# 6.101 Final Exam

## Fall 2024

- You have **180 minutes** to complete this exam. There are **11 problems**.
- The exam is **closed-book** and closed-notes, but you are allowed to bring a single 8.5×11" double-sided page of notes, handwritten directly on the paper (not computer-printed or photocopied), readable without a magnifying glass, created by you.
    - If you bring a handwritten page of notes, your name should be on the page, and you should **hand in your notes page** to a staff member at the end of the exam.
- You may also use blank scratch paper.
- You may use **nothing else on your computer** or other devices: no 6.101 website; no Python or programming tools; no web search or discussion with other people.
- Before you begin: you must **check in** by having the course staff scan the QR code at the top of the page.
- This page **automatically saves your answers** as you work. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the exam.
- If you feel the need to **write a note to the grader**, you can click the gray pencil icon to the right of the answer.
- If you have a question, or need to use the restroom, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- To **leave early**: enter *done* at the very bottom of the page, show your screen with the check-out code to a staff member, and give the staff member your handwritten page of notes (if any).
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

(you can **open this preamble in a separate tab**)

A *word square* is an NxN square grid of letters consisting of English words reading both across and down. For example, here is a 2x2 word square:

```
I S
T O
```

which contains the words `IS` and `TO` reading across, and `IT` and `SO` reading down.

A word square may be incomplete, with some letters filled in and some cells still empty. For example, here is a 3x3 word square with several empty cells:

```
C _ T
_ G O
_ _ _
```

The goal of a word square puzzle is to fill in the empty cells with letters so that every row and column of the completed grid is an English word.

For the sake of this exam, assume that:

- all letters we are using will be **uppercase**;
- an NxN word square will have at least one row and column (i.e. N>0)

---

# Problem ⤭1: `rows` and `columns`

(you can **open this problem description in a separate tab**)

One possible representation of a word square is a tuple of rows represented as strings, like this:

```
>>> grid1 = ( 'IS', 'TO' )
```

which represents this word square:

```
I S
T O
```

Or this:

```
>>> grid2 = ('C_T', '_GO', '___')
```

which represents this word square containing empty cells:

```
C _ T
_ G O
_ _ _
```

Write two **generator functions** `rows()` and `columns()`, that produce the rows and columns, respectively, of this representation.

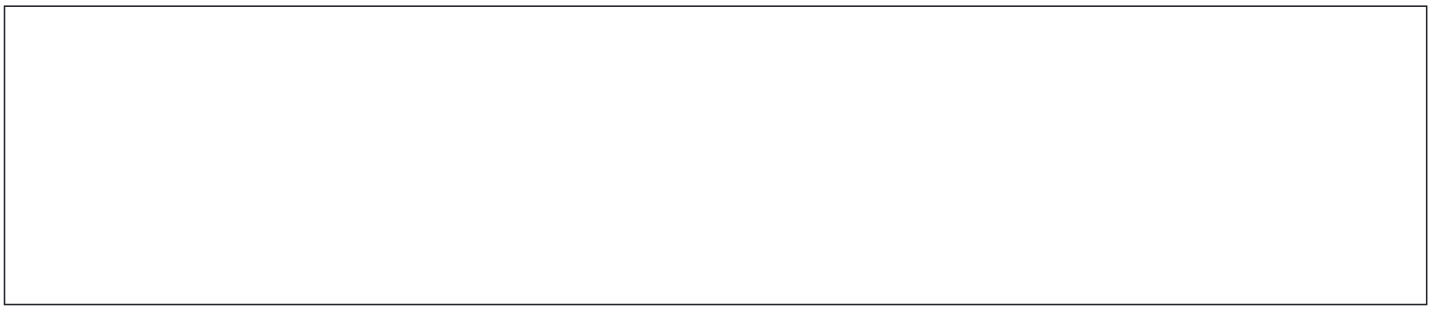
```
>>> for r in rows(grid1):
...   print(r)
IS
TO

>>> for c in columns(grid1):
...   print(c)
IT
SO
```

Your answers must be **generator functions**, using `yield`, not just returning a list or tuple or other iterable.

```
def rows(grid):
```

```
    yield from grid
```

```
def columns(grid):
```



```
    for i in range(len(grid)): # or grid[0]
        yield sum(row[i] for row in grid)
```

We can use `columns()` to implement a function that *transposes* a word square, so that the rows become columns, and vice versa. So the transpose of this word square:

```
C _ T
_ G O
_ _ _
```

becomes this word square:

```
C _ _
_ G _
T O _
```

```
def transpose(grid):
    return tuple(columns(grid))

>>> grid3 = ('C_T', '_GO', '___')
>>> transpose(grid2)
('C__', '_G_', 'TO_')
```

We will need some helper functions that can fill in a word square by *replacing* an entire row or entire column. Here are the helper functions, which rely on a generator function `replace_element()`:

```
def replace_row(grid, row_index, new_word):
    """
    returns a word square that is the same as `grid` except that
    the row at `row_index` is now `new_word`
    """
    return tuple(replace_element(rows(grid), row_index, new_word))

def replace_column(grid, column_index, new_word):
    """
    returns a word square that is the same as `grid` except that
    the column at `column_index` is now `new_word`
    """
    return transpose(replace_element(columns(grid), column_index, new_word))
```

Finish the job by implementing `replace_element()`, which must be a **generator function**.

```
def replace_element(iterable, i, new_element):
    """
    Generator function producing the same elements of `iterable`,
    except that the ith element (if any) is replaced by `new_element`.
    """
```

```
    for index, element in enumerate(iterable):
        yield new_element if index == i else element
```

Louis Reasoner suggests a much simpler way to replace rows:

```
def replace_row(grid, row_index, new_word):
    grid[row_index] = new_word
```

State two different problems with using Louis's `replace_row()` in place of the original version above. Each problem should fit in one box without scrolling.

