```clojure
(ns autocomplete.configuration
  "Contains general settings.
Also contains common tooling functions."
  (:use clojure.test))

(declare vals-comparator)

(def autocomplete-default-settings
  "Default settings used for autocompletion.

- The first setting, :position, express whether any returned string
  should match from the beginning or from elsewhere. identity means
  everywhere; for a precise position (from 0), put a number.

- The second setting, :sort, if a function used for sorting the
  result. This function should implements Java comparator.

- The third setting, :limit, enforces the requirement to pop out up to 4
  results.

- The fourth setting, :laxity, specifies whether the letters from the
  sequence to be found must be contiguous. Possible values are :lax
  or :terse.

- The last setting, :case, indicate whether your autocompletion should
  be case-sensitive or not. Possible values are :relax or :strict."
  {:position identity
   :sort #(vals-comparator (count %2) (count %1))
   :limit 4
   :laxity :terse
   :case :relax})

(def list-of-words
  "A non exhaustive list of words. I've augmented the original one with
  some custom words. This is used for tests for the sake of convenience
  and readability thus should not be modified (or don't run tests
  afterwards)."
  '("Pandora"
    "Pierre de Boisset"
    "Hire me!"
    "I even accept æ–‡ Chinese ;-)"
    "i am a test"
    "Pinterest"
    "Paypal"
    "Pg&e"
    "Project free tv Priceline"
    "Press democrat"
    "Progressive"
    "Project runway"
    "Proactive"
    "Programming"
    "Progeria"
    "Progesterone"
    "Progenex"
    "Procurable"
    "Processor"
```

```
     "Proud"
     "Print"
     "Prank"
     "Bowl"
     "Owl"
     "River"
     "Phone"
     "Kayak"
     "Stamps"
     "Reprobe"))

(def long-list-of-words
  (with-open [rdr (clojure.java.io/reader "resources/words.md")]
    (doall (line-seq rdr))))

(def letter-frequency
  "Source: http://en.algoritmy.net/article/40379/Letter-frequency-English"
  { \a 8.167 \b 1.492 \c 2.782 \d 4.253 \e 12.702 \f 2.228 \g 2.015 \h 6.094
    \i 6.966 \j 0.153 \k 0.772 \l 4.025 \m 2.406 \n 6.749 \o 7.507 \p 1.929
    \q 0.095 \r 5.987 \s 6.327 \t 9.056 \u 2.758 \v 0.978 \w 2.361 \x 0.150
    \y 1.974 \z 0.074})

(defn vals-comparator
  "Implements Java comparator. Takes two values and says which one is
  the greatest."
  ([val1 val2]
   (vals-comparator val1 val2 [-1 0 1]))
  ([val1 val2 [lt eq gt]]
   (cond (< val1 val2) lt
         (= val1 val2) eq
         (> val1 val2) gt)))

(defn ifloor
  "floor casted to int"
  [x]
  (int (Math/floor (double x))))

(defn iceil
  "floor casted to int"
  [x]
  (int (Math/ceil (double x))))

(defn iround
  "floor casted to int"
  [x]
  (int (Math/round (double x))))

(defn str-print
  [& strings]
  (println (reduce #(str % " " %2)
                   strings)))

(defn word-sort
  "Returns -1 if the word is too short to provide a letter at the
  specified position. This -1 means that shorter words are
  lexicographically sorted before longer words, as in any dictionary."
  ([expected word]
```

```clojure
    (word-sort expected 0 word))
   ([expected position word]
    (if (< position (count word))
      (let [actual (nth word position)]
        (vals-comparator (int actual) (int expected)))
      -1)))

(with-test
  (defn case-match
    "Returns an function which test equality of two strings given a
  certain case sensitivity."
    [letter mode]
    (cond (= mode :strict)
          #(= (str letter) (str %))
          (= mode :relax)
          #(or (= (str %) (clojure.string/lower-case letter))
               (= (str %) (clojure.string/upper-case letter)))))
  (is (= true (apply (case-match "e" :strict) "e")))
  (is (= false (apply (case-match "e" :strict) "E")))
  (is (= false (apply (case-match "e" :strict) "F")))
  (is (= false (apply (case-match "e" :relax) "f")))
  (is (= true (apply (case-match "e" :relax) "e")))
  (is (= true (apply (case-match "e" :relax) "E"))))

(defn contains-all?
  [coll items]
  (loop [items items]
    (if (empty? items) true
        (if (some (partial = (first items)) coll)
          (recur (rest items))
          false))))

(defn contains-case-letters?
  ([coll letters]
   (contains-case-letters? coll
                           letters
                           (:case autocomplete-default-settings)))
  ([coll letters case-mode]
   (loop [letters letters]
     (if (empty? letters) true
         (if (some (case-match (first letters)
                               case-mode) coll)
           (recur (rest letters))
           false))))))
```