Benjamin Garcia

Monoalphabetic Substitution Cipher

Decryption Design:

The approach I took with decrypting the message was looking at word frequencies and trying to piece together the key from there. I started with the word “the” since it’s the most common 3-letter word, and it also has ‘t’ and ‘e’, the two most common letters. With that as a starting point, I could keep looking for more frequent words and, using previously determined letters, slowly figure out what the key was.

This, of course, wasn’t sufficient. There are a handful of letters that don’t appear in common words such as ‘z’ and ‘x’. There was also the problem that there was a certain amount of guessing and hoping, even with using letter and word frequency.

To address the first issue, at the end of my program, I just fill in the remaining letters strictly by letter frequency. It’s not especially accurate, but it’s better than not guessing those letters in the key. Also, I only guessed one letter at a time, then re-ran my word checks to make sure nothing contradicted when guessing for a new word. If there were any contradictions, I reset the letter to unknown, and tried again with a different guess.

For the second issue, I implemented some overlap in my guessing. That is, I would use multiple words while trying to guess for a single letter. For example, when guessing for the letter ‘f’, we can examine two different common words: “?or” and “o?”. If the symbol being used is the same for both of these common words, we can make the assumption that the unknown symbol is probably ‘f’.

Accuracy:

Even after taking these measures, my program usually doesn't have 100% accuracy when determining the key. It does get pretty close, like around the 80% mark or so. So the decrypted message is fairly intelligible to a human. If I were to work more on the project, I think I would implement a spell check for the message after decryption, and then randomly reassign the letters that I was most unsure of, until I had an accuracy above 90%. That seems like it would take a lot of time though, so I didn't go about implementing it.

The accuracy, of course, improves on larger inputs. When using inputs only around 500 letters, the guessed key is quite poor. When approaching 2,000 letters, it gets better, and most of the time the outputed key is 70% accurate or higher. On inputs of about 5,000 letters, the accuracy approaches 85%, but it doesn't seem to go any higher than that, even on extremely large inputs (like 10,000+).