`grid` is a tuple, so it cannot be mutated.

`replace_row()` must return the resulting word square, not mutate it.

# Problem ×2: `fits`

(you can **open this problem description in a separate tab**)

One possible step in solving a word square puzzle is testing whether a possible word *fits* into a row or column – whether it is compatible with the letters and empty cells in that row or column.

Write a **recursive function** `fits(s, t)` that tests whether `s` and `t` are compatible in this way. Examples are shown below:

```
>>> fits('_A_', 'AAA')
True
>>> fits('I_', '_S')
True
>>> fits('', '')
True

>>> fits('NO', 'ON')
False
>>> fits('CAT', 'CAT_')
False
```

Note that `fits(s, t)` should be symmetric in the sense that either `s` or `t` (or both) may contain underscores. The function should do the right thing for all strings `s` and `t`.

Note also that `fits(s, t)` does not consult any English dictionary in making its decision – it is *only* looking at the letters and underscores of `s` and `t`.

Your answer must be **recursive**, using no loops. Feel free to define helper functions inside `fits()` if desired.

```
def fits(s, t):
```

```
    if not s and not t:
        return True # base case, empty strings
    if not s or not t:
        return False # base case, different lengths
    if s[0] != t[0] and not (s[0] == '_' or t[0] == '_'):
        return False # s and t disagree on first character
    return fits(s[1:], t[1:])
```

# Problem ⨳3: `Words`

(you can **open this problem description in a separate tab**)

To make it easier to find words that might fit into the empty cells of a word square, let's create a `Words` class to store the possible English words.

An instance of the class is created from an iterable of words:

```
>>> words = Words(['A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'])
```

This is just a small example – normally there will be many thousands of words.

There should be an `all` attribute which stores the **set** of all the words:

```
>>> words.all
{'A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'}
```

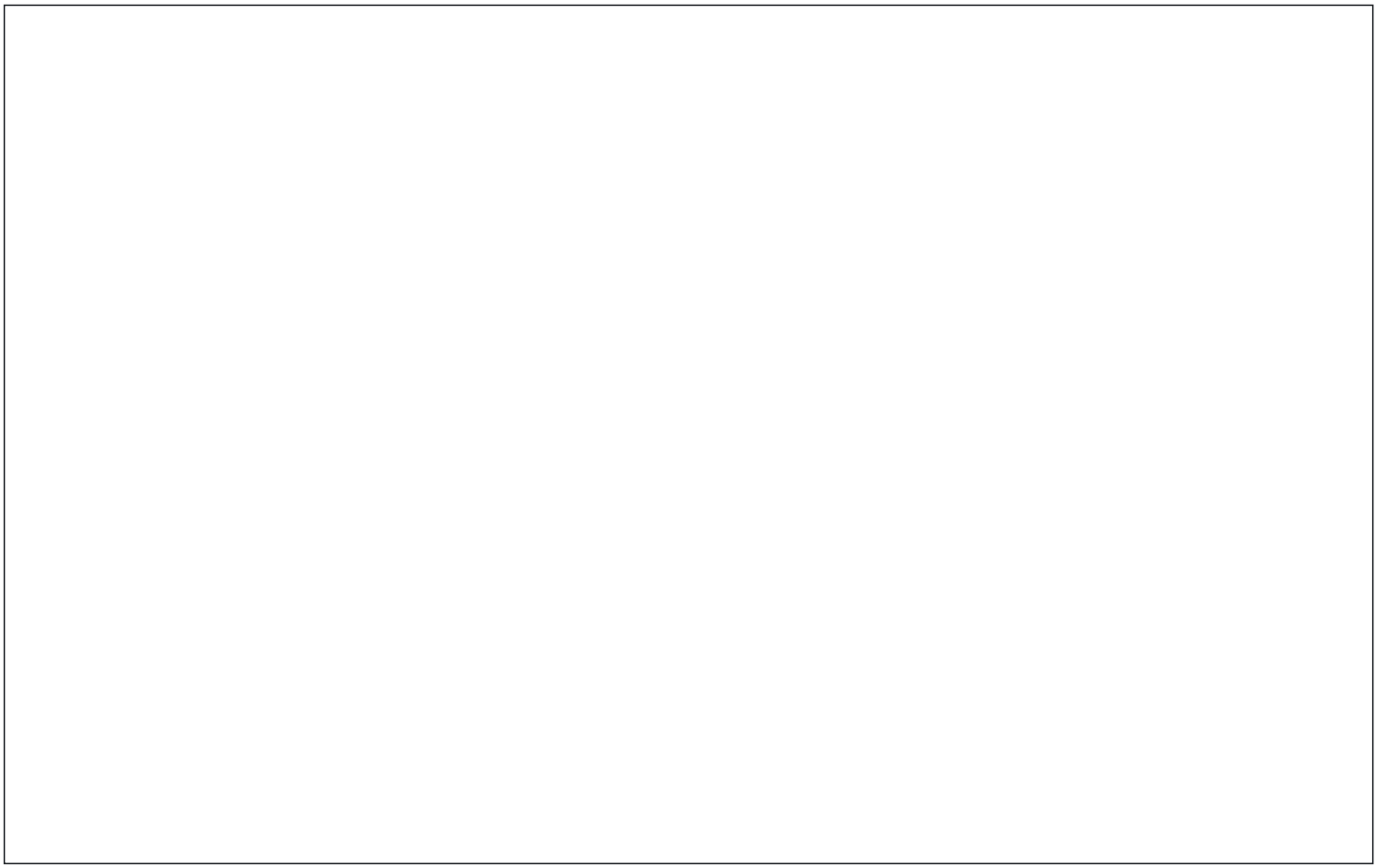
There should also be a method `lookup(letter, position)` that returns the **set** of all words with a given letter in a given position:

```
>>> words.lookup('A', 0)
{'A', 'APPLE'}

>>> words.lookup('A', 1)
{'CABLE', 'CAT', 'CATEGORY'}

>>> words.lookup('K', 20)
set()
```

