Project 1: Word Search

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Change Log

With a common lab assignment we do not typically have time to change the lab in response to student questions, concerns, and common misunderstandings. However, as the project lasts three weeks, it is relatively common for small updated, extra hints, and minor typos to be added. This page will list any such modifications.

- Version 0.9 2022-10-07 Initial "preview" Version. This version has all the information you need to get started, and the function requirements are not expected to change, **but** we havn't worked out the *perfect* test-code yet, nor have we finalized an autograder – so we can't really call this a "1.0" release.

2 Essential Information

2.1 Pacing and Planning

This is an approximately 3 week long assignment. As such you should expect it to be longer, and more complicated, than past assignments you have undertaken. As this is likely your first programming project of at a larger-scale, we will be providing a detailed functional-design. This will provide a structure so that the 2 core behaviors of this project will *feel* like a series of smaller tasks. Do not underestimate the additional time needed to understand a provided design like this – this is more than a simple reading task – make sure you understand how the functions piece together, and the role of each part of this document. If you find yourself thinking "I don't need this" or "I don't know what this does, so I'm going to ignore it" there's a good chance you're setting yourself up for a painful lesson later when you realize why we designed the problem as we did.

Finally please recognize that **understanding a complicated task like this is hard**. This difficulty is deliberate – computer scientists are rarely asks to solve simple straightforward tasks. By providing a writeup like this we are also **helping build your technical-reading skill**. While we will happily help you understand the role of any given function in an office hours setting, also remember that you can ask us for advice about how *we* approach reading technical documents like this, and how we make sure we don't miss important details.

2.2 Learning Goals

A project like this often has many learning goals, both *direct* and *indirect*

- Practice building larger-scale software. Including the time-management skills needed to pace yourself, the organization skills in seeing how each piece of a large program fits into one big picture, and of course, the debugging skills needed for a large task like this.
- Experience with nested lists
- Experience with 2-dimensional data structures
- Practice computing the big-O runtime efficiency of code you've written.
- Practice with alternate looping patterns based on more complicated designs and dimensions
- Practice with larger-scale problem solving
- Practice reading and understanding a larger-scale problem description, and mapping larger problems to specific python ideas.
- (last but not least) Just more practical experience with python

2.3 Deadlines

This assignment will be due Friday October 28th at 6:00pm.

important notes about project-deadlines

- There is no late-work deadline for Project 1. (I.E. the 24 hour 10% rule from labs **does not apply to projects**) Without prior arrangement late work will not be accepted.
- There will be a small *Grace period* window put into gradescope to help mitigate technical difficulties during submission. You can submit during the grace period without deduction – if gradescope accepts your work, you will be fine.
- (Of course, we will honor standard umn policy [1] as it pertains to emergency situations)
- Like with labs, you can make multiple submissions, we will only grade the final submission made before the deadline. If you wish to get credit plan on submitting **SOMETHING**
- Every semester there are people who fail to finish project 1 not-because they are bad programmers, but because they do not dedicate enough time. Do not be one of these people. Plan to be finished several days before the deadline, and aim to make incremental progress. This is not a "do it in one sitting" project.

2.4 Individual Assignment

Unlike labs, where partner work is allowed, this project is an *individual assignment*. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in a grade of 0 on this assignment, even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an incident of academic dishonesty.

To be very clear, you can ask other students questions only about this document itself ("What is Daniel asking for on page 3?"). Questions such as "how would you approach function X" or "I'm stuck on part B can you give me a pointer" are considered inappropriate collaboration even if no specific code is exchanged. Coming up with general approaches to problems, and finding active ways to become unstuck are all parts of the programming process, and therefore part of the work of this assignment that we are asking you to do independently.

