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

(use 'clojure.pprint)

(defn dissoc-in
  "Dissociates an entry from a nested associative structure returning a new
     nested structure. keys is a sequence of keys. Any empty maps that result
     will not be present in the new structure."
  [m [k & ks :as keys]]
  (if ks
    (if-let [nextmap (get m k)]
      (let [newmap (dissoc-in nextmap ks)]
        (if (seq newmap)
          (assoc m k newmap)
          (dissoc m k)))
      m)
    (dissoc m k)))


(defn create-indices "Returns a vector containing the indices of the first and last
characters of the sub-word in the word"
  [word sub-word] (let [first-char-index (.indexOf word sub-word)] [first-char-index (+
first-char-index (count sub-word))]))

(def create-indices-memo "Memoized create-indices" (memoize create-indices))

(defn dereference-indices "Returns the sub-word from word given the first and last
character indicies"
  [word indices] (str (subs word (first indices) (second indices))))

(def dereference-indices-memo "Memoized dereference-indices" (memoize dereference-
indices))


(defrecord restaurant_node [range children restaurant depth]); restaurant with values
as a vector with each element representing the number of customers at a table, depth
with values [start end]

(defn build_restaurant_node "Builds a new restaurant node object
[depth] node with empty range, children and restaurant, but with given depth
[range depth] node with the given range and depth but empty children and restaurant
[range letter depth] node with the given range and depth, empty children and one table
with one customer for the given letter in the restaurant
[range children restaurant depth] node with the given range, children, restaurant and
depth"
```

```
  ([depth] (restaurant_node. [] (hash-map) (hash-map) depth))
  ([range depth] (restaurant_node. range (hash-map) (hash-map) depth))
  ([range letter depth] (restaurant_node. range (hash-map) {letter [1]} depth))
  ([range children restaurant depth](restaurant_node. range children restaurant depth)))

(def d0 0.62)

(def d1 0.69)

(def d2 0.74)

(def d3 0.80)

(def d4 0.95)

(defn return_discount [level] (cond (= 0 level) d0
                                    (= 1 level) d1
                                    (= 2 level) d2
                                    (= 3 level) d3
                                    :else d4))

(def return_discount_memo (memoize return_discount))

(defn find_discount [depth] (if (= (first depth) (second depth))
                              (return_discount_memo (first depth))
                              (* (return_discount_memo (first depth)) (find_discount
[(inc (first depth)) (second depth)]))))

(def find_discount_memo (memoize find_discount))

(defn p_new_table [number_of_tables discount number_of_customers
 (/ (* number_of_tables discount) number_of_customers))

(def p_new_table_memo (memoize p_new_table))

(defn p_existing_table [number_of_customers_at_table discount number_of_customers]
(/ (- number_of_customers_at_table discount) number_of_customers))

(def p_existing_table_memo (memoize p_existing_table))

(defn find_max_prob [tables] (let [maxs (map #(apply max %) tables)]
                               (apply max maxs)))

(def find_max_prob_memo (memoize find_max_prob))

(defn generate_prob_limits_loop [probs_map] (if (= 1 (count probs_map)) (zipmap [(first
(keys probs_map))] [(apply vector (cumsum (first (vals probs_map))))])
                                              (let [letters (keys probs_map)
```

```
                                                    resulting_map
(generate_prob_limits_loop (dissoc probs_map (last letters)))]
                                                   (assoc  resulting_map (last letters)
(apply vector (map #(+ (find_max_prob_memo (vals resulting_map)) %) (apply vector
(cumsum (get probs_map (last letters)) ))))))))

(def generate_prob_limits_loop_memo (memoize generate_prob_limits_loop))

(defn generate_probs_for_table_array [tables discount number_of_customers] (if (>
(count tables) 1) (into (generate_probs_for_table_array (rest tables) discount
number_of_customers) [(p_existing_table (first tables) discount number_of_customers)])

[(p_existing_table (first tables) discount number_of_customers)]))