The `lookup()` method should take **constant time** (independent of how many total words there are, or how many words the method has to return), because it may be used many times while solving a word square puzzle.

Write the `Words` class below.

**`class Words:`**



```
class Words:
    def __init__(self, word_list):
        self.all = set() # set of all words in word_list
        self.lookup_dict = {} # maps (index, letter) to a set of words
        for word in word_list:
            self.all.add(word)
            for index_and_letter in enumerate(word):
                if index_and_letter not in self.lookup_dict:
                    self.lookup_dict[index_and_letter] = set()
                self.lookup_dict[index_and_letter].add(word)
    def lookup(self, letter, index):
        return self.lookup_dict.get((index, letter), set())
```

# Problem ×4

For the remaining problems, assume that you have a working version of the `Words` class to build on (but written by somebody else, so you can only count on behavior stated in its problem description above).

When solving an NxN word square, we don't need *all* the words in the dictionary, only the words of length N. For example, a 2x2 word square only uses two-letter words, both across and down. A 3x3 word square only uses 3-letter words.

For this part, we will create a `Words` subclass, called `NLetterWords`, which only returns words of a given length `N`. The `lookup` method for `NLetterWords` should still take **constant time** (independent of how many total words there are, or how many words it has to return).

Louis Reasoner queries ChatGPT to help him write `NLetterWords`, and ends up doing it several times, because Alyssa P. Hacker finds problems in every version that he and ChatGPT try to create.

**Version A**

```
class NLetterWords(N):
    def lookup(self, letter, position):
        return {
            word for word in Words.lookup(self, letter, position)
                if len(word) == N
        }

# example of intended use  (doesn't necessarily work...)
>>> five_letter_words = NLetterWords(5, ['A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'])
>>> five_letter_words.all
{'APPLE', 'CABLE'}
>>> five_letter_words.lookup('A', 0)
{'APPLE'}
```

State two problems with version A. Each problem should fit into one box without scrolling.

`class NLetterWords(N)` should be `class NLetterWords(Words)`.

There is no `__init__` method with the right number of parameters.

`N` is not defined in `lookup()`.

`lookup()` will not run in constant time.

The `all` attribute will still have all words in it, not just N-letter words.

**Version B**

```
class NLetterWords(Words):
    N = None
    def __init__(self, word_list):
        Words.__init__([
```

```
            word for word in word_list if len(word) == self.N
        ])

# example of intended use  (doesn't necessarily work...)
>>> five_letter_words = NLetterWords(['A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'])
>>> five_letter_words.N = 5
>>> five_letter_words.all
{'APPLE', 'CABLE'}
>>> five_letter_words.lookup('A', 0)
{'APPLE'}
```

State two problems with version B (without repeating problems you noted about version A). Each problem should fit in one box without scrolling.

`self.N` will still be `None` when it is needed by `__init__()`.

The call to `Words.__init__(...)` needs to include the `self` parameter.

`N = None` creates a class attribute, not an instance attribute.

The `all` attribute will have no words in it, instead of N-letter words.

Neither `lookup()` nor `all` will be affected by reassigning `self.N`.

**Version C**

```
class NLetterWords(Words):
    def set_N(n):
        N = n
        self.all = {
            word for word in self.all
                if len(word) == N
        }
    def lookup(self, letter, position):
        return {
            word for word in Words.lookup(self, letter, position)
                if len(word) == N
        }

# example of intended use  (doesn't necessarily work...)
>>> five_letter_words = NLetterWords(['A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'])
>>> five_letter_words.set_N(5)
>>> five_letter_words.all
{'APPLE', 'CABLE'}
>>> five_letter_words.lookup('A', 0)
{'APPLE'}
```

State two problems with version C (without repeating problems you noted about versions A or B). Each problem should fit in one box without scrolling.

`N` is not defined in `lookup()`, because `N = n` creates it as a local variable in the frame of `set_N()`.

`def set_N(n)` is missing the `self` parameter.

Multiple calls to `set_N()` won't work (because the original word list is lost).

(if not mentioned in previous answer) `lookup()` will not run in constant time.

