Now that we have a basis for the algorithm implemented on this site we can begin to discuss the implementation that was used. The original algorithm was written in Python, but for the purposes of this site it was converted into javascript. Therefore, for this description I will focus only on the javascript code.

First, I set some global variables that can be called on by any function. These included an array with twenty-six elements, each of which was a lowercase letter in the English alphabet, and another with twenty-six elements, each of which was a probability associated with a corresponding letter. That is to say, the probability value in index 0 corresponded to the probability of the letter "a". Next I imported two dictionaries. The first had digrams (or bigrams) as the keys and the natural log of their corresponding probabilities as values. The second was very similar, but had trigram keys and values. These two dictionaries are not relevant for any of the implementation of the previously discussed methodologies, but will be used later on.

Now that I had my values set, I began by encoding a function that counted the frequency of each letter used in a given string. The function took in two arguments, a string and an alphabet (an array, by default set to the array described earlier containing the English letters). For each element of the alphabet array, the function counted the number of times that it showed up in the string argument. This value was added to a new array of frequencies. In this manner, the frequency at a given index corresponded with a letter at the same index in the alphabet array. By default the frequency at index 0 would correspond to the letter "a".

Next, I coded a function for the Kasiski test. This function takes in four arguments. The first is a string, the second is a substring length (integer, set by default to 3), the third is a starting point (integer, set by default to 0), and the forth is a boolean set by default to false. The substring length is used to indicate the length of sequence of letters we would use to look for repeats in the ciphertext. The starting point corresponded to the index in the ciphertext where we will generate the substring from. For example, given the ciphertext

YSWBHXDOMSTJWJFCBYLDLWRNBTHXANYSKQH

YLDOWNYDMWXETZNFXBSCMSKRRKIWDSYRJE

we would begin by looking for repeats of the substring YSW. The function works by trying to match a section of the ciphertext to the substring. If a match is found, it records the index of the first letter of the match into an array. Obviously, the location where we got the original substring would be a match, so referring to our example we would get a result of [0]. This is because the only index at which the substring YSW appears is starting at 0. Next, the function would calculate the difference between each value in our array and the first location the substring shows up. In this case there is only one place the function shows up so our distance array is [0]. If the number of elements in the distance array exceeds 3 then the function returns the greatest common divisor of the elements in the distance array. If the number of elements in the distance

array does not exceed 3, the function returns itself, with a new value for the starting position argument. This value is equal to the value of the previous starting postion plus 1. So, we would find that our first iteration of the Kasiski test function on the example ciphertext would return the function with a starting position = 1. That means it now checks for repeats of the substring SWB. It continues in this manner until reaching the end of the ciphertext. If now substring was repeated at least four times (that is, so there are at least three locations for the subscript other than the first), then the function returns -1. For our example, the function would return -1.

Note: I am aware that the Kasiski test does not need to have so many repetitions of the substring in order to be effective. I put the repetitions safegaurd in place to prevent getting false results from the Kasiski test due to random chance. There is still the possibility of a wrong result, but this safegaurd mitigates that possibility. Also, regardless of the outcome of the Kasiski test, I still rely on the index of coincidence to double check the guessed string length.

Once I completed coding the function for the Kasiski test, I encoded a function to calculate the index of coincidence. This function takes in two arguments, a string and an integer value, m, corresponding to the key length being tested. By default $m = 1$. The function then splits the inputted string into m substrings of equal length. The first substring having the elements of the string at indices corresponding to 0 mod m, the second having elements from indices corresponding to 1 mod m, and so on. For each substring the algorithm calls on the function to calculate the frequency of letters, then uses the function

$$I_c(\mathbf{x}) = \frac{\sum_{i=0}^{25} f_i(f_i - 1)}{n(n-1)}$$

to calculate the index of coincidence for a given substring. The function returns the average value for index of coincidence for all the m substrings.

I then encoded a function that would find the most likely key, assuming some key length. This function took in three arguments, a string, an integer, m, key length, and an array of probabilities (in our case, the array described earlier with probabilities corresponding to the English letters). This function also broke the string into m substrings in the same manner as the index of coincidence function. Then, for all $0 \leq g \leq 25$ it calculated

$$M_g = \sum_{i=0}^{25} \frac{p_i f_{i+g}}{n'}$$

and found the g value for which M_g was closest to 0.065. This g value was added to an array which would eventually become the key array once a g was found for each of the m substrings. The function returned an array of numbers with each number being the likely key value.

