```
(ns autocomplete.core
  "This remote coding interview states:

You are given a list of keywords below. Write code that will offer up to
  4 suggested â€œautoÂcompleteâ€ based on the letters typed (not case
  sensitive). Similar to Google Autocomplete, except that results must
  be in order vs. Google ranked keywords.

These few lines of codes are a proposal. They features two ways of
  achieving autocompletion: one first constructs a graph (called
  substrate) then queries it. The second one is dichotomy and sorting to
  achieve faster and more scalable results. The remaining lines of this
  file will elaborate on the algorithms and data structures used by both
  ways."
  (:use [clojure.test]
        [autocomplete tooling
                      configuration
                      projectors]))

(comment
  "Here is a simple substrate built with two words: paypal and
  paywall. Paypay has identifer :id 0 and paywal is 1."
  {\p '({:id 0, :index 0, :next \a}
        {:id 0, :index 3, :previous \y, :next \a}
        {:id 1, :index 0, :next \a})
   \a '({:id 0, :index 1, :previous \p, :next \y}
        {:id 0, :index 4, :previous \p, :next \l}
        {:id 1, :index 1, :previous \p, :next \y}
        {:id 1, :index 4, :previous \w, :next \l})
   \y '({:id 0, :index 2, :previous \a, :next \p}
        {:id 1, :index 2, :previous \a, :next \w})
   \l '({:id 0, :index 5, :previous \a}
        {:id 1, :index 5, :previous \a, :next \l}
        {:id 1, :index 6, :previous \l})
   \w '({:id 1, :index 3, :previous \y, :next \a})})

(defn autocomplete-graph
  "The autocompletion function with a graph. This is not
  scalable. Moreover, you can't do that much with the tooling but I
  found that way nice to be coded because it deals with a graph and
  double-linked lists.

The basic idea is to construct a graph whose vertices are letters. The
  edges of this graph are made with double-linked lists standing for
  each words in the list. Elements of these double-linked lists are
  called tuples. Tuples are stored besides a node. Once this graph is
  built (with function contruct) it's called 'substrate'. The technical
  implementation has no redundancy. However, it is far from being
  minimal: I could have used Huffman tree for this.

The algorithm is thus pretty straightforward: for the first letter to be
  matched we just retrieve all tuples beneath the corresponding
  node (and they're filtered out to keep each word once or none). Then
```

```
  we iterate over following letters of the sequence to be matched and
  filter out non-conform tuples. The output is finally sorted and
  printed out."
  ([substrate letters]
   (autocomplete-graph substrate
                       autocomplete-default-settings
                       letters))
  ([substrate settings letters]
   (let [[sortf position case-mode limit]
         (map #(set-or-default %
                               settings
                               autocomplete-default-settings)
              [:sort :position :case :limit])]
     (->> [substrate position letters limit case-mode]
          (apply words-with-sequence)
          (map (partial full-word substrate))
          (sort sortf)))))

(defn autocomplete-dicho
  "The autocompletion function with dichotomy. This doesn't use a graph
  but a runs on a dichotomy-based interval search. It's highly more
  scalabe than the graph-based solution and the tooling is rather
  general, hence powerful.

Once more, the algorithm is not very far-fetched: first the word list is
  sorted (by dichotomy) according to some criterion. Then we find bounds
  of the sublist of all words which match the criterion (with a
  dichotomy-based threshold finding algorithm). We may iterate depending
  of what we want and once it's done the result is returned.

Whenever it's possible we try to take advantage of the tooling to avoid
  unnecesarry computations. For example, when we have to look for words
  containing some letters, we reduce data load by searching the rarest
  letters first.

One key concept throughout the code is `projector`. It's a function
  which returns a function. This letter one is used against a word to
  project it, that's to say to figure out it satisfy a criterion. For
  example, if the criterion is 'words of 5 letters' the list will sorted
  out such as shorter words are at the beginning then 5-letter words
  then longer words. Same tool is used inside the threshold-finding
  algorithm which is basically a dichotomy."
  ([words letters]
   (autocomplete-dicho words
                       autocomplete-default-settings
                       letters))
  ([words settings letters]
   (let [[sortf position laxity case-mode limit]
         (map #(set-or-default %
                               settings
                               autocomplete-default-settings)
              [:sort :position :laxity :case :limit])]
     (->> [words position letters laxity case-mode]
          (apply words-contain-sequence-at-dicho)
```

```
          (take limit)
          (sort sortf)))))

(let [substrate (construct list-of-words)]
  (println "\nLet's compare the two approaches")
  (print "graph: ")
  (time (autocomplete-graph substrate "Pro"))
  (print "dicho: ")
  (time (autocomplete-dicho list-of-words "Pro"))
  (str-print "I do perform better at LISt-Processing than at"
             "graph-processing :-).\n"))

(comment
  (time (autocomplete-dicho long-list-of-words "pro"))
  (time (autocomplete-dicho long-list-of-words
                            {:position identity
                             :limit 20
                             :laxity :terse
                             :sort (lexicographic)
                             :case :relax}
                            "pro"))
  (time (autocomplete-dicho long-list-of-words
                            {:position 0
                             :limit 20
                             :laxity :lax
                             :sort (lexicographic)
                             :case :relax}
                            "hire")))

(def most-anagrams
  "Said to be the word with most anagrams, let's find them."
  (->> "spare"
       (words-anagrams-of-dicho long-list-of-words)
       time))

(def longuest-english-word
  (->> long-list-of-words
       (sort #(vals-comparator (count %) (count %2)))
       last
       time))

(run-tests 'autocomplete.tooling
           'autocomplete.configuration
           'autocomplete.projectors)
```