---

## Problem 5: `NLetterWords`

(you can **open this problem description in a separate tab**)

Alyssa asks you to write a working version of `NLetterWords`, with this example of its intended use:

```
>>> five_letter_words = NLetterWords(5, ['A', 'APPLE', 'CABLE', 'CAT', 'CATEGORY', 'ZIPPER'])
>>> five_letter_words.all
{'APPLE', 'CABLE'}
>>> five_letter_words.lookup('A', 0)
{'APPLE'}
```

Assume that you have a working version of the `Words` class to build on (but written by somebody else, so you can only count on behavior stated in its problem description above).

```
class NLetterWords(Words):
```

```
    def __init__(self, N, word_list):
        Words.__init__(self, word for word in word_list if len(word) == N)
```

---

# Problem ⤭6: `make_possibilities_function`

(you can **open this problem description in a separate tab**)

Now that we have `NLetterWords` and its method `lookup(letter, position)`, let's write code that takes a word-square row or column containing underscores and/or letters, and generate all possible words that would fit it.

Write a function that takes a word length `N` and an iterable of legal words, and returns a *possibilities* function. This possibilities function should take a string (containing 0 or more underscores) and return the set of legal N-letter words that fit it.

You may assume that legal words are always uppercase and never contain underscores.

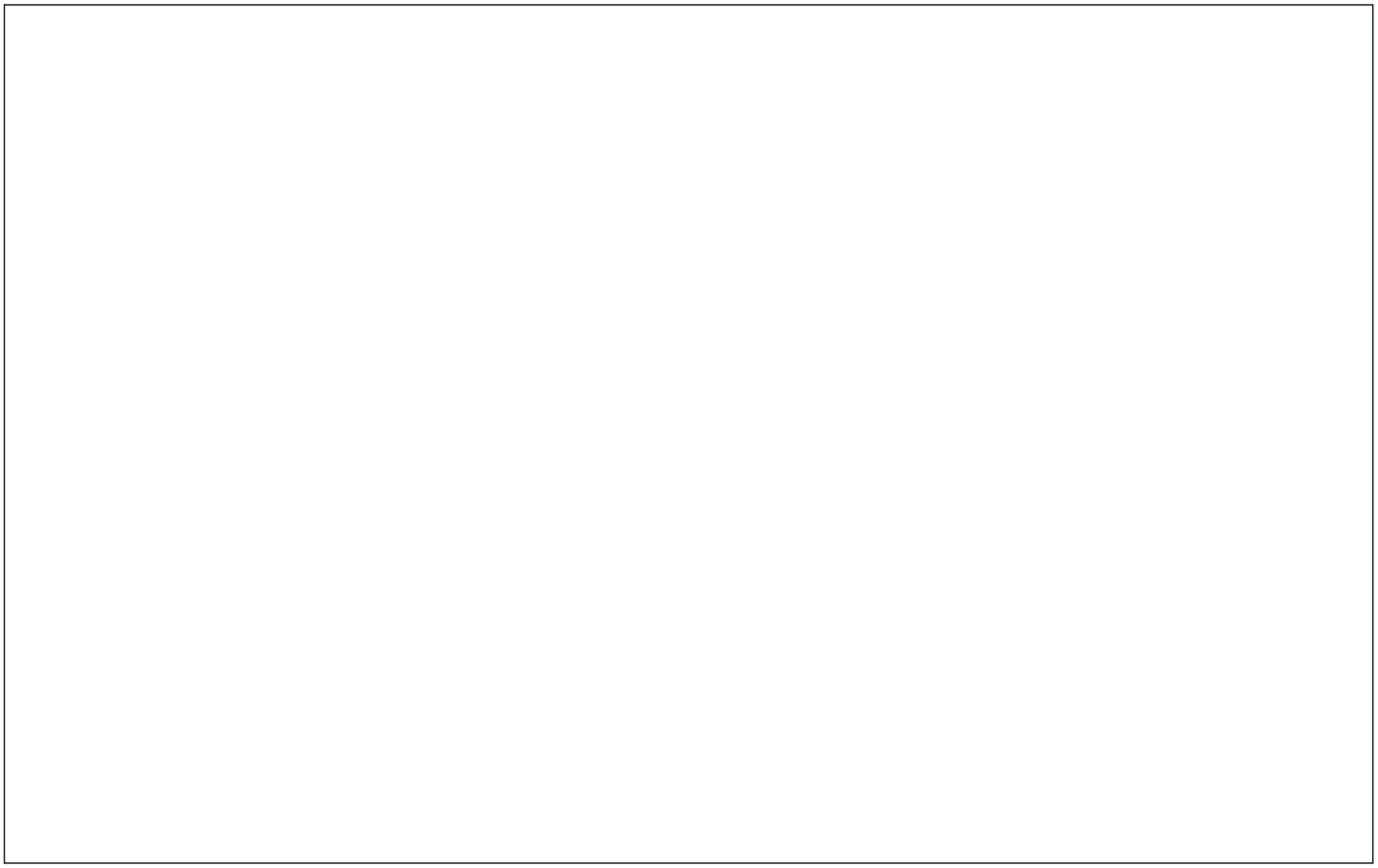
Example of use:

```
>>> five_letter_possibilities_for = make_possibilities_function(5, ['A', 'APPLE', 'CABLE', 'CAT', 'CAT
>>> five_letter_possibilities_for('_P_LE')
{ 'APPLE' }
>>> five_letter_possibilities_for('_____')
{ 'APPLE', 'CABLE' }
>>> five_letter_possibilities_for('C_T')
set() # because only returns 5-letter words
>>> three_letter_possibilities_for = make_possibilities_function(3, ['A', 'APPLE', 'CABLE', 'CAT', 'CA
>>> three_letter_possibilities_for('C_T')
{ 'CAT' }
```

Assume you have a working version of `NLetterWords` to use (but written by somebody else, so you can only count on behavior stated in its problem description above).

