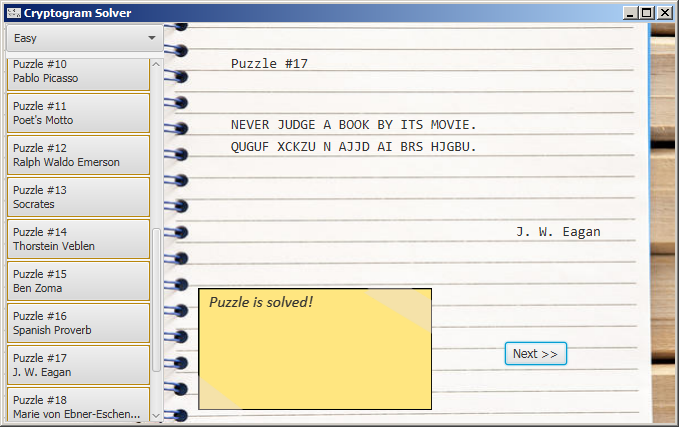
Cryptogram solver report

# Carolina kimbrough & zoey abrigo



## aBOUT

Using logic, we aim to solve cryptogram puzzles.

### What are Cryptograms?

It is a type of puzzle that consists of a short piece of encrypted text. Typically, a substitution cipher is used, where each letter is replaced by a different letter or number.

## Goal

Originally, our goal was to solve at least one hard puzzle, all easy puzzles, and a couple of medium puzzles using methods a human might use to solve a cryptogram.

However, due to time constraints and the nature of computers and natural language, we’ve made our goal a bit less ambitious: solve at least one medium puzzle and all easy puzzles.

## method

The language employed here is Java. In addition, this program utilizes a GUI interface, specifically utilizing JavaFX. The program is organized in a MVC format, model-view-controller, with **CryptoMain.fxml** as our view, **CryptoMainController.java** as our controller and the following classes in as our model:

* + **Dictionary.java** – Intended to house data and characteristics of words, contains an actual dictionary of the words collected from our quotes. This is an attempt to simulate knowledge a human being would have.
  + **PuzzleBox.java** – Contains the actual set of puzzles. Most importantly, contains the method to make puzzles out of the quotes.
  + **Quote.java** – The most busy part of the program. Originally it housed just the quote and its cryptotext equivalent but over time has developed to contain multitudes of data for each quote. It contains easy, medium and hard versions of the quote, hashmaps for character frequency, and keys that convert a character to its crypto version and back. There’s also information about the quote’s author, its length and it also has arraylists of the words in the puzzle along with its crypto equivalent and of the easy, medium and hard versions of the puzzle.
  + **Solver.java** – This part of the program does the problem-solving. An instance is generated for each puzzle and for each difficulty. This has info a human puzzle-solver might have such as a dictionary, the current quote, and various kinds of info about the quote puzzle including its difficulty and the characters found as well as a running solution in String form. There is a function for each puzzle-solving method.
  + **Word.java** – Contains information on each word

### Coming up With the puzzles

Using a .csv file of quotes from this website: <https://litemind.com/favorite-quotes/>, python was used to parse the quotes and narrow down our sample to around 40 puzzles with each quote being less than 46 characters long. Of the ~40 puzzles, only 20 are displayed in the actual program, however, all were utilized under the covers as the basis of our dictionary which was used to help the program to solve the puzzles.

The parsing was done in a class method inside PuzzleBox.java. The format for the text file is as follows: Quote goes here|Author. The python code and the quotes file are both located inside the quotes folder at the top level.

### Coming up with the Cryptograms

Because we’re dealing with a substitution cipher, this was actually pretty simple to implement. In order to simplify solving, we converted all letters to uppercase, so we could just focus on the 26-letter set. First, we created an array of 26 characters for each letter of the alphabet where index 0 corresponds with ‘A’, index 1 correspond with ‘B’ all the way to index 25 corresponding with ‘Z’. We also made another array just like this, specifically it was implemented as an ArrayList. After this, we shuffled the ArrayList and that was how we came up with our key, allowing us to convert regular letters to our ‘crypto’ version of a letter.

This was done by the encrypt() method in Quote.java.

### Coming up with Easy, Medium & Hard Puzzles

When setting the character limit for each puzzle to be max 45 characters, this also included special characters and spaces as well. If we wanted to come up with different difficulty settings for each puzzle, our solution would need to be dynamic. This is because choosing a flat number of characters to reveal (10 for easy, 5 for medium, 3 for hard) would make longer puzzles more difficult and shorter ones more easy. Not to mention, the case where sentences might have a smaller set of letters (for example, ‘She sells sea shells by the sea shore’ which is long but has many of the same character).

