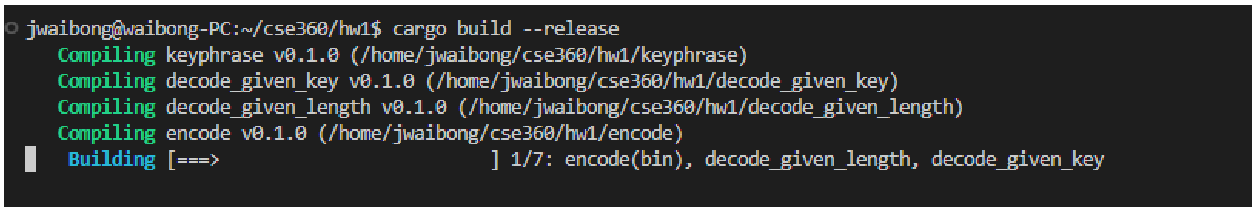
Jonathan Ng
113460607
CSE 360
Homework 1

If you wish to recompile the executable binaries, you must first install rust and cargo (easiest way is using rustup) from
https://doc.rust-lang.org/cargo/getting-started/installation.html

Then you must run the following command: 'cargo build --release'



```
jwaibong@waibong-PC:~/cse360/hw1$ cargo build --release
   Compiling keyphrase v0.1.0 (/home/jwaibong/cse360/hw1/keyphrase)
   Compiling decode_given_key v0.1.0 (/home/jwaibong/cse360/hw1/decode_given_key)
   Compiling decode_given_length v0.1.0 (/home/jwaibong/cse360/hw1/decode_given_length)
   Compiling encode v0.1.0 (/home/jwaibong/cse360/hw1/encode)
    Building [===>                        ] 1/7: encode(bin), decode_given_length, decode_given_key
```

All executables will be located in the directory /target/release/
And are the same ones specified in the "taskN.sh" files

(Also cited below) These the main code snippets I've used in my program (english quadgram data also taken from practicalcryptography.com):
http://practicalcryptography.com/media/cryptanalysis/files/ngram_score_1.py
https://stackoverflow.com/questions/30186037/how-can-i-read-a-single-line-from-stdin
http://cs.wellesley.edu/~fturbak/codman/letterfreq.html

Test cases found in /decode/src/main.rs were ones I found online from various sources (all found in README.md), including ones publicly available in the SBU CSE discord from past students. I also created my own test cases using a random paragraph generator https://randomwordgenerator.com/paragraph.php and "Gadsby by Ernest Vincent Wright – 50.000 words without the letter "e""
I used an online Vinegere decoder to verify my test cases.

TASK 1:
The source code for task 1 is located in:
/encode/src/main.rs

First, I had to parse two lines of input from stdin, one for the plaintext, and one for the key. I used a code snippet from stack overflow to help me achieve this (and similarly used in all other tasks).

Next, I created a data structure named “KeyPhrase” (located in /keyphrase/src/lib.rs) which would keep track of the current index of the key in order to encode the plaintext using the proper letters. The index would reset itself back to 0 once it reached the end of the key. This implementation saves memory by not having to generate a repeated string of the key equal in length to the plaintext.

Encoding the plaintext means building a ciphertext string. Everytime I encounter an ascii alphabetic character in the plaintext, I would encode it using our KeyPhrase object by calculating the value of the offset. Using some modular arithmetic, the program calculates the encrypted version for the current plaintext character (the main problem that had to be dealt with was when the offset produced a character beyond ‘Z’ or ‘z’). Finally, the println! macro prints the ciphertext to stdout.

TASK 2:
The source code for task 2 is located in:
/decode_given_key/src/main.rs
/decode_given_key/src/lib.rs

Similar to task 1, the first step is to read in two lines of input from stdin and create another instance of the “KeyPhrase” object for the given input key. Furthermore, decoding a given ciphertext ascii alphabetic character into its original plaintext character also involves using the current index of the key. The formula used to calculate the plaintext character was slightly different than in task 1, but still involved the same modular arithmetic. Once the plaintext was calculated, the program printed it to stdout.