Your answer should create a `NLetterWords` only once, but be able to use it multiple times, and should use its `lookup(letter, position)` method wherever possible.

```
def make_possibilities_function(N, word_list):
```



```
    n_letter_words = NLetterWords(N, word_list)
    def possibilities(s):
        if len(s) != N:
            return set()
        possible_words = n_letter_words.all
        for i,letter in enumerate(s):
            if letter != '_':
                possible_words = possible_words & n_letter_words.lookup(letter, i)
        return possible_words
    return possibilities
```

---

# Problem ⤭7: `solvable`

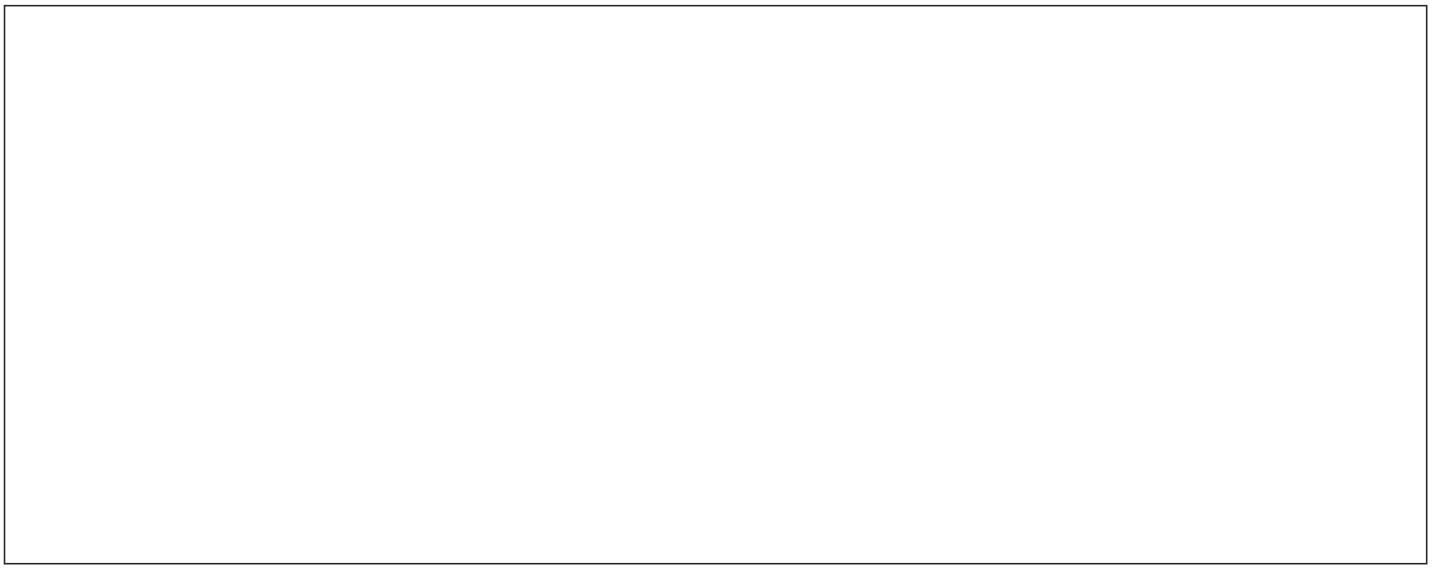
(you can **open this problem description in a separate tab**)

Write the helper function `solvable(grid, possibilities_function)`, which should return `True` if and only if there is at least one possible word for every row and column of the word square (considering each row and column *independently* of the others).

```
>>> two_letter_possibilities = make_possibilities_function(2, ['IS', 'IT', 'SE', 'SO', 'TO', 'WE'])
>>> solvable( ('IS', 'TO'), two_letter_possibilities)
True

>>> solvable( ('I_', '_O'), two_letter_possibilities)
True

>>> solvable( ('IS', '_E'), two_letter_possibilities)
True # because the second row _E could be WE, and the first column I_ could be IS or IT

>>> solvable( ('AS', 'NO'), two_letter_possibilities)
False  # 'AS' and 'NO' are not in the provided dictionary

>>> solvable( ('A_', 'N_'), two_letter_possibilities)
False
```

You can assume that you have working versions of helper functions and classes from previous problems (but written by somebody else, so you can only count on behavior stated in their problem descriptions).

```
def solvable(grid, possibilities_function):
```



```
    return (
            all( possibilities_function(row) for row in rows(grid) )
        and all( possibilities_function(column) for column in columns(grid) )
    )
```

## Problem ⨳8

Now write `solve()` below, which should fill in an incomplete word square using recursive backtracking search.

Your solution **must** use the following bits of code, which may need to be rearranged and integrated with other code in your solution, and which have places marked `TODO` that you need to fill in with a variable or expression:

- `for i,row in enumerate(rows(TODO))`
- `'_' in row`
- `possibilities_for = make_possibilities_function(TODO, word_list)`
- `for word in possibilities_for(row):`
- `solvable(TODO)`
- `replace_row(TODO, i, word)`

You may also use other helper functions or classes from previous problems if you wish, but this problem can be solved with just the helper functions shown in the bits of code above.

You can open the helper functions and classes from previous problems in **a separate tab**.

