method to be implemented! You can manually create a structure of PasswordNodes linked together using either the three-argument constructor or the set methods (see the provided testers for `toString()` and `isValidBST()`). We *strongly* recommend designing the intended tree structure on paper FIRST.

All the details and javadocs for the required methods are in the provided file. We advise completing the BST methods and their respective testers in roughly this order:

1. Constructor, `size()`, `isEmpty()`, `getComparisonCriteria()`
2. `toString()` and `isValidBST()`
3. `lookup()`, `getBestPassword()`, `getWorstPassword()`
4. `addPassword()`
5. `removePassword()`

Any method that states it is implemented recursively MUST be implemented recursively. (Iterative solutions will receive no points in manual grading!) If not specified, you may implement it recursively OR iteratively.

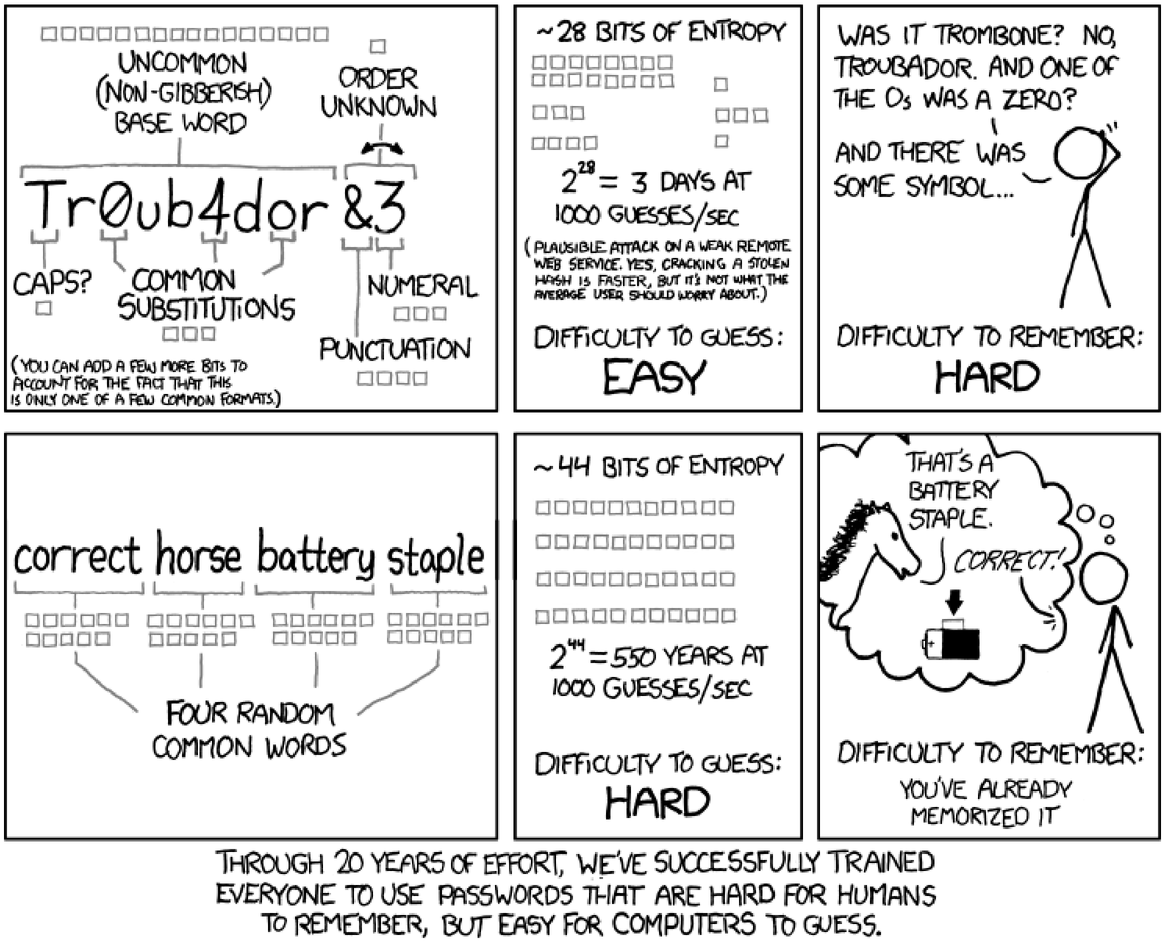
Be sure to test that your BST methods work for ANY comparison criteria; again, see the provided testers for `toString()` and `isValidBST()` for examples of how to build a BST without using `addPassword()`.

5 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only FOUR files that you must submit are:

- `Password.java`
- `PasswordNode.java`
- `PasswordStorage.java`
- `PasswordCrackingTester.java`

Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of **9:59PM on April 27th**. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline.


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.