Applied Cryptography

Project 1

Group: Daniel Suissa, Shuyuan Luo, Jessica Ko, Adam Kugelman

To run: In order to run the project, the best way is to throw all the files in an empty Visual Studio project, and run Project1EXE.cpp. **Enter the ciphertext into the file testInput.txt. The ciphertext must have a space character after the last number.** If the programing is running too slow, one way to reduce the running time is to change the constant KEYTRIES at the beginning of the Project1EXE.cpp file. Note that if the number is too small, we will get less accurate key.

Roles: Daniel Suissa wrote the rank functions. The rank functions included "rankDic" and "rankABC". The functions takes a vector of deciphered words and a key object and returns a value between 0 and 1. Shuyuan Luo wrote the main and findMax which find the best key. The main is responsible for reading the input find the best key and print output. It also calls the rank functions and mutate. Jessica Ko wrote the Key class. The key class makes a random key and maps each char to an int. Adam Kugelman put together the report. All team members were involved in debugging and putting all the different parts of the program together.

Cryptanalysis:

The specific permutation cipher that we are attacking in this project is an example of a homophonic substitution cipher. This type of cipher is characterized by an encryption scheme which maps specific letters of the plaintext being encrypted to multiple different possible ciphertexts. In some more secure homophonic substitution ciphers, the number of different mappings (as well as the possible mapping space) is not known, so the first step of a cryptanalysis in that situation would be to use a "hill-climbing" approach to first discover the mapping distribution: namely, given a plaintext character, how large is the set of possible outputs and what are they?

Fortunately, in our case this information is already known: the number of ciphertext values a given plaintext character can map to is given based on the frequency of the plaintext character's appearance in the English language. The letter "E" will have many possible mappings, while "X" or "Z" will have only one possible mapping. The mapping space is the range of integers {0..102}. Since we have this information already, we don't have to deal with the added layer of complexity of guessing what the number of possible mappings are and the mapping space, which would have resulted in greatly increased runtime. However, this scheme is still more difficult for us to attack compared to a situation where the mapping distribution was randomized: since the number of possible ciphertext values for a given plaintext character is pegged to the frequency of letters of the alphabet occurring, there's no way for us to try to detect a frequency peak from occurrences of the letter "E" in the ciphertext and gain information that would give us a leg up from that analysis.

Since the distribution of ciphertext mapping to plaintext is known, we can jump straight to our direct cryptanalysis of the cipher. Compared to a simpler cipher like a Caesar cipher, which has a very small keyspace that is easily brute-forceable, the number of possible keys for our homophonic permutation cipher is extremely large and it would not feasible to attempt every possible key in a reasonable amount of time. Instead, we will utilize the hill-climbing approach mentioned above. This strategy leverages the following procedure:

1. Generate a random key, use it to decrypt the ciphertext. Use some measure to determine the "fitness" of the resulting plaintext (see discussion of fitness calculation below).
2. Mutate the key slightly by swapping two characters of the key (at random or incrementally), decrypt the ciphertext with this new key, and measure the fitness of the resulting plaintext.
3. If the fitness of the plaintext resulting from decryption with the newer, mutated key is greater than the plaintext fitness that resulted from the previous key, store this new mutated key as the "best key so far" and try the next mutation.
4. Repeat, starting from step 2 -- unless we weren't able to improve our key in the last 1,000 iterations.

So how do we determine the "fitness" of the plaintext generated from decrypting some ciphertext with our randomly-generated key? We have to assign some sort of score in order to help gauge whether the key was closer to or farther from the true key than the last one we attempted. In our implementation, we take two different approaches to this problem depending on the context in which the cryptanalysis process is being run. If it's detected that the first test is being run, we know that the possible set of plaintexts is very small, so we can simply compare the plaintext generated by the key to each possible ciphertext letter-by-letter, and arrive at an overall score which represents the percentage of letters that matched up between the plaintext generated from our key, and the known plaintext.

If we detect that it is the second exercise being run, our job is a little trickier. The possible number of plaintexts is much higher, so we can't use the same trick we did in the first exercise without severely compromising the efficiency of the program. Instead, we try to make our best guess on the "fitness" of the plaintext based on how similar the decryption result looks like English. We can quantify this measurement by mapping the frequency that different "quadgrams", or groups of four letters, appear in the English language, or on a very large training set of English words. We break the resulting plaintext into quadgrams, and take the sum of their scores based on how likely the quadgrams were to be English. A high score would indicate that the plaintext looks (statistically) a lot like English, and a low score would indicate that the plaintext doesn't appear to be very English-like.

While we were very successful in our cryptanalysis of exercise 1, attacking exercise 2 was much more difficult for several of reasons: First, the composite fitness score of the calculated plaintext for any given key we generated didn't necessarily help to actually determine what the real plaintext input was if the plaintext was short. While we could say with some accuracy "this plaintext looks a lot like English", we wouldn't be able to tell if the original plaintext was "action" or "motion" or "notion" or "option", for example. This shortcoming, combined with the fact that there are many possibilities for words (or groups of words) in English that aren't common and aren't statistically very "English-y" at all, means the difficulty carries over even for longer plaintexts.

Secondly, our implementation of the "key mutation" step of the hill-climbing algorithm had issues both with efficiency and completeness. The efficiency problem (which we were able to mitigate fairly successfully) was that when randomly swapping just two characters in the key on every iteration of the hill-climbing process, there was no guarantee that we would even come close to exhausting all of the combinations in the allotted 2 minutes. We could be reasonably sure that the algorithm would complete *eventually*, but we wanted to ensure that our cryptanalysis completed in a timely fashion.

We mitigated this problem by “accelerating” the key mutation based on how close we were to a real plaintext match. If less than 5% of the characters were matched, we would mutate the key much more (swap more characters in the key) than if 20% of the characters were matched. This allowed us to quickly break out of bad starting points and local maxima and quickly arrive at a more favorable key.

The “completeness” problem was characterized by the fact that our implementation didn’t guarantee that the key mutations being carried out weren’t in fact reversing a previous mutation, essentially reverting the key to a prior iteration of the key that had already been checked and determined to be not close enough to the real plaintext. Theoretically, this error could happen at any point and in the worst case (especially with small ciphertexts and therefore small keys, although this situation is unlikely) could oscillate back and forth between two known bad keys. A solution would have been to calculate a list of all possible unique keys and to iterative through them, perhaps maintaining the same “mutation acceleration” solution detailed above. Our team did not implement this feature due to its computationally expensive nature and the fact that the cost of pre-calculating this list of keys increases exponentially with key size.