Your answer should create a possibilities function only once, but be able to use it multiple times.

```
def solve(grid, word_list):
    """
    grid is a word square represented as a tuple of strings, some of which may contain empty cells ('_
    word_list is an iterable of legal English words

    Returns a completed word square in tuple-of-strings format,
        with the same letters in the same places as the original grid,
        but with all underscores filled in,
        so that the grid contains only words from word_list, both across and down.
    Returns None if no solution can be found.
    """
```

```
def solve(grid, word_list):
    N = len(grid)
    possibilities_for = make_possibilities_function(N, word_list)
    def try_solving(g):
        if not solvable(g, possibilities_for):
            return None
        for i,row in enumerate(rows(g)):
            if '_' in row:
                for word in possibilities_for(row):
                    result = try_solving(replace_row(grid,  i, word))
                    if result:
                        return result
                return None
        return g
    return try_solving(grid)
```

## Problem ⨉9

Ben Bitdiddle observes that a completely-blank row is problematic for a backtracking search, because a statement like this:

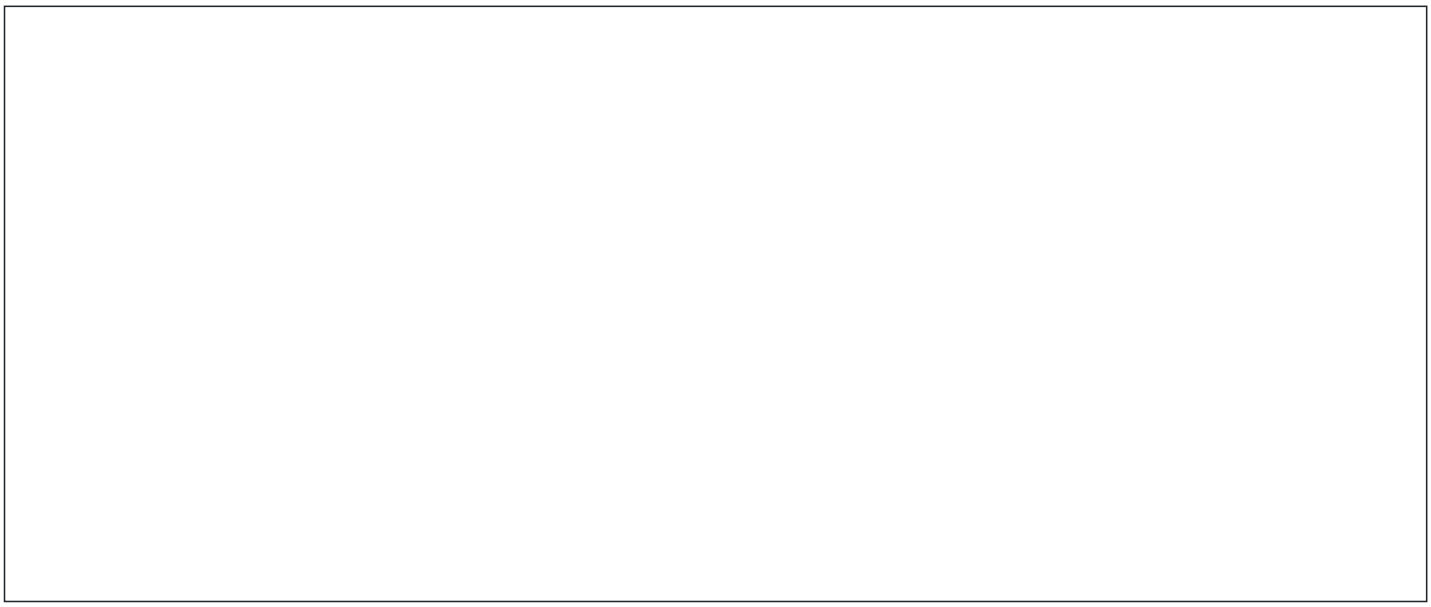
```
for word in possibilities_for(row):
    ...
```

...will try to replace the blank row with every possible N-letter English word.

He suggests being smarter about picking the next row to fill in, so that we have fewer choices to try.

Generalizing Ben's observation beyond just completely-blank rows, write a function that chooses the next row to try in the search:

```
def pick_best_row(grid, possibilities_for):
    """
    Return the index of a good row in the NxN grid to try filling in next.
    `possibilities_for` is a function that returns all the N-letter words that
    would fit a given length-N string.
    Assume `grid` has at least one empty cell.
    """
```



```
    best_row = None
    for i,row in enumerate(rows(grid)):
        if '_' in row:
            num_possibilities = len(possibilities_for(row))
            if best_row is None or num_possibilities < best_num_possibilities:
                best_row = i
                best_num_possibilities = num_possibilities
    return best_row
```

Ben realizes that his approach has a problem if every row is either completely blank or completely filled. In that case, maybe the best place for the search to start is by replacing a *column*. So he writes this function:

```
def pick_best_row_or_column(grid, possibilities_for):
    """
    Chooses the best row OR column to fill in next, considering all rows and columns.
```

```
    Returns a pair (dimension,index),
       where dimension is either 'row' or 'column'
       and index is the index of the row or column in `grid`.
    Assume `grid` has at least one empty cell.
    """
```

For example:

```
>>> dimension,index = pick_best_row_or_column( ('__', 'TO') )
>>> dimension
'column'
>>> index
0
```

Louis Reasoner observes that our `solve()` only knows how to try possibilities for rows, so he starts rewriting `solve()` so that it can work on columns as well, e.g.:

```
if dimension == 'row':
    ...
    for word in possibilities_for(row):
        ... replace_row(grid, index, word) ...
    ...
else:
    ...
    for word in possibilities_for(column):
        ... replace_column(grid, index, word) ...
    ...
```

Ben stops him before he gets very far. What is wrong with Louis's approach to making this change? It is enough just to name a general programming principle he is violating. If you need to say more, your answer should fit in the box without scrolling.

