GetAhead - Interview Practice 5

Word Hunting - Solution

Given a 4X4 grid of letters and a dictionary, find the longest word from the dictionary that can be formed in the grid.

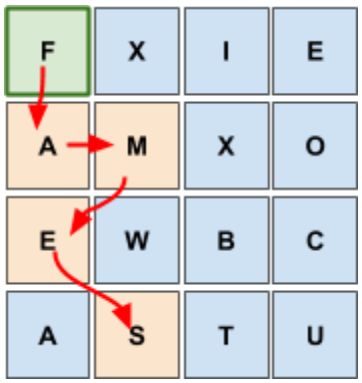
The rules for forming a word are:

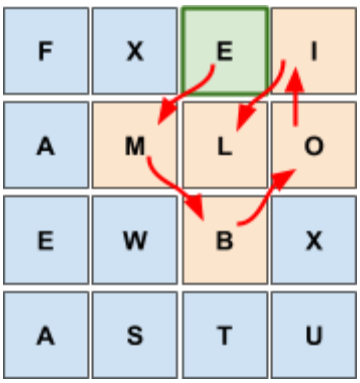
- Start at any position on the board
- Move in any of the up to 8 directions to choose another letter
- Repeat
- Words must be at least 3 characters long
- You cannot reuse a letter in a given position in the same word.

You are given a dictionary, with two helper functions: `isWord()` to check if a word is valid and `isPrefix()` to check if a string is a prefix of some valid word.

Examples:

For the grid on the left, the longest word is **FAMES**, for the one on the right, it is **EMBOIL** (note that there might be more than one solution, for example on the left **EMBOLI** is also valid, and same length)**:**





Solution

JAVA

```
import java.util.List;
import java.util.Queue;

// A class for location on the grid
class Location {
   int x;
   int y;

   Location(int x, int y) {
       this.x = x;
       this.y = y;
   }
}

// A class for each stage in our hunting
class Stage {
   private Location current_location;
   private String prefix;
   private List<Location> visited_locations;

   Stage(Location current_location, String prefix,
         List<Location> visited_locations) {
       this.current_location = current_location;
       this.prefix = prefix;
       this.visited_locations = visited_locations;
   }
   // Getters and Setters are nice but in an interview you can omit them
   // and make the variables public as long as you specify that you do
   // that only to save time and space.
   public Location get_current_location() {
       return this.current_location;
   }

   public String get_prefix() {
       return this.prefix;
   }

   public List<Location> get_visited_locations() {
```

```
        return this.visited_locations;
    }
}

public class WordHunter {
    private String[][] grid;
    private Queue<Stage> queue = new java.util.LinkedList<Stage>();
    private int max_len;
    private List<String> words = new java.util.ArrayList<String>();
    private String longest = "";
    private Location[] possible_moves = {new Location(1, 0), new Location(0, 1),
            new Location(-1, 0), new Location(0, -1),
            new Location(1, 1), new Location(1, -1),
            new Location(-1, 1), new Location(-1, -1)};

    WordHunter(String[][] grid) {
        this.grid = grid;
        this.max_len = grid.length;
        for (int i = 0; i < this.max_len; i++) {
            for (int j = 0; j < this.max_len; j++) {
                java.util.ArrayList<Location> visited_list =
                   new java.util.ArrayList<Location>();
                visited_list.add(new Location(i, j));
                queue.add(new Stage(new Location(i, j), grid[i][j], visited_list));
            }
        }
    }

    // These two functions are placeholders, given that the problems statement tells
    // us we are given a dictionary that implements them, we don't need to worry
    // about it, we can just call them in our solution.
    private boolean isWord(String word) {
        return true;
    }

    private boolean isPrefix(String word) {
        return true;
    }

    public List<String> getWords(){
        return this.words;
    }

    public boolean contains(final List<Location> list, Location location) {
        return list.stream().anyMatch(o -> o.x == location.x && o.y == location.y);
    }
```

```
public String findLongestWord() {
    while (!queue.isEmpty()) {
        Stage stage = queue.poll();
        // check all possible directions
        for (Location possible_move : this.possible_moves) {
            int new_x = stage.get_current_location().x + possible_move.x;
            int new_y = stage.get_current_location().y + possible_move.y;
            Location new_location = new Location(new_x, new_y);
            // check if the new location is inside the grid
            if (new_x >= 0 && new_x < max_len && new_y >= 0 &&
                new_y < max_len) {
                // check if the new location was not visited
                if (!this.contains(stage.get_visited_locations(),
                                   new_location)) {
                    String new_prefix = stage.get_prefix() +
                                        this.grid[new_x][new_y];
                    // check that the new prefix is a word
                    if (isWord(new_prefix)) {
                        words.add(new_prefix);
                        // check if longest word so far
                        if (new_prefix.length() > this.longest.length()) {
                            this.longest = new_prefix;
                        }
                    }
                    // check that the new prefix is a prefix
                    if (isPrefix(new_prefix)) {
                        // add to queue
                        java.util.ArrayList<Location> visited_list =
                            new java.util.ArrayList<Location>(
                                stage.get_visited_locations());
                        visited_list.add(new_location);
                        Stage new_stage = new Stage(
                            new_location, new_prefix, visited_list);
                        queue.add(new_stage);
                    }
                }
            }
        }
    }
    return longest;
}
```

