**Project 1: Cryptanalysis of Permutation Ciphers**

Rob Chiarelli and Salomon Idy

**Introduction:**

The purpose of this project was to implement a decryption strategy which is capable of breaking a permutation cipher in less than two minutes, with a message size of 500 ASCII characters.

The decryption strategy discussed in this report utilizes brute-force searching on portions of a provided dictionary, determined by word length. The search is optimized through the use of tries, which are search trees that link characters to form words.

**Decryption Strategy:**

The first thought that we had was to implement frequency analysis to break the cipher; however, since the relationship between ciphertext characters and plaintext characters was not one-to-one, this approach did not prove to be fruitful, and it was scrapped early in the brainstorming process.

After concluding that frequency analysis was not a viable strategy, we turned to brute-forcing the cipher key as our next option. The downside of this strategy was obvious from the start; with a keyspace of 102!, which is on the order of 10161, blindly attempting every key would be terribly inefficient and fail to decrypt ciphertexts in the requisite two minute time frame. As a result, we decided to implement a strategy which narrows down the possibilities for each word.

The first attribute of a word that is assessed in decryption is the length. By analyzing how many characters are in each word of the ciphertext, we narrow the options down to, at the worst, 19461 possible words, which is the number of 8-letter words found in the provided dictionary of English words. At best, a single word can be deciphered in one attempt because there exists only one word with either 25 or 28 letters.

The primary feature of our decryption strategy is the use of tries, i.e. search trees which contain one or more characters at each node. Before the actual decryption is performed, the program contains the entire dictionary in series of tries. By doing so, the brute-force algorithm for finding a word that fits a partially completed key can be run in significantly less time; if half of a word in the ciphertext has been deciphered, a trie makes it very easy to located words that fit that description without viewing the entire dictionary.

The decryption itself is broken into two distinct sections: the pre-processing stage and the processing stage. Any functions that can be performed before the program is exposed to the ciphertext is performed in the pre-processing stage. The details of what happens in each of the stages is described below.

Pre-processing stage:

* Sort the english\_words dictionary in order of word length, with the largest words being at the top of the list
* Create a vector containing 25 tries, each corresponding to one available word length
  + Available word lengths include 1-23, 25, and 28

Processing stage:

* Sort the words in the ciphertext in descending length order
  + If the first and/or second word contains either 25 or 28 letters, fill in the key appropriately
* Given the length of the largest word in the ciphertext, assign to it a random word of equal length from the dictionary
  + If there are any conflicts that make it impossible that the assigned word corresponds to the word in the ciphertext, e.g. there are two Z's in the word with key indices that are not the same, assign a different word from the dictionary
  + Otherwise, fill in the key accordingly
* Recursively travel through the trie of ciphertext words sorted by size, and perform the above step on each one
* If, at any point, conflicts occur, backtrack until they are resolved and travel down the next branch in the trie
* At some point, a key will be produced which correlates words from the dictionary with all words in the ciphertext

**Project Tasks:**

The first task that needed to be performed for this project was coming up with the idea for an efficient decryption strategy. Brainstorming was done as a team effort over the course of a few days. The final idea of using tries to store the contents of the dictionary for easy lookups was provided by Salomon.

Once the concept for the program was create, the programming of the program itself began. The subtasks of this section, along with by whom they were performed, are detailed below.

Tasks performed by Rob:

* Encryption algorithm (for later testing)
* Dictionary and ciphertext sorting helper functions
* Outline of trie class

Tasks performed by Salomon:

* Decryption strategy
* Details/implementation of trie class
* Brute-force algorithm

Constructing the recursive algorithm to actually decode the ciphertext was the largest task by far, but it was difficult for both team members to work on it at the same time, even with the use of a collaboration tool like GitHub. As a result, Salomon largely conducted in the implementation on his own, with some input from Rob, while Rob handled the creation of the report.

**Conclusion**

While a brute-force strategy to decrypting a random-key system such as this one is the easiest to implement, it is simply not efficient enough to be practical for real-life use. As a result, certain shortcuts need to be made in order to break the challenge into manageable tasks; sorting the dictionary by length and storing the words in tries served excellently for our purposes.