Don't Repeat Yourself

Ben suggests that if `dimension` is `"column"`, there is a simpler way to handle the situation, without rewriting any of the rest of `solve()`, by just transposing the grid and solving that instead. Fill in the code for the place marked TODO in Ben's solution:

```
dimension,index = pick_best_row_or_column( grid )
if dimension == 'column':
    TODO
else:
    # continue with trying replacements for the row at `index`
    ...
    for word in possibilities_for(row):
        ... replace_row(grid, index, word) ...
    ...
```

TODO should be:

```
    solution = solve(transpose(grid))
    return transpose(solution) if solution else None
```

# Problem ×10

Alyssa decides to experiment with a new representation for word squares. She is inspired by the way Scheme and Lisp represent linked lists by using a *pair* to store each element. But where linked lists are one-dimensional, a word square is two-dimensional, so Alyssa decides to use a *triple* for each cell of the word square.

Specifically, in Alyssa's "linked grid" representation, each cell of the NxN grid is represented by a three-element list:

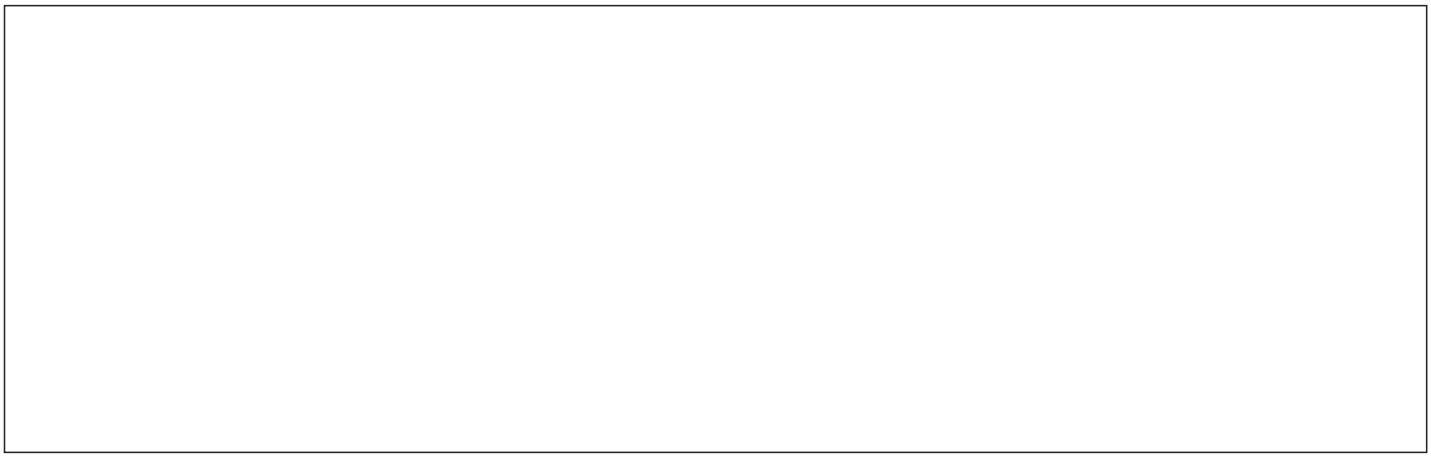
- The first element is the letter in the cell (or `'_'` if the cell is empty).
- The second element points down, to the neighboring cell below (or is `None` if this cell is in the bottom row of the grid).
- The third element points right, to the neighboring cell on the right (or is `None` if this cell is in the rightmost column of the grid).

Just as the starting point of a Scheme/Lisp linked list is the pair containing the first element, the starting point of a linked grid is the cell in the upper-left corner. By following right links and/or down links, every cell in the word square can be reached from that starting cell.

Write Python code to construct the following word square using this new linked-grid representation, and assign it to the variable `grid`:

```
I S
_ O
```

Your code may use any temporary variables you want, but the variable `grid` should end up assigned to the starting cell of the word square (its upper-left corner). Use multiple lines of code, and make your code as clear and simple as possible.



```
row0 = ['I', None, ['S', None, None]]
row1 = ['_', None, ['O', None, None]]
row0[1] = row1
row0[2][1] = row1[2]
grid = row0
```