You don't necessarily have to write test code in an interview, but you are still expected to provide meaningful test cases and try some manually.

```
    public static void main(String[] args) {
        String[][] grid1 = new String[][]{
                {"f", "x", "e", "i"},
                {"a", "m", "l", "o"},
                {"e", "w", "b", "x"},
                {"a", "s", "t", "u"}
        };
        String[][] grid2 = new String[][]{
                {"f", "x", "i", "e"},
                {"a", "m", "x", "o"},
                {"e", "w", "b", "c"},
                {"a", "s", "t", "u"}
        };
        WordHunter wordHunter1 = new WordHunter(grid1);
        String word = wordHunter1.findLongestWord();
        System.out.println("Longest word is " + word);
        System.out.println("The found words are " + wordHunter1.getWords());

        WordHunter wordHunter2 = new WordHunter(grid2);
        String word2 = wordHunter2.findLongestWord();
        System.out.println("Longest word is " + word2);
        System.out.println("The found words are " + wordHunter2.getWords());
    }
}
```

C++

```
// C++ Solution
#include <stdlib.h>

#include <algorithm>
#include <cstdio>
#include <deque>
#include <iostream>
#include <string>
#include <vector>

using Grid = std::vector<std::vector<char>>;

// This is a quick but non-comprehensive and non-efficient implementation
// of the dictionary. Since a dictionary has been provided to us, we don't worry
// about this bit. Note: An efficient implementation of a Dictionary here would
// involve a Trie. In an interview you don't have to code this class, the problem
// clearly states that you have it available already.
class Dictionary {
 public:
```

```
  Dictionary(std::vector<std::string> words) : words_(words) {}

  bool IsWord(const std::string& to_match) {
    return std::find(words_.begin(), words_.end(), to_match) != words_.end();
  }

  bool IsPrefix(const std::string& to_match) {
    for (const auto& word : words_) {
      if (word.find(to_match) == 0) return true;
    }
    return false;
  }

  bool empty() { return words_.empty(); }

 private:
  std::vector<std::string> words_;
};

class Location {
 public:
  Location(int x, int y) : x(x), y(y) {}

  int x;
  int y;
};

class Stage {
 public:
  Stage(Location current_location, std::string prefix,
        std::vector<Location> visited_locations)
      : current_location_(current_location),
        prefix_(prefix),
        visited_locations_(visited_locations) {}

  Location current_location() { return current_location_; }

  std::string prefix() { return prefix_; }

  std::vector<Location> visited_locations() { return visited_locations_; }

  bool Visited(const Location& to_match) {
    for (const auto& location : visited_locations_) {
      if (to_match.x == location.x && to_match.y == location.y) return true;
    }
    return false;
  }
```

```
 private:
  Location current_location_;
  std::string prefix_;
  std::vector<Location> visited_locations_;
};

class LongestWordFinder {
 public:
  LongestWordFinder(Dictionary dictionary, Grid grid)
      : dictionary_(dictionary), grid_(grid), max_length_(grid.size()) {
    assert(!grid.empty());
  }

  void Init() {
    stages_to_explore_.clear();
    for (int i = 0; i < max_length_; i++) {
      for (int j = 0; j < max_length_; j++) {
