1. The ciphertext (you can download the file vig2.txt from the canvas) was generated using the Vigenere method with a key of length at most 6. Decrypt it. To this end build a suite of programs for breaking the Vigenere cipher.

a. Describe all the steps of the process and how the results guided you towards the key.

The source code can be found in the "vig" directory of the zip attached to the submission alongside this pdf. As it is a python script, all that is needed to run it is to execute the command "python3 main.py" from the "vig" directory.

This program first loads in the pre-given cipher text, then finds its key. For this it first uses the key_length function, which, when given the ciphertext, determines the most likely key length using the IOC process described here:

http://practicalcryptography.com/cryptanalysis/text-characterisation/index-coincidence/

and here:

https://pages.mtu.edu/~shene/NSF-4/Tutorial/VIG/Vig-IOC.html.

Using this process, the key length which produces the most "English-like" text is chosen. This key length is then used by the find_key function in calling the frequency analysis function to determine each character in the key. This is done through an application of a statistical variance function. As the FA function is given the English alphabet and the frequencies with which each letter of this alphabet occurs in English, sourced from here:

https://en.wikipedia.org/wiki/Letter_frequency

the variance formula is able to determine which character shift produced a text that is the most similar to English. That is to say, the least varied from English. The shifted text where each letter's frequency differs the least from English's is decided to be the correct one.

b. Comment on what is unusual about the plaintext. How did it affect the results?

The plaintext, as hinted at by the encryption key "noes," contains none of the character "e." This might prove challenging for a frequency analysis approach, as the letter "e" is among the most common in English. The results were unaffected by this, however, as the variance equation takes into account the frequency of every letter in the shifted text.

2. Implement the Blum-Blum-Shub algorithm for generating pseudorandom sequences of 0s and 1s (re-member that it requires the input of two specially chosen primes and a seed). Use the algorithm with p= 1000003andq= 2001911. Select a valid seed x (there are valid seed that are less than 10, pick one of them) and generate a sequence of length 100,000.

The source code can be found in the "bbs" directory of the zip attached to the submission alongside this pdf. As it is a python script, all that is needed to run it is to execute the command "python3 main.py" from the "bbs" directory.

The results below highlight the efficacy of BBS as a random sequence generator. Each sequence of 1000 bits, on average, is split nearly 50/50 in its composition of 0's and 1's, and each 4-bit sequence appears consistently between 1500 and 1600 times. (Note: 100000 / 4 / 16 = 1562.5) This demonstrates that the generated sequence does not contain any significant biases, as, at such a large sample of generated bits, we would expect these counts to converge where they seem to have.

a. Compute the average number of 0's in a subsequence of length 1000 over all such subsequences.

As output by main.py, the average number of 0's in subsequences of length 1000 is 500.76.

b. Among all subsequences of length 4 (there are 16 of them) tabulate the frequency of each of them occurring as a subsequence of the sequence you generated.

As output by main.py, the frequency of each subsequence of length 4 is:

seq : freq
0000 : 1578
0001 : 1524
0010 : 1582
0011 : 1570
0100 : 1557
0101 : 1612
0110 : 1512
0111 : 1513
1000 : 1596
1001 : 1530
1010 : 1537
1011 : 1564
1100 : 1610
1101 : 1561
1110 : 1607
1111 : 1547