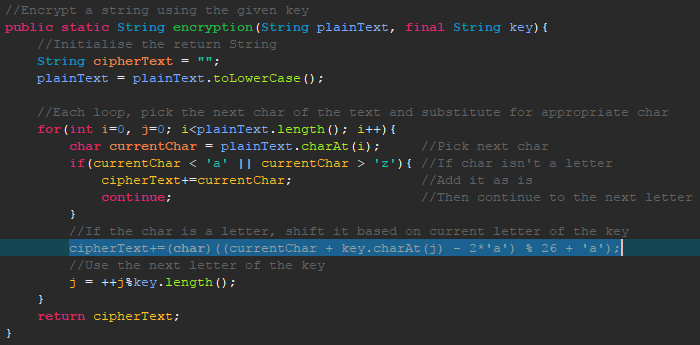
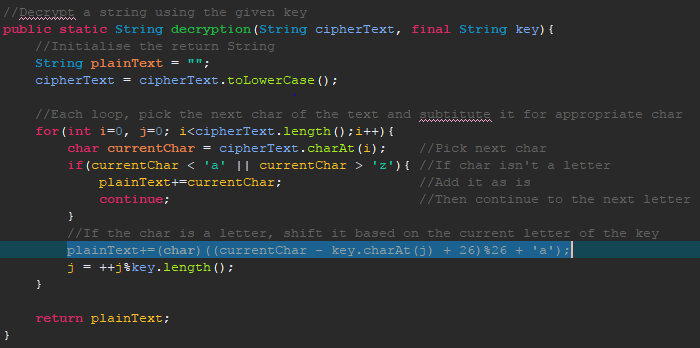
**Exercise 2 Report**

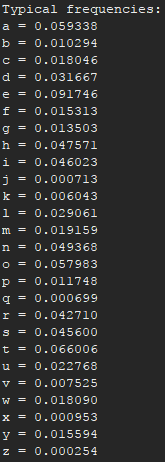
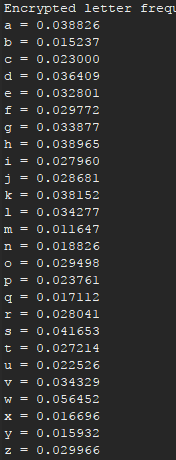
The first part of exercise 2 was simple to implement in Java. Encrypting and decrypting a text file use very similar methods, only really having one line different between them.



Notice how the encryption adds the currentChar and key.charAt(j), to shift the letters to the right, where decryption subtracts key.charAt(j) from currentChar. The encryption needs to subtract 2\*’a’ to use the numerical value of the letters (i.e. a = 0), rather than the Unicode value (i.e. a=65). Decryption adds 26 so if currentChar is less than key.charAt(j), the value will be looped back around the end of the alphabet. Both methods use %26 which ensures that the character is in the range 0-25, then ‘a’ is added to make it a char again.

My main method takes a password from the user, then encrypts ‘pg1661.txt’ using this as the key, writing the resultant encryption to ‘encrypted.txt’. It then decrypts the encrypted file and write it to ‘decrypted.txt’.

Using the key ‘password’ produces a different frequency weighting to English text. In English text, ‘e’ is the most frequent letter and ‘z’ the least frequent, but in the encrypted file, ‘w’ is the most frequent and ‘m’ the least. Also, the frequencies of the encrypted text are a lot less spread out, showing that there hasn’t just been a shift in the letters.



There is one thing I’m unsure about in my program, which is that at the start of a decrypted text file, a ‘?’ is the first character. I believe this may be due to an unreadable character.



Part two of this exercise took some time to implement. Since the key and its length are unknown, it isn’t obvious how to decipher the text. As such, I began by having a loop go through possible key lengths, from 1 up to 10. This is changeable in the program, but I guessed that the key wouldn’t be longer than 10 letters (and it wasn’t). Either way, keeping the key length short is helpful, as a longer key would mean smaller blocks which would take much longer to decrypt.

The first part of this loop splits the ciphertext into substrings according to the key length. After this, each substring is ‘deciphered’ by a reverse Caesar cipher function. This uses the commonly known letter frequencies of typical English text and compares it to the frequencies of each block. By doing this, we can narrow it down to the closest shift value for each encrypted block. Similarly, by comparing the ‘closeness’ of each string for every possible key length, we narrow it down to the closest likely key length.

Once every block has been deciphered, the strings are recombined to form what may be the original plaintext. Then, again, we compare how close the frequencies of this text are to typical English, and if it’s closer than any previous ‘guess’, make it the current contender.