6.101 Final Exam

Spring 2025

Name: **Answers**

Kerberos/Athena Username:

7 questions 3 hours

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- **This exam is closed-book**. You may not use any notes, cheat sheets, or electronic devices (including computers, calculators, phones, etc.).
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit, but if your work is on another page, please include a clear reference to the relevant page number in or near the answer box.
- **Do not write on the QR codes.**
- If you have a question, please **come to us to ask it**. You can also raise your hand and we will try to find you, but if we do not see your raised hand, please come to us to ask your question.
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still working.
- You must return **ALL** exam sheets to us, even extra sheets you remove from the end of the exam.
- You may not discuss details of the exam with anyone other than course staff until exam grades have been assigned and released.

Worksheet (intentionally blank)

1 Recursion vs Iteration

Your friend R. E. Cursion (the "R" stands for "R. E. Cursion"), inspired by the discussion in 6.101's readings about iteration versus recursion, has attempted to write purely-recursive versions of several iterative functions, and they have come to you for help with testing their work.

For each pair of functions on the following pages, indicate whether those two functions (one of which is iterative and one of which is recursive) will produce equivalent outputs for all valid inputs.

If the programs would produce different results, write a small program (using only valid, well-formed inputs to the function) that prints different values depending on which version of the function is used, as well as the specific output generated by that program in each case. If an input would cause an infinite loop, write **Infinite Loop** in the box. If an input would cause an infinite recursion, write **Infinite Recursion** in the box. If an input would cause an exception to be raised, try your best to approximate the error message that would be produced.

1.1 Program 1

`function_cascade` takes as input a list of 0 or more functions as input, where each function in that list takes a number as input and produces a number as output.

```
def function_cascade(functions):
    functions_copy = functions[:]
    def _inner(x):
        out = x
        for f in functions_copy:
            out = f(out)
        return out
    return _inner
```

```
def function_cascade(functions):
    if len(functions) == 0:
        return lambda x: x
    last, rest = functions[-1], functions[:-1]
    return lambda x: last(function_cascade(rest)(x))
```

Do these produce the same results for all valid inputs? (circle one): **Yes** / **No**

If not, a short example program that prints different results depending on which version is used:

Iterative Version Produces:

Recursive Version Produces:

1.2 Program 2

`unravel` takes as input a linked list in the format we've used in recitation, where `None` represents the empty linked list and all other linked lists are represented as tuples. For example, `(2, (3, (4, None)))` represents the linked list containing 2, 3, and 4, in that order.

```
def unravel(LL):
    out = []
    while LL is not None:
        out.append(LL[0])
        LL = LL[1]
    return out
```

```
def unravel(LL):
    out = []
    def helper(L):
        out.append(L[0])
        if L[1] is not None:
            helper(L[1])
    helper(LL)
    return out
```

Do these produce the same results for all valid inputs? (circle one): **Yes** / **No**

If not, a short example program that prints different results depending on which version is used:

```
print(unravel(None))
```

Iterative Version Produces:

```
[]
```

Recursive Version Produces:

```
TypeError:
'NoneType' object is not subscriptable
```

1.3 Program 3

`elts` takes as input a tree, represented as a list. The value at index 0 of this list represents the value of the tree's root note, and the values at index 1 onward (if they exist) are themselves trees (each represented in this same format). Here is one example of a valid tree in this format: `[13, [7], [8, [99], [16, [77]]]]`

```
def elts(tree):
    agenda = [tree]
    while agenda:
        item = agenda.pop(0)
        agenda.extend(item[1:])
        yield item[0]
```

```
def elts(tree):
    for t in tree[1:]:
        yield from elts(t)
    yield tree[0]
```

Do these produce the same results for all valid inputs? (circle one): **Yes** / **No**

If not, a short example program that prints different results depending on which version is used:

```
print(list(elts([13, [7]])))
```

Iterative Version Produces:

```
[13, 7]
```

Recursive Version Produces:

```
[7, 13]
```

1.4 Program 4

`clean_list` takes as input a list containing 0 or more integers.

```
def clean_list(L):
    while len(L) > 0 and L[0] == 0:
        L = L[1:]
    while len(L) > 0 and L[-1] == 0:
        L = L[:-1]
    return L
```

```
def clean_list(L):
    if not L:
        return L
    if L[0] == 0:
        L = L[1:]
    if L[-1] == 0:
        L = L[:-1]
    return clean_list(L)
```

Do these produce the same results for all valid inputs? (circle one): **Yes** / **No**

If not, a short example program that prints different results depending on which version is used:

```
print(clean_list([7]))
```

