```
(ns n-gram.letters.letter-probs (:require [n-gram.misc.misc-functions :refer :all]
                                          [n-gram.letters.file-reader :refer :all]
                                          [n-gram.words.file-reader :refer
[formattedText]]))

(def M-letter "Size of letter vocabulary" 256)

(def alpha1-letter "alpha for 1-gram letters" 1)

(def alpha2-letter "alpha for 2-gram letters" 1)

(def alpha3-letter "alpha for 3-gram letters" 1)

(defn p1-letter "1-gram probability for letters" [theLetter]
  (let [theLetter (clojure.string/lower-case theLetter)]
               (float (/ (+ (get-count counts-ASCII-1 theLetter) alpha1-letter)
                         (+ N-letter (* M-letter alpha1-letter)))))))

(def p1-letter-memo "Memoized p1-letter" (memoize p1-letter))

(defn p2-letter "2-gram probability for letters" [theLetters]
  (let [theLetters (clojure.string/lower-case theLetters)]
                (float (/ (/ (+ (get-count counts-ASCII-2 theLetters) alpha2-letter)
                             (+ (- N-letter 1) (* M-letter M-letter alpha2-letter)))
                             (p1-letter-memo (str (first theLetters)))))))

(def p2-letter-memo "Memoized p2-letter" (memoize p2-letter))

(defn p3-letter "3-gram probability for letters" [theLetters]
  (let [theLetters (clojure.string/lower-case theLetters)]
                (float (/ (/ (+ (get-count counts-ASCII-3 theLetters) alpha3-letter)
                             (+ (- N-letter 2) (* M-letter M-letter M-letter alpha3-
letter)))
                             (p2-letter-memo (str (first theLetters) (second
theLetters)))))))

(defn p-letter "Returns n-gram probability depending on number of input letters"
  [theLetters] (let [n (count theLetters)]
                (cond (= 1 n) (p1-letter-memo theLetters)
                      (= 2 n) (p2-letter-memo theLetters)
                      (= 3 n) (p2-letter-memo theLetters)
                      :else (println "Please enter a letter string of length 3 or
less"))))

(def p-letter-memo "Memoized p-letter" (memoize p-letter))

(defn log2 [n] (/ (Math/log n) (Math/log 2)))
```

```
(defn loop_sum_probs-2 [text] (let [prob (/ (counts-ASCII-2 (reduce str (take 2 text)))
(counts-ASCII-1 (str (reduce str (take 1 text)))))]
        (if (> (count text) 2) (+ (log2 prob) (loop_sum_probs-2 (subs text 1 (count
text))))
        (log2 prob))))

(defn loop_sum_probs-3 [text] (let [prob (/ (counts-ASCII-3 (reduce str (take 3 text)))
(counts-ASCII-2 (reduce str (take 2 text))))]
        (if (> (count text) 3) (+ (log2 prob) (loop_sum_probs-3 (subs text 1 (count
text))))
        (log2 prob))))

(defn loop_sum_probs-4 [text] (let [prob (/ (counts-ASCII-4 (reduce str (take 4 text)))
(counts-ASCII-3 (reduce str (take 3 text))))]
        (if (> (count text) 4)(+ (log2 prob) (loop_sum_probs-4 (subs text 1 (count
text))))
        (log2 prob))))

(defn ell [n] (let [probs (cond (= n 2) (loop_sum_probs-2 formattedText)
                                (= n 3) (loop_sum_probs-3 formattedText)
                                (= n 4) (loop_sum_probs-4 formattedText))] (- (* probs
(/ 1 (count formattedText))))))

(defn perplexity [n] (Math/pow 2 (ell n)))
```