(def generate_probs_for_table_array_memo (memoize generate_probs_for_table_array))

(defn generate_prob_limits
        ([tree params]
              (let [restaurant (get-in tree (into params [:restaurant]))
              letters (keys restaurant)
              discount (find_discount_memo (get-in tree (into params [:depth])))
              number_of_customers (reduce + (map #(reduce + (val %)) restaurant))
              number_of_tables (reduce + (map #(count (val %)) restaurant))
              probs_map (assoc  (zipmap letters (map
      #(generate_probs_for_table_array_memo (get restaurant %) discount
      number_of_customers) letters)) "" [(p_new_table_memo number_of_tables discount
      number_of_customers)])]
               (generate_prob_limits_loop_memo probs_map)))
        ([tree params letter]
              (let [restaurant (get-in tree (into params [:restaurant]))
              discount (find_discount_memo (get-in tree (into params [:depth])))
                number_of_customers (reduce + (map #(reduce + (if (= (key %) letter)
        (val %) [0])) restaurant))
                number_of_tables (reduce + (map #(count (if (= (key %) letter) (val %)
        [])) restaurant))
                probs_map (assoc  (zipmap [letter] [(generate_probs_for_table_array_memo
        (get restaurant letter) discount number_of_customers)] ) "" [(p_new_table_memo
        number_of_tables discount number_of_customers)])]
                 (generate_prob_limits_loop_memo probs_map))))

(def generate_prob_limits_memo (memoize generate_prob_limits))

(defn find_min_prob_and_index [prob_limits] (let [mins_map (zipmap (keys prob_limits)
(map #(apply min %) (vals prob_limits))) the_min (apply min-key val mins_map)]
                                    [the_min (.indexOf (get prob_limits (key the_min))
(val the_min))]))

(def find_min_prob_and_index_memo (memoize find_min_prob_and_index))
```

```
(defn vec-remove
  "remove elem in coll"
  [coll pos]
  (vec (concat (subvec coll 0 pos) (subvec coll (inc pos)))))

(defn prob_limits_and_index_loop [deciding_prob prob_limits]
        (let [min_limit (find_min_prob_and_index_memo prob_limits)]
                 (if (<= deciding_prob (val (first min_limit))) [(key (first
        min_limit)) (second min_limit)]
                (if (> (count (get prob_limits (key (first min_limit)))) 1)
                     (let [new_tables (vec-remove (get prob_limits (key (first
      min_limit))) (second min_limit))]
                     (prob_limits_and_index_loop deciding_prob (assoc prob_limits (key
      (first min_limit)) new_tables)))
                   (prob_limits_and_index_loop deciding_prob (dissoc prob_limits (key
             (first min_limit)))))))))

(def prob_limits_and_index_loop_memo (memoize prob_limits_and_index_loop))


(defn loop_find_tables "Cycles through all the children of the node to find the total
number of tables for the given letter"
  [children node letter params]
        (let [result (get-in node (into params [:children (first children) :restaurant
letter]))
              tables (if (nil? result) [] result)]
         (if (>= 1 (count children)) (count tables)
           (+ (count tables) (loop_find_tables (rest children) node letter params)))))

(def loop_find_tables_memo "Memoized loop_find_tables" (memoize loop_find_tables))

(defn find_child_table_count "Returns the total count of all tables for the given letter
in the node's children"
  [node letter params] (let [children (keys (get-in node (into params [:children])))]
(loop_find_tables_memo children node letter params)))


(def find_child_table_count_memo "Memoized find_child_table_count" (memoize
find_child_table_count))

(defn assign_customer [tree params letter]
  (prob_limits_and_index_loop_memo (rand)
                                   (generate_prob_limits_memo tree params letter)))

(defn check_table_customer_consistency "Checks that number of tables for the given
letter in the node's children is equal to the number of customers for the letter in the
node's restaurant"
  [node letter params]
```