Iterative Version Produces:

```
[7]
```

Recursive Version Produces:

Infinite Recursion

2 Anagrams

Your friend Anna Graham is interested in a program that can determine whether two words are anagrams (that is, whether they contain exactly the same letters, but perhaps in a different order). Anna has asked some classmates for help and provided each of them with the following set of test cases:

```
def test_anagram_0():
    assert is_anagram('cat', 'act')

def test_anagram_1():
    assert is_anagram('trimmed', 'midterm')

def test_anagram_2():
    assert not is_anagram('tar', 'rats')

def test_anagram_3():
    assert not is_anagram('stops', 'pots')

def test_anagram_4():
    assert not is_anagram('dropped', 'propped')

def test_anagram_5():
    assert not is_anagram('top', 'pots')
```

For each of the implementations on the following page, indicate which test cases that implementation would **pass** by circling the corresponding numbers. For example, if a function passes only tests 0 and 3, you should circle those numbers but leave the rest uncircled.

Note also that some of these functions make use of string methods that we haven't seen very much of in 6.101:

- `s1.count(s2)` returns the number of non-overlapping occurrences of `s2` in `s1`.
  For example, `"ferret".count("r")` returns 2.
- `s1.index(s2)` returns the lowest index in `s1` where `s2` is found, or raises a `ValueError` exception if `s2` does not exist in `s1`. For example, `"ferret".index("e")` returns 1.

Code Sample A:

```
def is_anagram(w1, w2):
    counts1 = [w1.count(letter) for letter in w1]
    counts2 = [w2.count(letter) for letter in w2]
    return counts1 == counts2
```

Passing Tests:

(0) 1 (2) (3) (4) (5)

Code Sample B:

```
def is_anagram(w1, w2):
    for letter in w1:
        if letter not in w2:
            return False
    return True
```

Passing Tests:

(0) (1) 2 3 4 5

Code Sample C:

```
def is_anagram(w1, w2):
    if w1[0] not in w2:
        return False
    ix = w2.index(w1[0])
    return is_anagram(w1[1:], w2[:ix] + w2[ix+1:])
```

Passing Tests:

0 1 2 (3) (4) 5

3 Loopy Lists

Your pet goldfish loves listening to music, but you have a limited library of songs. Conveniently, though, by the time you reach the end of your playlist, your fish has already forgotten about the first song, so they will be equally excited to hear that first song all over again. In this problem, we'll implement a data structure that we can use to keep your fish happy: a `LoopyList`. A `LoopyList` behaves much like a regular Python list, except that its elements logically repeat themselves over and over again, forever.

Instances of `LoopyList` should be indexable with any integer number (even beyond the number of elements in your list), and the resulting value should be the element that *would* be at that index if the list repeated itself forever.

Also, looping over a `LoopyList` should result in the values from the list being repeated indefinitely as well.

```
>>> x = LoopyList(['good 4 u', 'Espresso'])
>>> for i in range(-3, 4):
...     print(i, x[i])
-3 Espresso
-2 good 4 u
-1 Espresso
0 good 4 u
1 Espresso
2 good 4 u
3 Espresso
>>> x[9]
'Espresso'
>>> x[999999999999999999999999999999999999999999999]
'Espresso'
>>> 'Espresso' in x
True
>>> x.append('Somebody That I Used to Know')
>>> for ix, elt in enumerate(x):
...     print(ix, elt)
...     if ix >= 7:
...         break  # this would keep looping forever if we didn't explicitly exit the loop
0 good 4 u
1 Espresso
2 Somebody That I Used to Know
3 good 4 u
4 Espresso
5 Somebody That I Used to Know
6 good 4 u
7 Espresso
```

Fill in the definition of the `LoopyList` class on the following page, inheriting from Python's built-in `list` class and only overriding those methods that you need to override to implement the behavior described and illustrated above. **You are not allowed to override `__init__`.**

For reference, a list of the dunder methods we've used/implemented in 6.101 labs is included as the last page in this exam (page 33), which you may remove if you wish.