2.5 Appropriate Resources and libraries

You are expected to be careful to only use appropriate resources in this work. Word-search is a well-known and reasonably common problem. As such there are many ways to solve it, with some of these solutions documented online. Many of these solutions will be completely useless to you in your assignment. Nonetheless, you should avoid searching for any information about programming 2d grids in python, programming word-searches (in general) or other problem-specific ideas. The whole point of this lab is to exercise your own problem-solving skills, not to find someone else's solution and re-code it as your own.

[1] `https://policy.umn.edu/education/makeupwork`

You are, of course, free to use online resources to learn about *python itself* but you are forbidden from using outside resources to learn how to solve these problems. If you do use any resources that have not been discussed in class, you are expected to leave a citation about these resources using a python comment in your code. This will allow us to know what resources you've used, especially if we see something unexpected in your code. Failure to do so may be viewed as an academic integrity issue.

Additionally, please remember the "common libraries" rule from class: You can only use python libraries we've explicitly taught in class, such as math or random. Other libraries require you to explicitly seek permission. In particular use of the deepcopy library is explicitly forbidden, and will lead to a score of 0 on any related function. The "make a copy" problems in this code are deliberate test of your ability to understand what is a copy, and how to make one.

3 Introduction

Word searches are a type of puzzle in which a grid of letters is provided, along side a series of words. The task is then to find the given words inside the grid of letters. Depending on the word search itself, the words can be found in different orientations, left-to-right, top-to-bottom, or even at angles. While these are not terribly taxing puzzles compared to, for example, a crossword, or a sudoku, word searches remain a simple and entertaining type of puzzle for all ages.

From a programmer's perspective, word searches create two interesting, and closely related, challenges. First, given a 2d grid of characters and a specific string – determine whether or not the string is present, and if so report not only the location, but direction at which the word can be found. Second, given a target size, and a list of words, generate a pseudo-random 2d grid of characters such that all (or most) of the given words can be found in the generated grid. By the end of this project you should have a program capable of solving both of these problems.

4 Data Representation

A first step in any large program is making a deliberate decision about how real-world data will be represented in the program. For this program we have chosen four things that need specific, consistent decisions on data representation:

1. letter grids
2. locations
3. directions
4. solutions

4.1 Letter grids

Letter grids (or word grids – both terms will be used to mean the same thing) will be represented as a list, of lists, of strings (where each string represents a single letter). This representation might feel a little silly initially – after all, it has few apparent benefits over a list of strings, which is easier to type, and would be used the exact same way. However, there are some processes, especially generating puzzles, in which it is convenient to be able to modify individual letters – for that we need the full list-of-lists-of-letters design.

An example: The following word grid:

```
catos
mrdog
qtown
```

(words: cat, dog, art, town)

would be represented by the following python list-of-lists-of-letters

```
grid = [['c', 'a', 't', 'o', 's'],
        ['m', 'r', 'd', 'o', 'g'],
        ['q', 't', 'o', 'w', 'n']]
```

A few notes:

- The primary list (grid in the example above) contains lists of *rows* not lists of columns. This effects the correct way to order indices for row vs. column. So to get the 'd' in the second row, 3 from the left, we would use `grid[1][2]`
- Each letter is represented by it's own string. This makes it easy to change individual letters
- You are free to assume (without checking) that there is at least one row, at least one column, and that the list-of-lists is properly "rectangular" (each sub-list of the primary list has the same length). Your code will not be expected to deal with empty lists or situations where different sub-lists have different lengths.
- We will represent the grid as only lowercase alphabetical letters.
- The *width* of the grid will be the number of letters left-to-right. The above example has width 5
- The *height* of the grid will be the number of letters up-to-down. The above example has height 3.

4.2 Locations

A location in a 2d grid is commonly represented as a pair (x, y) where x represents left-to-right displacement, and y represented up-to-down displacement. We will follow the same pattern, with the top-left of the grid ('c' in the above example) being at location $(0, 0)$. For a few examples, the q in the above example is at location $(0, 2)$, and the 'g' is at location $(4, 1)$, and 'o' can be found at the three positions: $(3, 0), (3, 1), (2, 2)$.