```
  (let [table_count (find_child_table_count_memo node letter params)
        parent_restaurant (get-in node (into params [:restaurant letter]))
        parent_restaurant (if (nil? parent_restaurant) [] parent_restaurant)
        parent_customer_count (reduce + parent_restaurant)
        difference (- table_count parent_customer_count)]
              (cond (> difference 1)
                    (if (not (zero? table_count))
                      (if (not (zero? parent_customer_count))
                      (let [assignment (assign_customer node params letter)]
                        (if (= (first assignment) "")
                          (check_table_customer_consistency (assoc-in node (into
params [:restaurant letter]) (into parent_restaurant [1])) letter params)
                          (let [new_tables (apply vector  (map #(if (= (.indexOf
parent_restaurant %) (second assignment)) (inc %) %) parent_restaurant))]
                            (check_table_customer_consistency (assoc-in node (into
params [:restaurant letter]) new_tables) letter params))))
                      (check_table_customer_consistency (assoc-in node (into params
[:restaurant letter]) [1]) letter params)))
                    (= difference 1)
                    (if (not (zero? table_count))
                     (if (not (zero? parent_customer_count))
                      (let [assignment (assign_customer node params letter)]
                        (if (= (first assignment) "")
                          (assoc-in node (into params [:restaurant letter]) (into
parent_restaurant [1]))
                          (let [new_tables (apply vector (map #(if (= (.indexOf
parent_restaurant %) (second assignment)) (inc %) %) parent_restaurant))]
                            (assoc-in node (into params [:restaurant letter])
new_tables))))
                      (assoc-in node (into params [:restaurant letter]) [1])))
                    (< difference (- 1)) (let [new_tables (if (= (last
parent_restaurant) 1) (into [] (butlast parent_restaurant)) (apply vector (map #(if (=
(.indexOf parent_restaurant %) (.indexOf parent_restaurant (last parent_restaurant)))
(dec %) %) parent_restaurant)))]
                                           (check_table_customer_consistency (assoc-
in node (into params [:restaurant letter]) new_tables) letter params))
                    (= difference (- 1)) (let [new_tables (if (= (last
parent_restaurant) 1) (into [] (butlast parent_restaurant)) (apply vector (map #(if (=
(.indexOf parent_restaurant %) (.indexOf parent_restaurant (last parent_restaurant)))
(dec %) %) parent_restaurant)))]
                                          (assoc-in node (into params [:restaurant
letter]) new_tables))
                    (zero? difference)
                    node)))

 (defn check_for_children "Checks if there are any children starting with the given
letter"
  [letter children] (if (< 0 (count children))
```

```
                    (if (= (first children) letter) true (check_for_children letter
(rest children))) false))

(def check_for_children_memo "Memoize check_for_children" (memoize check_for_children))

(defn check_range "Compares the prefix to the range on the child node and returns a
vector with the values [(matching string) (is the match length equal to the range
length) (is the prefix longer than the match)]"
  ([range prefix root_word] (check_range range prefix root_word (count (dereference-
indices-memo root_word range))))
  ([range prefix root_word length] (let [range_letters (clojure.string/reverse
(dereference-indices-memo root_word range)) match (re-find (re-pattern (str "^"
range_letters)) prefix)]
         (if (nil? match) (check_range (create-indices-memo root_word
(clojure.string/reverse (subs range_letters 0 (dec (count range_letters))))) prefix
root_word length)
         [match (= length (count match)) (> (count prefix) (count match))])))

(def check_range_memo "Memoized check_range" (memoize check_range))

(defn check_consistency_all_tables "Checks that the number of tables in all children is
equal to the number of customers in the node's restaurant for all letters"
  [node tables params] (if (<= 1 (count tables))
                         (check_table_customer_consistency node (first tables) params)
                         (check_consistency_all_tables (check_table_customer_consistency
node (first tables) params) (rest tables) params)))


