0.1 Rosetta Stone

This section involves determining the language of text context, and the official documentation provides comprehensive guidance. Just follow the documented steps for implementation.

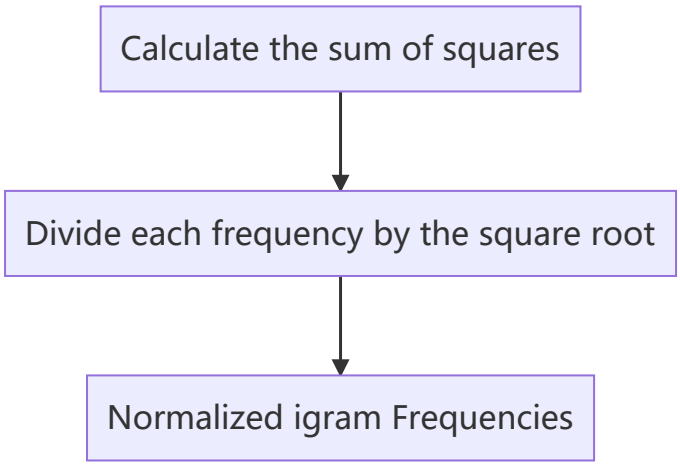
```
==== Tests for RosettaStone.cpp ====
   pass: Provided Test: kGramsIn works when the text length exactly matches the k-gram length.
   pass: Provided Test: kGramsIn works when the text length is one more than the k-gram length.
   pass: Provided Test: kGramsIn works when the text length is one more than the k-gram length.
   pass: Provided Test: kGramsIn works on a sample when k = 1.
   pass: Provided Test: kGramsIn works on a sample when k = 2.
   pass: Provided Test: kGramsIn handles non-English strings.
   pass: Provided Test: kGramsIn works when the input is shorter than the k-gram length.
   pass: Provided Test: kGramsIn reports errors on bad inputs.
   pass: Provided Test: normalize does not add or remove keys.
   pass: Provided Test: normalize works on positive numbers.
   pass: Provided Test: normalize works on negative numbers.
   pass: Provided Test: normalize works on unequal values.
   pass: Provided Test: normalize works on huge numbers.
   pass: Provided Test: normalize reports errors on inputs that are all zero.
   pass: Provided Test: topKGramsIn finds the most frequent k-gram.
   pass: Provided Test: topKGramsIn finds the top two k-grams.
   pass: Provided Test: topKGramsIn works if all k-grams are requested.
   pass: Provided Test: topKGramsIn works if more k-grams are requested than exist.
   pass: Provided Test: topKGramsIn works if no k-grams are requested.
   pass: Provided Test: topKGramsIn reports errors on bad inputs.
   pass: Provided Test: topKGramsIn works even if frequencies are negative.
   pass: Provided Test: topKGramsIn works with fractional weights.
   pass: Provided Test: cosineSimilarityOf works with one shared key.
   pass: Provided Test: cosineSimilarityOf works with two maps with non-overlapping keys.
   pass: Provided Test: cosineSimilarityOf works where keys are present in RHS but not LHS.
   pass: Provided Test: cosineSimilarityOf works where keys are present in LHS but not RHS.
   pass: Provided Test: cosineSimilarityOf works with keys missing from both sides.
   pass: Provided Test: guessLanguageOf works with perfect similarity.
   pass: Provided Test: guessLanguageOf works with imperfect similarity.
   pass: Provided Test: guessLanguageOf works on more complex inputs.
   pass: Provided Test: guessLanguageOf reports errors if no corpora are provided.
All tests in this section passed!
```

0.1.1 Form k-Grams

This function calculates the frequency of the k-gram in the text. This requirement primarily helps us become familiar with various collection types and their application scenarios on a personal level.

In this function, the critical operation is searching, enabling the Map to swiftly find the value associated with a specific key. To achieve this, it's essential to split the text based on the kGramLength.

0.1.2 Normalize Frequencies



To normalize gram frequencies across texts of different lengths, we can use the following mathematical model: Let f_i be the frequency of the gram T_i in a text, and N be the total number of distinct grams in the text.

First, calculate the sum of the squares of all gram frequencies:

$$S = \sum_{i=1}^{N} f_i^2$$

Then, normalize each gram frequency by dividing it by the square root of the sum of squares:

$$\text{Normalized Frequency of } T_i = \frac{f_i}{\sqrt{S}}$$

This normalization process ensures that gram frequencies are consistent across texts of different lengths.

```
double sum = 0;
for (string key : input) {
    sum += pow(input[key], 2);
}
sum = sqrt(sum);
for (string key : input) {
    result.put(key, input.get(key) / sum);
}
```

0.1.3 Filter Out Uncommon Trigrams

In this section, we need to filter out less frequently occurring grams and retrieve a specified number of the most frequent ones. The official documentation recommends using a priority queue for this implementation. In a priority queue, elements are sorted based on their weights, where smaller numbers represent higher priority by default.

```
PriorityQueue<string> queue;
for (string key : source) {
    queue.enqueue(key, INT_MAX - source.get(key));
}
```

However, for our case, larger numbers indicate higher importance in terms of frequency. This means that frequency is a measure of high significance. To handle this, when inserting elements into the priority queue and setting their weights, we need to transform this measure of high significance into a priority value. One common approach is to subtract the current frequency from a maximum value plus one. However, finding the maximum value would add additional time complexity. Therefore, we directly use the integer upper limit as the maximum value for this transformation.

0.1.4 Implement Cosine Similarity

This part is not very challenging; we can straightforwardly implement the formula for cosine similarity.

0.1.5 Guess a Text's language

This requirement, once all the aforementioned functions are completed, can be invoked logically.

0.1.6 Explore and Evaluate

After completing the implementation, click the "Run" button. The program will begin by loading a series of corpora. Once the loading process is finished, you can input the specific language you wish the program to predict into the text box.