NWEN243 Project 1

Part 1: Encode

The first step that needs to be done is removing duplicates from the key. I have done this in the fix key method so that the fix key method does everything to the key that is needed for it is encoded. For each character, it checks that it isn't already in the array and if it isn't then it adds to, before returning the fixed key.

With the new key it then puts the key into the array to encode. The key is put in by an offset of the size of the original key. Once the key is entered the rest of the array is filled in. It goes from the end of the key and continues through the alphabet to fill in the array. If it gets to Z it then goes back to A and when it reaches the end of the array it goes back to the beginning to fill in the rest. It also checks that the letter isn't already in the array so that it doesn't add letters from the key again.

Once that is done it is the key to encode the modified Caesar Cipher.

For encode, I have left most of the code in the build table method as that is the most relevant place to put it since it is building the table. The only part that is elsewhere is removing duplicates from the key, since that makes sense to do in the fix key method.

Since duplicates are removed from the key it means that the key can't be longer than 26 characters in the English alphabet so it can safely be put into an array of that length without causing problems. Beyond that the code is fairly robust as that is the only user input and most of the code is done with variable use. The only hard-coded values are for the size of the English alphabet which isn't going to change.

The code is fairly efficient. This is helped by the use of the method strchr as it can check for a character in an array much faster than using for loops would be. The code isn't very repetitive and doesn't have additional steps for creating the encode table.

For testing, I tested keys of various lengths and with various duplicate values in that all encoded correctly. This included keys that were the alphabet and keys longer than 26 characters long to test the unusual cases for this program.

Part 1: Decode

The decode starts off the same way as the encode, so the code has been copied in. It also fixes the key so there is no duplicates before creating the encode key. The decode it first fills a temporary array with all the letters of the alphabet in order for A to Z. From there to create the decode key, it calculates the index of each letter by finding it in the encode array. It then puts that number letter in the decode array by getting that index from the temporary array.

Once that is done, it is the key to decode the modified Caser Cipher.

Since the decode is very similar to the encode the code is in very similar places. Like the encode most of the code is left in the build table as that is where makes the most sense and removing duplicates from the key occurs in the fix key method.

Ignoring the code that is the same as the encode method, and has been copied from there, the code is fairly efficient. After creating the encode method, all the decode method does is fill an array with the alphabet and then map it with the encode table, which is made easier with the strchr method. This makes the code fairly efficient.

For testing, I did some testing of this code by itself but it is harder to tell if the code is correct than the encode method. I ran both the encode and decode together and the message was correctly decrypted with the keys were the same and not correct when they were different. Like with the encode method, I tested with various keys of various lengths and it all seemed to work.

Part 2: Crack

For the polyalphabetic cipher, the user enters how many keys there are. The program then loops through up to that number of times to try and decrypt the text.

The first step is to divide the text into the number of subtexts required. This is put into a 2D array so each array is an array of that sub text. Then, for each array the frequencies of each letter are counted, which is stored in another 2D array with the frequency of A being the first value in an array, but still separated by subtext.

From there, the most frequent letters and matched to the most common English letters which is then used to substitute the encrypted value for the English letter. Once this is done, the subtexts are then merged back together again and written to the file.

This process is repeated for different numbers of subtexts up to and including the key number entered with all the options being written to the file.

For this code, everything has been left in the main method. It could've been divided up into method but none of the sections of code are that large so having them in method and having to pass a number of variables around doesn't make a lot of sense. And with having it all in one place it makes it much easier to read when something has gone wrong as you can step through it.

This code is fairly efficient. There are a lot of for loops but that is unavoidable as the data is stored in a 2D array for most of the time and for loops work the best for that. The code is fairly robust as like with encode and decode most of the data in stored in variables, meaning that it is extremely flexible for different texts and key numbers.

For testing, I tested it with the various files that we had been given with various numbers of keys and it all seemed to work.