(defn branch "Replaces the node with another with range new_range and children a copy of
the old node but with the remaining range"
  [node letter new_range new_letter params]
        (let [old_range (get-in node (into params [:children letter :range]))
        restaurant (get-in node (into params [:children letter :restaurant]))
        children (get-in node (into params [:children letter :children]))
        old_depth (get-in node (into params [:children letter :depth]))
        new_range_length (- (last new_range) (first new_range))]
         (check_consistency_all_tables
               (assoc-in
                     (assoc-in
                           (dissoc-in node (into params [:children letter]))
                     (into params [:children letter]) (build_restaurant_node new_range
           [(first old_depth) (dec (+ (first old_depth) new_range_length))]))
                                                 (into params [:children letter
:children new_letter]) (build_restaurant_node [(first old_range) (- (last old_range)
new_range_length)] children restaurant [(+ (first old_depth) new_range_length) (last
old_depth)])) (keys restaurant) (into params [:children letter]))))
```

```
(defn build_tree "Builds a suffix tree for the given word"
  ([word]  (build_tree word (build_restaurant_node [0 0]) word [] 1))
  ([word root_node root_word params new_depth]
         (let [word_length (count word)
        root_node
              (if (empty? params)
                    (cond (> 2 word_length) root_node
                    :else (build_tree (subs word 0 (dec word_length)) root_node
            root_word params new_depth)) root_node)
        prefix (cond (< 1 (count word)) (clojure.string/reverse (subs word 0 (dec
word_length)))
              :else "")
      new_depth (if (= prefix "") (dec new_depth) new_depth)
      prefix_count (if (= prefix "") 1 (count prefix))
      prefix_start (str (first prefix))
      letter (str (last word))
      children (get-in root_node (into params [:children]))]
              (if (empty? children)
                    (check_table_customer_consistency (assoc-in root_node (into params
            [:children prefix_start]) (build_restaurant_node (create-indices-memo
            root_word (clojure.string/reverse prefix)) letter [new_depth (dec (+
            new_depth prefix_count))])) letter params)
             (if (check_for_children_memo prefix_start (keys children))
                (let [match_results (check_range_memo (get-in root_node (into params
   [:children prefix_start :range])) prefix root_word)]
                (if (second match_results)
                    (if (nth match_results 2)
                       (let [new_prefix (subs prefix (count (first
            match_results))(count prefix))]
                    (check_table_customer_consistency
                          (check_table_customer_consistency
                            (build_tree (str (clojure.string/reverse new_prefix)
                   letter) root_node root_word (into params [:children prefix_start])
letter (into params [:children prefix_start])) letter params))
 (check_table_customer_consistency root_node letter params))
         (let [new-range (create-indices-memo root_word (clojure.string/reverse (first
match_results))) matching_branch_range (get-in root_node (into params [:children
prefix_start :range]))]
               (check_table_customer_consistency
                    (check_table_customer_consistency
                         (assoc-in
                               (branch root_node prefix_start new-range (subs
                  (clojure.string/reverse (dereference-indices-memo root_word
                  matching_branch_range)) (count (first match_results)) (inc (count
                  (first match_results)))) params) (into params [:children prefix_start
                  :children (subs prefix (count (first match_results)) (inc (count
                  (first match_results))))]) (build_restaurant_node (create-indices-memo
                  root_word (clojure.string/reverse (subs prefix (count (first
```

```
                match_results)) (count prefix)))) letter [(+ new_depth (count (first
                match_results))) (dec (+ new_depth (count prefix)))]) letter (into
                params [:children prefix_start]) letter params))))

        (check_table_customer_consistency
              (assoc-in root_node (into params [:children prefix_start])
        (build_restaurant_node (create-indices-memo root_word (clojure.string/reverse
        prefix)) letter [new_depth (dec (+ new_depth (count prefix)))]) letter
        params))))))

(def uniform_letters_distribution_limits (generate_prob_limits_loop_memo (zipmap (map
#(str %) (map char (concat (range 32 33 )(range 48 58) (range 97 123)  ))) (repeat 37
[(float (/ 1 37))]))))