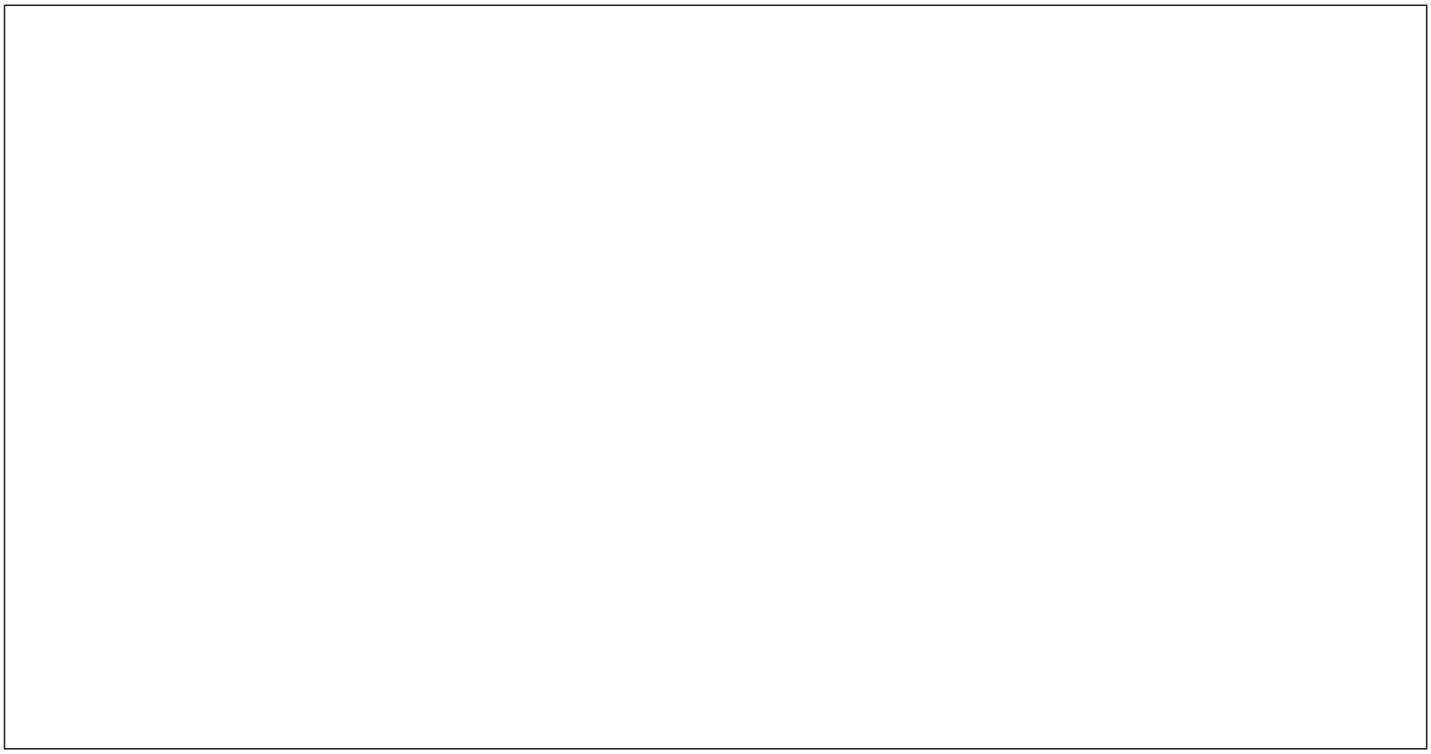
Next, Alyssa writes some helper functions for her representation:

```
def letter(cell):
    """
    returns cell's letter, or '_' if the cell is empty
    """
    return cell[0]

def down(cell):
```

```
    """
    returns the cell just below this cell, or None if at bottom of column
    """
    return cell[1]

def right(cell):
    """
    returns the cell just to the right of this cell, or None if at end of row
    """
    return cell[2]
```

Using Alyssa's helper functions, write `get_letter_at()` below. It must be a **recursive function**, using no loops.

```
def get_letter_at(grid, r, c):
    """
    Returns the letter or underscore in the cell at row `r` and column `c` of `grid`.
    Raises IndexError if `r` or `c` is out of bounds.

    For example, if grid is the linked-grid representation of `('IS','_O')`:
    >>> get_letter_at(grid, 0, 0)
    'I'
    >>> get_letter_at(grid, 1, 0)
    '_'
    """
```



```
    if not grid or r < 0 or c < 0:
        raise IndexError
    if r == 0 and c == 0:
        return letter(grid)
    if r > 0:
        return get_letter_at(down(grid), r-1, c)
    return get_letter_at(right(grid), r, c-1)
```

## Problem 11

Consider the code below:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

def apply(f, m):
    for i in range(m):
        n = m
        f(i)

apply(fact, 3)
```

After this code runs to completion, how many frames have been created that contain a binding for the name n?

7

Briefly show your work by saying which frames you counted. Your answer should fit in the box without scrolling.

`apply()` is called once, and its frame will contain `n`. It then calls `fact(0)`, `fact(1)`, and `fact(2)`, and the recursive computation of `fact(n)` creates `n+1` frames each containing the variable `n`, for a total of 1 + 2 + 3 = 6 frames. Adding the frame for `apply` gives 7.

---

After running the code below:

```
a = [1,2,3]
b = ['A'] + a + ['z']
c = { 'w': a }
d = a[0:3]
e = [a,a,a]
e.reverse()
f = [a[0], a[1], a[2]]
g = [[4],[a],[6]][1]
```

Write three different expressions that return an alias of `a`. None of the expressions should use the variable `a` itself, and each expression must use different variables: if one expression uses `b`, for example, the other two expressions may not.

`c['w']`

`e[0]` (or 1 or 2)

`g[0]`

Which of the variables above *cannot* be used to get an alias of `a`? Give a brief reason why, for each one. Your answer should fit in the box without scrolling.

`b` cannot, because concatenation of lists copies the elements.

`d` cannot, because a slice copies the elements.

`f` cannot, because it makes a fresh list containing the three elements of `a`.

---

After running the code below:

```
log = []
def monitor():
    x = 4
    def log_x():
        log.append(x)
    x = 5
    log_x
    x = 6
    return log_x
log_x = monitor()
x = 7
log_x()
x = 8
(lambda : log_x)()
```

What is the value of `log` after the code runs to completion?

`[6]`

(This is the end of the exam.)