Now that I had implemented the methodologies of cryptanalysis often used for the Vigenere cipher, I was able to code my first version of the decoding algorithm. This

function took in one argument which was a string corresponding to the ciphertext. First, the function called the Kasiski function on the ciphertext. If the returned value was a positive integer, k, then the index of coincidence function was called with $m = k$. If the result of this call was an index of coincidence close to 0.065, then we moved on to finding the key. If the index of coincidence was not close to 0.065 it must mean that there is some error due to random chance in our Kasiski test, and we we call the Kasiski test again, only starting with a different substring. If the returned value of the Kasiski test was -1, then we would have to find m another way. This was done by iterating through potential values of m starting with $m = 1$ and increasing (until some predefined max possible key length), and calculating the index of coincidence for each of these iterations. For any value of m where the index of coincidence was close to 0.065, the function added that value of m to a dictionary as a key, and the value for that dictionary entry was the corresponding key of length m calculated by using the function described earlier and the plaintext decoded using the guessed key. The function would then return this dictionary of potential key lengths with their corresponding guessed keys and plaintext.

As it turns out, when calling this function on the ciphertext from our previous example we would get a Kasiski test value of -1. Running through iterations of the Friedman test (index of coincidence) we find that the values for m with index of coincidence closest to 0.065 are $6, 9, 11, 12$. For this specific cipher I happen to know that the key is of length 6, so I will only focus on the result when looking at $m = 6$. The function would return the key: $[10, 5, 18, 10, 13, 20]$. Using this key to decode our cipher would give us

onerudtjuigpmensoebytmetropnntonsgue

bywmaethenrzpinnoyshaaexadetfehem

which is pretty clearly not English. Thus we see that using only the methods described thus far would not necessarily give an accurate key. Also, this version of function returned a dictionary of potential keys and plaintext. Ideally our function will just return the most likely key and plaintext pair, while being more accurate. This is where I introduced the optimization using digrams and trigrams.

The final cryptanalysis function works as follows. The arguments the function takes are the ciphertext (as a string) and the max key length (as an integer) if it is known. If the max key length is not known, it is by default set to be the floor value of the square root of the ciphertext length. The reason for this is that if the key length is to large relative to the ciphertext, then the analysis would not be effective. Also, the function returns an error if the ciphertext length (in terms of the number of English letters) is less than 30. This is because frequency analysis is not very effective for text that short. Once the function has been initialized as above, it begins with a Kasiski test. As in the original version of the function, it measures the index of coincidence for the result of the Kasiski test. If the index of coincidence is close to 0.065, it accepts

the value for the key lengths and skips ahead. Otherwise, if the index of coincidence is not close to 0.065 or the Kasiski test returned -1, then it begins testing the index of coincidence. This part of the algorithm runs the same way as before and creates a dictionary of potential key length values with their corresponding keys and plaintext. However, there is one change that was implemented. When finding the corresponding key, the function first uses the frequency analysis as described before to come up with a potential key, then it performs a frequency analysis based on digram frequencies. It does this by looking at the digrams of the ciphertext, and the most likely corresponding digram key sequence that would provide the best English text. When it has both a simple key using individual frequencies, and a key of digram sequences, it compares the values of the two keys and picks the one that optimizes the frequencies in terms of probabilities. Once this key is generated, a corresponding plaintext is created.

Finally, returning a dictionary of potential lengths, keys, and plaintext is not ideal for a cryptanalysis algorithm. Therefore, I use a trigram frequency analysis of the potential plaintext values. The plaintext that corresponds to the most likely set of trigrams is then chosen with its corresponding key and those are the values that the program returns.

Returning to our previous example, after getting the key $[10, 5, 18, 10, 13, 20]$, the digram frequency analysis returns the following possible set of keys

$$[[10, 5], [5, 18], [18, 15], [15, 13], [13, 5], [5, 24]].$$

This means that using the digram frequency analysis we expect the first key value to be either 10 or 24, the second key value to be 5, the third 18, the forth 15, the fifth 13, and the sixth 5. Since 10 appears in the original guessed key, optimizing for all frequencies we find that we would get the key: $[10, 5, 18, 15, 13, 5]$. Using this key we get the plaintext

onemustjudgemennotbytheiropinionsbut
bywhattheiropinionshavemadeofthem

which is plain English.