(defn determine_letter [tree params]
        (let [deciding_prob (rand)
        prob_limits (generate_prob_limits_memo tree params)
        letter (first (prob_limits_and_index_loop_memo deciding_prob prob_limits))]
              (if (= letter "")
                    (let [new_params (into [] (butlast (butlast params)))]
                          (if (empty? new_params) (first
              (prob_limits_and_index_loop_memo deciding_prob
              uniform_letters_distribution_limits))
                                          (determine_letter tree new_params)))
letter)))

(defn next_letter [tree context root_text]
        (let [children (keys (get tree :children))
        context (clojure.string/reverse context)]
         (if (check_for_children_memo (str (first context)) children) ;are there any
children starting with first letter of context
                                               (let [range_results (check_range_memo
(get-in tree [:children (str (first context)) :range]) context root_text)]
                                                    (if (second range_results)
                                                      (if (last range_results)
                                                        (next_letter (get-in tree
[:children (str (first context))]) (clojure.string/reverse (subs context (count (first
range_results)) (count context))) root_text)  ;context length> match length -
recursivley call with increased depth
                                                      (determine_letter tree [:children
(str (first context))])) ;match length=range length - search with child params
                                                      ;otherwise

                                                        (determine_letter (branch tree
(str (first context))

(create-indices-memo root_text (clojure.string/reverse (first range_results)))
```

```
(subs (clojure.string/reverse (dereference-indices-memo root_text (get-in tree
[:children (str (first context)) :range]))) (count (first range_results)) (inc (count
(first range_results))))

[]) [:children (str (first context))])));otherwise, instantiate node mid-way and search
parent distribution

                                                (determine_letter tree []))))

(defn find_prob [tree context root_text letter] (let [context (clojure.string/reverse
context)]

                                                    (let [range_results
(check_range_memo (get-in tree [:children (str (first context)) :range]) context
root_text)]
                                                      (if (second range_results)
                                                        (if (last range_results)
                                                          (find_prob (get-in tree
[:children (str (first context))]) (clojure.string/reverse (subs context (count (first
range_results)) (count context))) root_text letter)  ;context length> match length -
recursivley call with increased depth
                                                          (let [probs
(generate_probs_for_table_array_memo (get-in tree [:children (str (first context))
:restaurant letter]) (find_discount_memo (get-in tree [:children (str (first context))
:depth]))

(reduce + (map #(reduce + (val %)) (get-in tree [:children (str (first context))
:restaurant])))]
                                                            (/ (reduce + probs) (count
probs))));match length=range length - search with child params
                                                          ;otherwise

                                                            ))))

(def find_prob_memo (memoize find_prob))

(defn loop_predict [tree context root_text length] (if (> length 1)
                                                     (let [the_next_letter (next_letter
tree context root_text)]
                                                       (str the_next_letter
(loop_predict tree (str context the_next_letter) root_text (dec length))))
                                                     (next_letter tree context
root_text)))

(defn predict_text [tree context root_text length] (str context (loop_predict tree
context root_text length)))

(defn log2 [n]
```

```
  (/ (Math/log n) (Math/log 2)))

(defn loop_find_probs [tree tree-text root-text] (let [text-count (count tree-text)]
                                                   (if (> text-count 1) (+ (log2
(find_prob_memo tree (subs tree-text 0 (dec text-count)) root-text (str (last tree-
text)))) (loop_find_probs tree (subs tree-text 0 (dec text-count)) root-text))
                                                     (log2 (find_prob_memo tree (subs
tree-text 0 (dec text-count)) root-text (str (last tree-text)))))))

(defn ell [tree tree-text] (- (* (/ 1 (count tree-text)) (loop_find_probs tree tree-text
tree-text))))

(defn perplexity [tree tree-text] (Math/pow 2 (ell tree tree-text)))
```