4.3 Directions

Our program will allow four "legal" word directions:

- right (words positioned left-to-right)
- down (words positioned up-to-down)
- right-down (words going diagonal from top-left to bottom-right)
- right-up (words going diagonal from bottom-left to top-right)

While four more directions are logically possible, the other four directions would be simple reverses of these directions, and are often considered hard to find, confusing, and unfair by word-search fans, therefore we will not deal with them.

We will represent these directions in our computer as a pair (dx, dy) where dx describes how to change the x value of the location to go in a direction, and dy describes the change in the y values. The four legal directions can be defined as follows:

```
RIGHT = (1, 0)
DOWN = (0, 1)
RIGHT_DOWN = (1, 1)
RIGHT_UP = (1, -1)
DIRECTIONS = (RIGHT, DOWN, RIGHT_DOWN, RIGHT_UP)
```

For example, the tuple for right-down indicates that to go in the right-down direction you add 1 to both x and y. You should not program your code using if statements to say "if the direction equals right do this, if the direction equals down do this, etc." instead you should think about how you can use the values of these tuples to navigate the 2-d grid of letters. "solving" that puzzle (how to loop over 2d space given a location and direction) will be your primary hurdle for many of the required functions.

4.4 Solution

A "solution" to a word search is always defined relative to a specific letter grid, and a specific word. Within this context – a "solution" defines where the word is (a location) and in what direction. Therefore, we will represent a solution in python as a tuple with two elements a location (specifically, the location of the first letter of the word), and a direction (indicating which way to go to find the other letters of the word)

So, for example, consider the following word grid:

```
pcndthg
waxoaxf
otwgdrk
ljpibet
fvltown
```

word "cat" has solution `((1, 0), (0, 1))` meaning it can be found at location (1,0) following direction (0,1) I recommend looking for the words "wax", "paw", "lid", and "bet" and formulating a "solution" to those [2]

5 Core required functions

A template file has been provided on canvas. You will be required to rename this file `word_search.py` and fill in the following functions:

- `get_size(word_grid)`
- `print_word_grid(word_grid)`
- `copy_word_grid(word_grid)`
- `extract(grid, position, direction, max_len)`
- `find(word_grid, word)`

[2]It would be reasonable to check your "solutions" to those words against other students if you wanna check your understanding.

- `show_solution(word_grid, word)`
- `make_empty_grid(width, height)`
- `can_add_word(word_grid, word, position, direction)`
- `do_add_word(word_grid, word, position, direction)`
- `fill_blanks(word_grid)`

Make sure that the file you turn in is able to have functions imported using standard python imports. To do this, make sure any user interactions occurs with a `if __name__ == "__main__":` block.

We will break these up into three groups of functions: helper-functions, solver-functions, and generation-functions.

6 Helper functions

These functions represent basic tasks that are needed throughout the code, which make other functions easier, or which ultimately exist to help make sure you know how to work with the data representation used in this problem. Do these first, and test your answers very carefully (write your own tests in addition to those provided) before moving forward.

6.1 get_size

The get_size function should have a single parameter – a letter grid in the list-of-lists-of-letters format discussed earlier in this document. The return value of this function should be a tuple with two elements: the width, and height of the grid, in that order.

notes

- This should not modify the word grid passed to it.
- This should not be a long function, it can be done in 1 line, and should not need more than 4 lines of code in total.
- If you are having trouble with this function review the basic data representation and make sure you understand how many lists work together to represent a letter grid.
- make sure you're paying attention to the expected return order: width, height. Getting these "backwards" is an easy mistake to make.

