```clojure
(ns n-gram.words.word-probs (:require [n-gram.misc.misc-functions :refer :all]
                                      [n-gram.words.file-reader :refer :all]
                                      [clojure.string :refer [lower-case]]
                                      [n-gram.words.good-turing :refer :all]))

(def M "Size of vocabulary" (count counts-1))

(defn p1 "1-gram probability" ([word1] (float (/ (+ (get-count-memo counts-1 [word1])
alpha1) (+ N (* M alpha1)))))
 ([word1 word-counts] (float (/ (+ (get-count-memo word-counts [word1]) alpha1)
                                                 (+ N (* M alpha1))))))

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

(defn p2 "2-gram probability" ([word1 word2] (float (/ (+
        (get-count-memo counts-2 [word1 word2]) alpha2)
                                  (* (+ (- N 1) (* M M alpha2)) (p1-memo word1))))))
(def p2-memo "memoized p2" (memoize p2))

(defn p3 "3-gram probability" ([word1 word2 word3]
  (float (/ (+ (get-count-memo counts-3 [word1 word2 word3]) alpha3)
             (* (+ (- N 2) (* M M M alpha3)) (p2-memo word1 word2))))))
(def p3-memo "memoized p3" (memoize p3))

(defn p4 "4-gram probability" ([word1 word2 word3 word4]
  (float (/  (+ (get-count-memo counts-4 [word1 word2 word3 word4]) alpha4)
            (* (+ (- N 4) (* M M M M alpha4)) (p3-memo word1 word2 word3))))))
(def p4-memo "memoized p4" (memoize p4))

(defn p "Returns n-gram probability depending on number of input words"
  ([word1] (let [word1 (lower-case word1)] (p1-memo word1)))
  ([word1 word2] (let [word1 (lower-case word1) word2 (lower-case word2)]
                  (p2-memo word1 word2)))
  ([word1 word2 word3] (let [word1 (lower-case word1)
                             word2 (lower-case word2)
                             word3 (lower-case word3)] (p3-memo word1 word2 word3)))
  ([word1 word2 word3 word4] (let [word1 (lower-case word1)
                                   word2 (lower-case word2)
                                   word3 (lower-case word3)
                                   word4 (lower-case word4)] (p4-memo word1 word2 word3
word4))))

(def p-memo (memoize p))

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

(defn loop_sum_probs-2 [text] (let [prob (p2-memo (first (first text)) (first (second
text))))] (if (> (count text) 2) (+ (log2 prob) (loop_sum_probs-2 (rest text))) (log2
prob))))

(defn loop_sum_probs-3 [text] (let [prob (p3-memo (first (first text)) (first (second
text)) (first (nth text 2)))] (if (> (count text) 3) (+ (log2 prob) (loop_sum_probs-3
(rest text))) (log2 prob))))

(defn loop_sum_probs-4 [text] (let [prob (p4-memo (first (first text)) (first (second
text)) (first (nth text 2)) (first (nth text 3)))] (if (> (count text) 4) (+ (log2 prob)
(loop_sum_probs-4 (rest text))) (log2 prob))))

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

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