NWEN 243
Lab Report 1
Caesar Cyphers

Daniel Eisen
300447549

August 2, 2019

Encode

char *rmdups(char *s)

Initially, the after the key is 'fixed', ie removed non letters and capitalised, this function removes all duplicate letters from the key. I do this using strch to check if a char is already in an array, if not I add it. This has relatively low overhead as it only assigned to empty memory.

void buildtable(char *key, char *encode)

In build table, I loop through the fixed and non duped key, adding it into the (empty) encode table at the pre-calculated offset (length of unalterd key). This loop ensures full insertion as it checks if it has reach the last place and loops to the first to continue insertion.
Secondly I (starting at the last char of the inserted key, in the remaining space populate the rest of the encode table with chars, checking if; it's reached the end of the table (loop to beginning), the char to insert is Z (reset to A), and if it is already in table (loop, incrementing char until unique). This is a single loop, of n=26, so practically nothing in runtime.

Decode

void buildtable(char *key, char *decode)

Decode is simply done by reconstructing the encode table (from the known key) in exactly the same way as described in section Encode. Then I map the encode table, by looping from A $\rightarrow$ Z, to the decode table by successfully setting each value in decode (table) to the letter in the alphabet at the position of the current letter (of loop) in the encode table. This is essentially the 'inverse' of encode and thus using decode a cipher text can be decrypted.
This is not the most efficient method, as it relies on reconstructing the ecode table, ie added overhead but was the simplest to implement.

Crack

Crack is in 5 separate parts, each sectioned with separate c-doc comments in the main statement

- Splitting the text into n subtexts
- counting the occurrence of each letter in each subtext
- sorting a clean alphabet relative to the letter frequency in each subtext
- using these frequencies to 'decode' each char in the subtext according to known English letter frequencies
- Finally splicing subtexts back together to form full 'decoded' text.

Splitting

To split the text in into n subtext, I double loop [from i=0 $\rightarrow$ n, j=i $\rightarrow$ end of text], with outer representing the subtext currently being populated and the internal starting at the beginning of the text+i and grabbing every nth letter.
This quickly splits the text up in an efficient linear fashien with out an backtracking.

Occurrences

I have an array of ints to store frequency value, such that each index represents a different letter 0:A, 26:Z etc. This then just loops through each subtext, incrementing the right value at the correct index.

Sorting

This a standard and reasonable efficient sorting algorithm that is less than $O(n^2)$ due to internal loop j = i + 1. In essence it sorts the int arrays formed the previous section and transform an alphabet by the same transforms to create char array sorted by frequency.

Mapping

Mapping loops through each subtext, and for every letter, [if (isalpha(subtexts[i][j]))], get the index of that letter in the english frequency array and overwrite it with the value at that index (in eng frq)
This is very fast, single loop again, only affected by size of subtext.

Splicing

Is just the splitting in reverse.