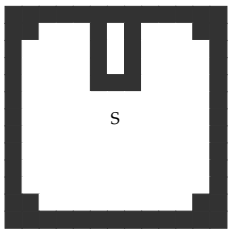
```
class LoopyList(list):
    # override whatever methods you need to below, but do not override __init__
    def __getitem__(self, ix):
        return list.__getitem__(self, ix % len(self))

    def __iter__(self):
        while True:
            yield from list.__iter__(self)
```

4 Flood Fill Variants

In this problem, we will consider an implementation of the "flood fill" program from week 3's readings, which recolors all the cells of a particular color in an enclosed region in an image. The code for a nearly complete implementation of flood fill is included below. Note that this code is correct and *almost* complete except for a missing segment of code near the bottom.

```
def flood_fill(image, location, new_color):
    original_color = get_pixel(image, *location)

    # r is the vertical component, increasing downwards
    # c is the horizontal component, increasing to the right
    # r=0, c=0 corresponds to the upper-left corner of the image
    def get_neighbors(cell):
        r, c = cell
        potential_neighbors = [(r-1, c), (r+1, c), (r, c-1), (r, c+1)] # up, down, left, right
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <= nc < get_width(image)
        ]

    to_color = [location]
    visited = {location}

    while to_color:
        this_cell = to_color.pop(0)

        set_pixel(image, *this_cell, new_color)

        neighbors = [
            neighbor
            for neighbor in get_neighbors(this_cell)
            if neighbor not in visited and get_pixel(image, *neighbor) == original_color
        ]

        ################
        # MISSING CODE #
        ################

        for neighbor in neighbors:
            visited.add(neighbor)

    return image
```

Here, we'll consider running the flood-fill process in the following image by clicking on the location labeled "S" and filling it with a grey color. The second-from-last page of this handout (page 31, which you may remove) contains several copies of the original image, which you may find helpful when thinking through the question below (though you do not need to use them if you don't want to).



Below are several snippets that could be used to fill in the missing part of the code on the facing page. For each, write a single letter in the box representing the resulting image **just before we remove the 26th item from the agenda**. The candidate images can be found on the third-from-last page of this handout (page 29), which you may remove. You may safely assume that the output for every code snippet is included on that sheet.

Code Segment 1

```
for neighbor in neighbors:
    to_color.append(neighbor)
```

Corresponding image: G

Code Segment 2

```
for neighbor in neighbors:
    to_color.insert(0, neighbor)
```

Corresponding image: K

Code Segment 3

```
to_color += neighbors
```

Corresponding image: G

Code Segment 4

```
to_color = neighbors + to_color
```

Corresponding image: I

Worksheet (intentionally blank)

5 Mouse Hunt

The mice you befriended in the Mice-sleeper lab in week 8 have been snoring away for the past several weeks, and now the time has come again to clean up the room. However, some of the mice may have moved around (or vacated the premises entirely), and some new mice may have moved in since then.

Recall that in our Mice-sleeper game, we kept track of a game board as a list of lists, where each location contained either a string `'m'` (indicating the presence of a mouse in that square) or a number representing the number of mice in adjacent squares (including diagonals). For example, the following is a valid game board:

```
complete_board = [
    ['m',  1,  0 ],
    [ 2,   2,  0 ],
    ['m',  2,  1 ],
    [ 1,   2, 'm'],
]
```

In this problem, we'll consider a tweak on this representation, where the game board is only partially known; the representation will be the same as before, but locations may now contain a string `'?'`, which means we are unsure whether that location contains a mouse or not, for example:

```
test_board = [
    ['?',  1,  '?'],
    ['?', '?', '?'],
    ['m',  2,  '?'],
    ['?', '?', '?'],
]
```

Each question mark either contains a mouse or an empty space, but only a small subset of those possible arrangements are consistent with the constraints that are visible on the board (for example, the constraint that location `(0,1)` must have exactly one mouse as a neighbor in the board above).

Your job for this problem will be to complete a function `make_consistent` that takes a board of this form (possibly including question marks) as an input and outputs a set of (row, column) locations where the unknown squares ("?") could be replaced by mice in a way that would be consistent with the known squares on the board.

```
>>> make_consistent(test_board)
{(3, 2), (0, 0)}  # <-- this example corresponds to complete_board shown above
```

This is not the only possible answer, though. For example, returning `{(1, 2)}` or `{(3, 1), (0, 0)}` would have been equally valid. But `{(3, 2)}` would not be a valid answer because that would mean that there were no mice adjacent to location `(0, 1)`, which must have 1 adjacent mouse because it contained a `1` in the input board.

We have provided several helper functions on the following page, which you are welcome to make use of in your solution.