6.2 print_word_grid

The print_word_grid function should print the word grid in a "dense" format suitable for an end-user. For example, the following function call:

```
print_word_grid([['a', 'b', 'c'], ['c', 'd', 'q']])
```

should cause the following text to be output to the user:

```
abc
cdq
```

Note, this function has no return value – it is expected to output directly to the user.

notes

- This should not modify the word grid passed to it.
- You may find it useful to review the advanced usage of python's print statement, such as changing the `sep` or `end` parameters. (This is talked about in zybooks 1.2)
- Make sure your output isn't "sideways" – check the above example to make sure you are thinking about the x and y locations in the list correctly relative to how it's stored.

6.3 copy_word_grid

The copy_word_grid function takes one parameter, a grid, and returns a copy of this grid. Notice, the copy must be made in such a way that it is fully independent from the original. For this function to be correct it must be possible to make a copy of a letter grid, modify the letter grid, and have the original remain unmodified.

An example may make things more clear here:

```
grid1 = [['a', 'b', 'c'], ['c', 'd', 'q']]
grid2 = copy_word_grid(grid1)
grid1[0][0] = 'z'
grid1[1][0] = 'z'
print_word_grid(grid1)
# expected:
#zbc
#zdq
print_word_grid(grid2)
# expected:
#abc
#cdq
```

notes

- This should not modify the word grid passed to it.
- This task is harder than it appears. Don't move on from it until you've tested that you're doing it right.
- If you're struggling with this, make sure you're copying *every list*. The above example has 3 lists, for example, so copying that grid would require 3 new lists to be made.
- The deepcopy library is explicitly not allowed here. If you use it here you will get a 0 on this function.
- Make sure your copy isn't "sideways"

6.4 extract

The extract function should have four parameters:

1. grid – a letter grid

2. position – a location (tuple of two integers x and y)

3. direction – a direction (tuple of two integers as documented earlier)

4. max_len – an integer

The purpose of the extract function is to extract a string from the grid of letters starting at the given position, moving in the given direction containing no more than max_len letters. If there are max_len letters available starting from the provided start location, going in the provided direction, then a string of length max_len should be returned. However, if the top, left, right, or bottom edge of the grid is reached before max_len is reached, a shorter string should be returned.

The behavior of this function is best understood through a series of examples:

Consider the following letter grid:

```
pcndthg
waxoaxf
otwgdrk
ljpibet
fvltown
```

Then:

- `extract(grid, (0,0), RIGHT, 4)` is "pcnd" (the first 4 letters going left-to-right starting at the top-left)
- `extract(grid, (0,0), DOWN, 5)` is "pwolf" (the first 5 letters going top-to-bottom start at the top-left)
- `extract(grid, (1,3), RIGHT_UP, 3)` is "jwo" (the first 3 letters starting from the 'j in (1,3) and going up and right)
- `extract(grid, (4,2), RIGHT, 5)` is "drk" (shorter than the requested 5 letters because we hit the left-edge of the grid)
- `extract(grid, (3,2), RIGHT_UP, 5)` is "gah" (shorter than the requested 5 letters because we hit the top-edge of the grid)
- `extract(grid, (5,2), DOWN, 5)` is "rew" (shorter than the requested 5 letters because we hit the bottom-edge of the grid)

Notes

- This should not modify the word grid passed to it.
- Make sure you check for hitting the top, right, and bottom side – each is possible, and missing any 1 check can lead to incorrect behavior in later functions.
- While the examples above look like `extract(grid, (1,1), RIGHT_DOWN, 2)` remember that grid and the directions here are just variables, so your function would see something more like: `extract([[...],[...],...], (1,1), (1,1), 2)`
- The hardest part of this function is deciding how to loop given a location and a direction. Remember that both location *and direction* are represented as a tuple with 2 integers. If you're struggling, try this:

  - Look at each of the examples above.
  - Do these examples yourself, but don't just write-out the letters returned, also write out the *series of locations that are used* Then find a relationship between the location-tuple, the direction-tuple, and the (x,y) positions visited
  - For example: `extract(grid, (0,0), RIGHT, 4)`

letter	position
p	(0,0)
c	(1,0)
n	(2,0)
d	(3,0)

- Remember that you can write helper functions to make this task easier, or to avoid code duplication between this and other functions.