        stages_to_explore_.push_back(
            Stage(Location(i, j), std::string(1, grid_[i][j]),
                  std::vector<Location>({Location(i, j)})));
      }
    }
  }

  bool WithinGrid(Location location) {
    return location.x >= 0 && location.x < max_length_ && location.y >= 0 &&
           location.y < max_length_;
  }

  std::string FindLongestWord() {
    if (dictionary_.empty()) return std::string();

    Init();

    while (!stages_to_explore_.empty()) {
      // Pop an unexplored stage from the queue and make all possible moves from
      // there.
      Stage to_explore = stages_to_explore_.front();
      stages_to_explore_.pop_front();

      for (Location possible_move : possible_moves_) {
        int new_x = to_explore.current_location().x + possible_move.x;
        int new_y = to_explore.current_location().y + possible_move.y;
        Location new_location(new_x, new_y);
        if (!WithinGrid(new_location)) continue;
        if (to_explore.Visited(new_location)) continue;
```

```
        std::string new_prefix = to_explore.prefix() + grid_[new_x][new_y];
        if (dictionary_.IsWord(new_prefix)) {
          if (new_prefix.length() > longest_.length()) {
            longest_ = new_prefix;
          }
        }
        if (dictionary_.IsPrefix(new_prefix)) {
          to_explore.visited_locations().push_back(new_location);
          stages_to_explore_.push_back(
              Stage(new_location, new_prefix, to_explore.visited_locations()));
        }
      }
    }
    return longest_;
  }

 private:
  Dictionary dictionary_;
  Grid grid_;
  std::deque<Stage> stages_to_explore_;
  std::string longest_;
  int max_length_;
  std::vector<Location> possible_moves_ = {
      Location(-1, 0),  Location(1, 0),  Location(0, -1), Location(0, 1),
      Location(-1, -1), Location(1, -1), Location(-1, 1), Location(1, 1)};
};
```

You don't necessarily have to write test code in an interview, but you are still expected to provide meaningful test cases and try some manually.

```
int main() {
  Dictionary empty_dictionary({});
  Dictionary dictionary({"fame", "fames", "west", "emboil", "emboli", "lie",
                         "fax", "boil", "saw", "mew", "mews"});

  Grid grid1({{'f', 'x', 'e', 'i'},
              {'a', 'm', 'l', 'o'},
              {'e', 'w', 'b', 'x'},
              {'a', 's', 't', 'u'}});

  Grid grid2({{'f', 'x', 'i', 'e'},
              {'a', 'm', 'x', 'o'},
              {'e', 'w', 'b', 'c'},
              {'a', 's', 't', 'u'}});
```

```
  LongestWordFinder word_finder0(empty_dictionary, grid1);
  assert((word_finder0.FindLongestWord() == std::string()));


  LongestWordFinder word_finder1(dictionary, grid1);
  assert((word_finder1.FindLongestWord() == std::string("emboil")));

  LongestWordFinder word_finder2(dictionary, grid2);
  assert((word_finder2.FindLongestWord() == std::string("fames")));

  return 0;
}
```

Python

Questions to ask ourselves:

1. Can we compute this by brute force? (yes, it's a 4x4 grid).
    a. Follow-up: what's (approx) the maximum number of combinations?
2. What's the right level of abstraction? (classes Grid, Cell).
3. When do we stop recursing (if it's no longer the prefix of a valid word).

Trick: for the Grid, use a two-level dictionary mapping x value to y value to cell! I personally find this easier than working with a list of lists.