TASK 3:
The source code for task 3 is located in:
/decode_given_length/src/main.rs
/decode_given_length/src/lib.rs
/keyphrase/src/lib.rs
Expected english letter frequency table taken from
http://cs.wellesley.edu/~fturbak/codman/letterfreq.html

Given an input length n, the program calculates the most probable candidate for the key using the chi-squared test. The program does this by first grouping all ascii-alphabetic characters in the ciphertext into n buckets (all other characters such as whitespace are ignored). Each bucket is a hashmap that keeps count of the number of occurrences of each character. The first character of the ciphertext would go into the first bucket, second character into the second,…, nth character into the nth bucket, n+1th character into the first bucket, and so on.

Once the ciphertext is split into n buckets, each bucket (ordered 1 to n) represents a group of text all shifted with the same Caesar Cipher. This means I can try to decode a given bucket with all possible Caesar Cipher shifts (26), create another character occurrence hashmap, and choose the one with the lowest chi-squared score (this represents the text whose frequency distribution matches most closely with English text). The actual calculation for the chi-squared test can be performed by calculating the ratio between the square of the difference between the actual count and the expected count (where the expected count is calculated using the length of the ciphertext and the expected English letter frequency table). Keeping track of a running sum for all 26 letters calculates the chi-squared score for the entire ciphertext.

I used http://practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic/ To help me understand how to implement this test.

Let's say that decoding the i'th bucket with a shift of 'C' results in the lowest chi-squared score. This implies that the i'th character of the key is 'C'. Doing this process for all n buckets gives us a good candidate for the key. This method does not always produce the correct result because the lowest chi-squared score does not imply that the determined shift was the one actually used.

Because of this inaccuracy, I also implemented quadgram frequency analysis to "error correct" the candidate key. **My code was highly based off of this python code:**
http://practicalcryptography.com/media/cryptanalysis/files/ngram_score_1.py
http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-vigenere-cipher-part-2/

Data of frequency of english quadgrams (located in english_quadgrams.txt) also taken from the same site.

In essence, creating an Ngram instance (located in /keyphrase/src/lib.rs) calculates all frequencies of every quadgram in the given file and takes its logarithm. For instance, let's say we have 1,000,000 total quadgrams, and "TION" occurs 13,168,375 times. The Ngram maps "TION" to log(13,168,375 / 1,000,000). The reason for the logarithm is because the ratio between occurrences and total count can get too small and imprecise.

Computing the quadgram score for a given ciphertext would mean to analyze all of its quadgrams (meaning all contiguous substrings of length 4). Looking at a particular quadgram in the ciphertext, I would look at its corresponding log probability in the Ngram object, and add it to a running sum. Overall, the higher the score, the more likely it is that the text is English plaintext.

The method that practicalcryptography.com suggests to finding a key using quadgram analysis only is to use a parent key starting with length 1 and continually finding better and better parent keys (and incrementing the length too). Since I already have a good candidate key using chi-squared, my code uses this as the parent key.

Using this parent key, I can loop for the length of the key and calculate the quadgram score of the plaintext when I change the i'th character of the key to another letter (I try all possible 26 letters). Doing this for all characters in the key, if I determine that no letters have changed, then I have found the best possible key. If a letter did change, I repeat the entire process starting with a new parent key.

My implementation relies on the fact that I have determined the correct length for the key (since this is task 3, we know it's the correct length). Once I determine the best key, I can use the decode function from task 2 to produce the plaintext and print it to stdout.

Task 4:

The source code for task 3 is located in:
/decode/src/main.rs
/keyphrase/src/lib.rs

The only difference between task 4 and task 3 is that I need to determine the key length before I can determine the key.
In order to determine the key length, I implemented another statistical analysis called index of coincidence. Inspiration of how to implement came from
http://practicalcryptography.com/cryptanalysis/text-characterisation/index-coincidence/

Starting from i=2 and incrementing by 1 each time (where i is our candidate length), I first group the ciphertext into i buckets (similar to how I did for chi-squared). I can calculate the index of coincidence of each individual bucket by calculating the likelihood that picking two random characters from the bucket would result in the same character. If the candidate key length is wrong, then I would expect that the index of coincidence matches closely with a uniform distribution. If the key matches, the index of coincidence would match the English frequency distribution (around 0.68). I calculate the average index of coincidence for all buckets and see if it matches around 0.68. If it does, I determine this as the best possible key length, otherwise I increment the candidate and try again.

This process is not perfect because it might find a multiple of the key length rather than the key length itself, but my program does not take this into account.
Once I determine a good key length, I can use task 3 to decode the ciphertext and output the plaintext.