7 solver functions

These functions are those that you would need to build a practical user interface for *automatically solving* word-searches. With these functions, a practical user-interface would be a relatively simple matter.

7.1 find

The find function should take a letter grid and a word. If the word can be found in the grid, then the location, and direction, at which the word can be found should be returned as the solution. If the word cannot be found in the given grid, then the special value None should be returned to indicate failure.

On it's own this function can be solved many different times, and many different ways. Depending on the exact approach you take, this may also be quite hard to program and debug. To make things easier, we are strongly recommended a "brute-force" solution to this problem.

Brute-force solutions are often less efficient, but are also (typically) easier to program, test, and validate In this case, the brute force algorithm will be simple: For each location in the grid, and each direction allowed, test if the word is contained starting at that location, going in that direction. This may sound like an inefficient solution (a 10 by 25 word-grid would involve testing 1000 possible "solutions" for each word). However, a computer can handle this much work without any noticeable delay. Most word_grids are not large by computer standards, so we don't need a more efficient solution.

notes

- This should not modify the word grid passed to it.
- Think about how other functions can be used to test a specific combination of a location and a direction. This ultimately shouldn't need to do the whole thing on it's own and is expected to be making use of other functions.
- The return value of this function is somewhat tricky so we'll recap it here: if the word can't be found return None (this is not a string, it needs no quotes. It's a special

python value.) If the word can be found it should be a tuple with a location and direction. In practice this might look like `((1,4), (1,-1))`

7.2 show_solution

The show function takes two parameters:

- the letter grid
- the word we are showing a solution for

If the word cannot be found in the word grid, a simple message saying the word could not be found should be printed. As an example (for the word "cat"):

```
cat is not found in this word search
```

Alternatively, if the word is in the word_grid, the word should be capitalized, and then the grid should be printed. (Note, this should not modify the grid provided, you may find it useful to make a copy of the grid in this function)

An example (again, for the word "cat")

```
CAT can be found as below
pCndthg
wAxoaxf
oTwgdrk
ljpibet
fvltown
```

Note that the word cat is capitalized, both in the printed word grid, and in the first line.

8 Generation functions

These functions work together (with the help of two additional provided functions) to generate novel word-search puzzles. While we would have liked to give you the task "write a novel word-search generation algorithm" this task ends up being a little too open-ended for the first-project. As such we've designed a general solution and provided you the core functions for this solution. Your task will be implementing 4 important helper functions (I.E. the functions that do the actual hard-work of making the new puzzle)

You should begin by reading the provided add_word and generate functions. Getting the "big picture" on how these functions work together will help you program the 4 required functions

8.1 make_empty_grid

The make empty grid function takes two integers: width and height, and return a new word grid. The word-grid should initially have the specified width and height. Initially, the word-grid should have '?' in all positions.

As an example (which I recommend testing)

```
grid = make_empty_grid(3,2)
print_word_grid(grid)
# expected
# ???
# ???

grid[0][0] = 'q'
grid[0][1] = 'k'
print_word_grid(grid)
# expected
# qk?
# ???
```

notes

- Like copy_word_grid this function has a few pitfalls that are easy to miss in testing. Make sure you test your function carefully to make sure each position in the generated grid can be changed independently.
- The '?' character represents an un-used space in the letter-grid. This will let us remember as we're generating a new word-search where we've already placed a word, and where we have not.