Python stuff which makes out life easier:

1. Module collections.
2. Module itertools.

Suggestions for live-coding this exercise:

- Start `find_words()` thinking of the first iteration of the recursion; then show how we have to modify to recurse down (e.g. by passing a copy of the visited elements).
- Emphasize: at a coding interview we don't want anythyng fancy. Keep it simple and "compose" out solution with building blocks that are intuitively simple, even if (acceptably) slower. E.g. first compute all possible words, then find the longest one with max().

```
import collections
import copy
import itertools
from typing import Iterator, List, Sequence, Set
import unittest

import util
```

```
Cell = collections.namedtuple("Cell", "x, y, letter")


class Grid:
  """A grid of letters."""

  def __init__(self):
    self._dictionary = util.Dictionary()
    self._cells = collections.defaultdict(dict)

  def __getitem__(self, x: int) -> Cell:
    return self._cells[x]

  def set(self, x: int, y: int, letter: str) -> None:
    self._cells[x][y] = Cell(x, y, letter.lower())

  def set_row(self, column_index: int, values: Sequence[str]) -> None:
    """Set the letters of an entire column in the grid."""
    for row_index, letter in enumerate(values):
      self.set(row_index, column_index, letter)

  def is_inside(self, x: int, y: int) -> bool:
    """Check whether (x, y) falls within the grid."""
    try:
      _ = self[x][y]
      return True
    except KeyError:
      return False

  def neighbors(self, x: int, y: int, visited: Set[Cell]) -> Iterator[Cell]:
    """Return all not-yet-visited contiguous cells."""

    for dx, dy in itertools.product([-1, 0, 1], repeat=2):
      # If both offsets are zero we remain at the same cell.
      if (dx, dy) == (0, 0):
        continue

      nx, ny = x + dx, y + dy
      if not self.is_inside(nx, ny):
        continue

      n = self[nx][ny]
      if n not in visited:
        yield n

  def find_words(self, x: int, y: int, prefix: str,
```

```
                seen: Set[Cell]) -> List[str]:
    """Form all possible words that complete `prefix` from cell (x, y)."""

    current = self[x][y]
    seen.add(current)

    # Note the copy of 'seen' that we pass down when recursing! This
    # is important: each different "parallel universe" is independent;
    # otherwise we keep the reference to the same object and modify it.

    all_words = []
    if self._dictionary.is_prefix(prefix):
      all_words.append(prefix)

    for neighbor in self.neighbors(x, y, seen):
      word = prefix + neighbor.letter
      if self._dictionary.is_prefix(word):
        all_words.extend(
            self.find_words(neighbor.x, neighbor.y, word, copy.copy(seen)))

    return all_words

  def find_longest_word(self, x: int, y: int) -> str:
    """Return the longest word starting in cell (x, y)."""

    start = self[x][y]
    words = self.find_words(x, y, start.letter, set())
    # Sort before finding the maximum so that if 2+ words have the same length,
    # we return the first one lexicographically.
    words.sort()
    return max(words, key=len)
```

You don't necessarily have to write test code in an interview, but you are still expected to provide meaningful test cases and try some manually.

```
class GridTests(unittest.TestCase):

  def test_find_logest_word_fames(self):
    """https://screenshot.googleplex.com/wGhd2GHoDMs."""

    b = Grid()
    b.set_row(3, "FXIE")
    b.set_row(2, "AMXO")
    b.set_row(1, "EWBC")
    b.set_row(0, "ASTU")
```

```
    self.assertEqual(b.find_longest_word(0, 3), "fames")

  def test_find_logest_word_emboil(self):
    """https://screenshot.googleplex.com/SiSQoK3mZbX."""

    b = Grid()
    b.set_row(3, "FXEI")
    b.set_row(2, "AMLO")
    b.set_row(1, "EWBX")
    b.set_row(0, "ASTU")
    self.assertEqual(b.find_longest_word(2, 3), "emboil")


if __name__ == "__main__":
  unittest.main()
```