Our solution to this was to choose a percentage of characters to reveal:

|  |  |
| --- | --- |
| Difficulty | Characters shown |
| easy | 70% of the set |
| medium | 50% of the set |
| hard | 20% of the set |

But which characters do we show? The least frequent ones! So, we made a hash table of letters to the number of times they appear in the quote and sorted it by least frequent. Using the percentage above we picked the top percentage as the characters to reveal.

This was done by the findCharFreq(), toEasy(), toMedium(), toHard() methods in Quote.java.

### Solving the Actual puzzles

#### a problem

Initially, we had wanted to use the methods to solve our cryptogram that a human being would use, such as the methods on this site: <https://www.cryptograms.org/tutorial.php>

However, a problem with this is that in order to say, pick from a list of two-letter words, the way to narrow down an answer would be to understand the context of the sentence. For a computer, this is not very straightforward and could possibly involve having to tag parts of speech and a deeper understanding of the English language that would have to be ported over so that a computer could understand.

Additionally, and more fundamentally, how does a computer know what a legitimate word is? A computer understands 3,938 but it doesn’t know that ‘mptipn’ is not an actual word or that ‘motion’ is an actual word.

#### A solution

For the fundamental part, we solved that problem by giving the computer its own dictionary of words parsed from the actual set of quotes it’s setting out to solve. However, this is from the set of ~40 quotes so while, like a human, it now knows what an actual word is, its knowledge is limited to just the words in those quotes, making puzzle-solving a bit easier for it. A well-read person would have a slightly more difficult time since the breadth of their vocabulary is larger.

As, for the first problem, rather than try to solve these puzzles like a person, we just use what a computer excels at: speed. While a person might hem and haw over a possible solution, a computer can quickly iterate through a list of possible solutions and narrow it down using different rules.

Dictionary.java and Word.java both aid in this. Dictionary.java houses the actual vocabulary as well as some additional rules. Word.java has the potential to give each word in the quote more information.

Using these tools, we were able to proceed with finding ways to solve these puzzles.

#### Solver

The part of our program made to solve the puzzles was Solver.java. In it, there are the main methods for solving the puzzles and additional subroutines to modularize some of the repetitive bits of code in the problem-solving methods. Also, for each successful answer found, a count is made so as to report back to CryptoMainController.java for display.

Many of these methods involves comparing possible solutions with those in our dictionary, it matches every letter in the puzzle, ignoring ‘\_’ and other special characters.

Currently there are four methods for solving a puzzle:

* **checkforFilledInOne()**
  + This searches for words with one empty space (i.e. ‘BOO\_’ or ‘TH\_’)
  + Compares this word with our dictionary to aggregate possible solutions
  + If there's just ONE solution, then it is the answer.
* **checkforFilledInAndNeighbors()**
  + This searches for words with one empty space (i.e. ‘BOO\_’ or ‘TH\_’)
  + Compares this word with our dictionary to aggregate possible solutions
  + Next, it narrows down the solutions by getting rid of solutions which attempt to use characters already in the solution set
  + If there's just ONE solution, then it is the answer.

These two methods are pretty similar. The first one just uses comparing to find a possible answer. The second one is a little more smart by looking at surrounding words and their solutions.

* **checkforContractions()**
  + Searches for words that are contractions (words with apostrophes, basically)
  + Compares this word with our dictionary to aggregate possible solutions
  + Next, it narrows down the solutions by getting rid of solutions which attempt to use characters already in the solution set
  + If there's just ONE solution, then it is the answer.

This one came from the idea that there’s a smaller subset of word that are contractions, so finding matches is more likely. However, because the current set doesn’t have many contractions, this method doesn’t get a lot of flexing. Also, there are no safeguards against false positives, so it might produce strange results in harder puzzles.

* **checkforTwoAndSearch()**
  + Searches for words with two empty spaces (i.e. ‘ACC\_RAT\_’ or ‘Y\_\_TH’)
  + Compares this word with our dictionary to aggregate possible solutions
  + Narrows down solutions by checking the possible solution’s effect on neighboring words
    - Does it produce words that are actually words?
  + Each word in the new possible solution is checked.
  + If each word passes this test, a solution is found

Perhaps the most ambitious method. This method uses a little bit of search to find a possible solution. Words with two empty spaces are chosen because three could produce some odd results and there’s more than enough method to solve words with one empty space. The way this algorithm works is that for each possible solution generated by choosing two characters, each word is checked to see if it’s an actual word. In this method, there’s a corresponding array of 1’s and 0’s with 1 being true, this is a possible word, and 0 being false, this isn’t an actual word or does not resemble one in our dictionary. Each word should have a corresponding 1 to pass the test. This method could also produce some strange results in more difficult puzzles.