8.2 can_add_word

The can add word function should take 4 parameters:

- a word grid in the list of lists format described earlier in this document
- a single word (a string)
- a position (tuple of two numbers)
- a direction (tuple of two numbers)

This function should check if it is currently possible to add a given word to a given word grid, in a given place/direction. The function should return True if the placement is possible and False if not. The rules for this are simple:

- There must be enough space starting in the given position for the word
  - For example, if you want to place the word "ape" in a 4x3 grid, you cannot place it at position (2,1) going right. There would not be room for 3 more letters.
  - You could, however, put "ape" in a 4x3 grid starting at (2,0) if the word goes down.
- The word cannot *conflict* with existing word placements. In general, a word is said to conflict if-and-only-if putting the word in this location would mean changing a non-blank (not-'?') letter in the word-grid to some other letter. So you can place a word where it would change any '?' to another letter, or you could place the word in such a way that it overlaps with already-placed words (so long as no letter changes.)
- This is another situation that is best shown through examples first-and-foremost: Consider the following *partially filled* word grid:

```
d????
?orat
??ga?
??c??
```

We could place bat in several places without conflict. First, we could place the word so it only overlaps unused '?' spaces

```
d????
Borat
A?ga?
T?c??
```

We could place the word so it's 'a' overlaps with the 'a' from "rat" (as we wouldn't need to change the 'a' stored there.)

```
d?B??
?orAt
??gaT
??c??
```

But we can't place it in such a way that other letters have to change (in this case, the 'a' in (3,2) has to become a 'T' causing a conflict.)

```
d??B?
?orAt
??gT?
??c??
```

Hints

- Consider how other functions can drastically simplify this task.
- This is easy to get wrong so I'll say it again: the rule **is not** "you can't place a word where any non-blank letter is" the rules allow you to place a word only where it wouldn't need to **change** any non-blank letters.
- This function should not actually modify the word-grid, that's another functions job. This function just indicates if a given placement would be valid.

8.3 do_add_word

The do add word function should take 4 parameters:

- a word grid in the list of lists format described earlier in this document
- a single word (a string)
- a position (tuple of two numbers)
- a direction (tuple of two numbers)

This function should change the word grid to add the word at the indicated position (in the indicated direction). Your function does not have to worry about running out of space, or changing letters in an invalid way: this function will only be called with inputs that can_add_word indicates are valid and safe.

Hints

- This function will have a similar structure to extract, but instead of reading a word out of the grid, it's writing a word into the grid.

8.4 fill_blanks

This function takes a single parameter, a word grid. It should loop over this word grid and fill each position that currently has the blank letter '?' with a random lower-case alphabetical letter.

hints

- You should pick letters at random (not following any specific pattern or scheme)
- You should pick a new letter for each blank-space in the word grid (remember '?' represents a blank space)
- The `choice` method from python's `random` module might be useful here, and does work with strings.
- The autograder will be *very light* for this function since it has random (hard to test) behavior. Make sure you're running this a few times and looking our for anything that feels like a "patern".

8.5 Final testing

When you're done with all the above functions you should be able to use the generate function to create new word grids:

```
grid, words = generate(10, 10, ["java", "python", "list", "set", "tuple","string"])
print_word_grid(grid)

OUTPUT (will be random, here's an example)

uwchnlkdsf
nkttuplesd
uqhtbcjgal
ytstringli
pewclwljis
ybwlfslupt
teswjavaev
hzoojtcuap
ojineaxfvl
nwaslizzsz
```

You should test this on your own, both to validate that your code is correct, and because it's fun.

9 Additional Requirement

Finally, when done with all functions there is an additional requirement. At the top of the template PDF is a series of 4 questions. Each question asks for you to evaluate the worst-case big-O running time of one function you've written. See the comments in that template file for assumptions you should make. Input your answer to each question by filling in the provided strings. This is a bit of a hokey way to do this Q&A part of the lab, but it keeps all the answers in one file.

Each questions will be graded on the following rubric:

1. 0 points – no answer given, or a wrong answer is given with no explanation.
2. 1 points – A wrong answer with a (mostly) sound explanation is given.
3. 2 points – a correct answer is given

10 Grading

Grading information is not available in the 0.9 version of this document.

Testing information is not available in the 0.9 version of this document.