Encode: “encode.c”

- buildtable -> In this method I decided to break it down into 4 steps.
    - 1: Fill encode with ‘_’ characters. This was an idea that would make keeping track of where I had been easier. Especially knowing that we are going to be dealing with wrapping indexes. Although slightly computationally heavier I would say that this is immeasurable and made the program more reliable.
    - 2: Remove duplicates in key, so I was able to cleanly copy the values of it into the encode table.
    - 3: Starting at the index after the last from the previous step. Skip values that have already occured in the encode.

Decode: “decode.c”

- buildencodetable -> The same method as previously explained.
- buildtable -> This method was very quick, given the encode table which I was able to copy and paste into the same file, all I had to do was a simple substitution for characters. Although this might not be the most computationally efficient it makes reading the code much easier to understand.

Crack: “crack-skel.c”

- main -> This method is responsible for calling and coordinating any method suck that it is able to calculate and print out cracked text. I chose to use 2d pointer arrays with malloc as they would allow me to cleanly pass them through to other methods as parameters. Which in turn let me let my program become more modular. This was broken into 5 steps.
    - 1: Repeat the following using 1 -> n keyLength.
    - 2: Allocate memory.
    - 3: Split the text into subtexts. This step although arguably not necessary it was really helpful for making code easier to write and read later on. Each of these subtexts can be cracked via frequency analysis individually.
    - 4 Frequency analysis on these subtexts. I decided to load characters into a mapping array for each subtext to make decoding easier.
    - 5: Decode the text using the generated maps. To make my program more robust I filter out all non letter characters such that I don’t reference invalid indexes.
- splitText: Splits a bigger text into keyLen amount of subtexts.
- frequencyAnalysis: Split into 3 steps
    - 1: Count frequencies of each letter.
    - 2: Sort the letters via frequency using bubble sort.
    - 3: Generate the decoding maps based off the letters sorted by frequency and aligning it with the ETA frequencies.

To make life easier, I wrote a few quick bash scripts to compile execute and compare the outputs for me. I will include them in my submission. This was very helpful with testing as well as I outputted logs to the same dir to help diagnose bugs.