### RESULTS

So far, these are the results of our methods:

* **KEY:** 
  + O – Solved – This puzzle looks just like it’s quote
  + X – Not solved – Either no progress was made, or it got stuck
  + M – mixed results – Produced strange results that diverged from the solution

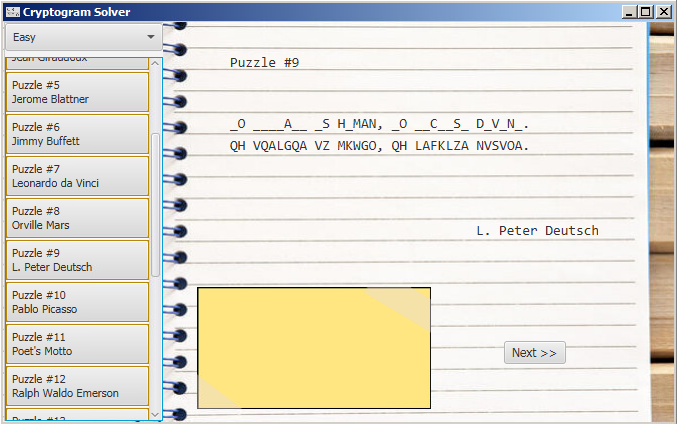
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Puzzle# | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Easy | O | O | X | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | X |
| Medium | X | O | X | O | O | O | O | O | O | O | O | O | O | X | O | O | M | M | O | X |
| Hard | X | O | M | X | X | O | X | M | X | M | X | X | X | X | O | X | M | X | X | M |

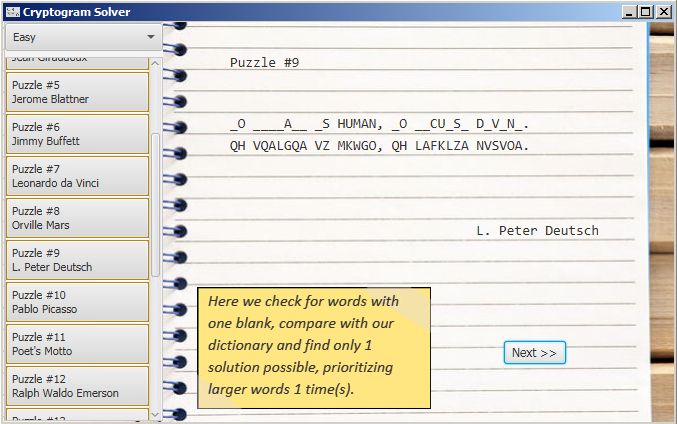
In total, 18 easy puzzles, 14 medium puzzles & 3 hard puzzles were able to be successfully solved.

#### Did we reach our goal?

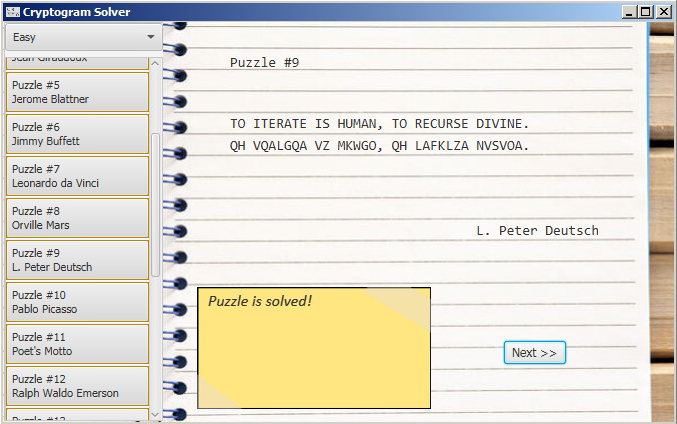
Sorta. We were able to solve at least one medium puzzle and then some, but not all of the easy puzzles were able to be solved.

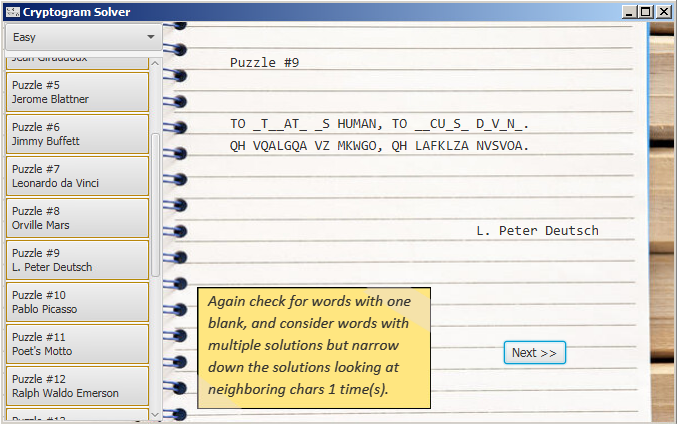
### Output

Here is an example of our GUI in action:

Here is the puzzle in its beginning state.

The puzzle after using the checkforFilledInOne() method

The puzzle after using the checkforFilledInAndNeighbors() method

The puzzle is solved!

text