Helper Functions:

```
def neighbors(board, row, col):
    """
    Given a board and a row and column, return the coordinates of all in-bounds neighbors.
    """
    return [
        (r, c)
        for r in range(row-1, row+2)
        for c in range(col-1, col+2)
        if 0 <= r < len(board) and 0 <= c < len(board[r])
    ]

def first_unknown_square(board):
    """
    Given a board, return the (row, col) location of the top-left-most '?' on the board.
    If the board does not contain any '?', return None instead.
    """
    for r, row in enumerate(board):
        for c, val in enumerate(row):
            if val == '?':
                return (r, c)
    return None

def satisfiable(board, row, col):
    """
    Given a board and a (row, col) location that contains a number, check whether the
    constraint imposed by that number can still be satisfied (False if it has been
    violated, True if it's either correct or can be made correct by replacing some number
    of neighboring question marks with mice)
    """
    n = board[row][col]
    assert isinstance(n, int)
    neighbor_vals = [board[r][c] for (r, c) in neighbors(board, row, col)]
    num_unknown = neighbor_vals.count('?')
    num_mice = neighbor_vals.count('m')
    return num_mice <= n <= num_unknown + num_mice

def check_neighbors(board, row, col):
    """
    Given a board and an arbitrary location on that board, check whether all constraints
    imposed by neighboring squares are valid (according to the definition of satisfiable)
    """
    return all(
        satisfiable(board, r, c)
        for r, c in neighbors(board, row, col)
        if isinstance(board[r][c], int)
    )
```

Complete the definition of `make_consistent` below. You may make use of the helper functions from the facing page, and you may assume that you are only given boards that can be completed successfully (i.e., you may assume that the initial board you are given does not contain any un-satisfiable constraints).

```
def make_consistent(board):
    location = first_unknown_square(board)
    if location is None:
        return set()
    else:
        r, c = location

        # try leaving this spot blank
        new_board[r][c] = 'x'
        if check_neighbors(new_board, r, c):
            # if leaving this blank was OK, recurse (if not, move on)
            result = make_consistent(new_board)
            if result is not None:
                return result


        # try placing a mouse here
        new_board[r][c] = 'm'
        if check_neighbors(new_board, r, c):
            # if placing this mouse was OK, recurse (if not, move on)
            result = make_consistent(new_board)
            if result is not None:
                return {(r, c)} | result

        # make sure to re-set this spot afterward since we mutated the board
        new_board[r][c] = '?'
        return None
```

6 Classes and Scoping

For each of the example programs below, what will be printed to the screen when it is run? None of the programs raise exceptions when run.

Program 1

```
n = 6

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        n = 4

bar = Bar()
n = 10
Bar.n = 12
print(bar.n)
```

Prints: 12

Program 2

```
n = 6

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

bar = Bar()
n = 10
Bar.n = 12
print(bar.n)
```

Prints: 6

Program 3

```
n = 6

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

class Baz(Bar):
    n = 8

baz = Baz()
n = 10
Bar.n = 12
print(baz.n)
```

Prints: 6

Program 4

```
n = 6

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

class Baz(Bar):
    n = 8

    def __init__(self):
        n = self.n + 1

baz = Baz()
n = 10
Bar.n = 12
print(baz.n)
```

Prints: 8

7 String Repeater

In this problem, we will consider a small language designed for specifying repeating patterns within text. In this language, an integer within a string specifies how many times the following part of the string should be repeated. These parts are either single characters or groups of characters surrounded by curly braces.

We would like to take inputs in this language and output the corresponding strings, using a function called `expand` that we'll implement in this problem. Consider the examples below:

- `expand('a')` → `'a'`
- `expand('1a')` → `'a'`
- `expand('5a')` → `'aaaaa'`
- `expand('cat')` → `'cat'`
- `expand('{cat}')` → `'cat'`
- `expand('3{cat}dog')` → `'catcatcatdog'`
- `expand('b2{an}a')` → `'banana'`
- `expand('b5{4{an}}a')` → `'bananananananananananananananananananananananananana'`
- `expand('3{2{2ac}d}4e')` → `'aacaacdaacaacdaacaacdeeee'`

The `expand` function can be written to make use of a helper function that takes both a string and an index as input and returns only a small piece of the output but also an index corresponding to the output of that part, such that the definition of `expand` looks like:

```
def expand(s):
    out, ix = '', 0
    while ix < len(s):
        c, ix = expand_helper(s, ix)
        out += c
    return out
```

Fill in the definition of `expand_helper` on the facing page so that this function will work as expected. Your code only needs to work for well-formed inputs, and you may assume that digits and curly braces are never part of the text that should be repeated (i.e., any occurrences of digits or curly braces only exist to specify a number of repetitions or a group of characters, rather than as part of a string to be repeated). You may also assume that the number of repetitions is always a `single-digit` number.

```
def expand_helper(s, ix):
    char = s[ix]
    if char.isdigit():
```

```
num_repetitions = int(s[ix])
pattern, nextix = expand_helper(s, ix+1)
return pattern * num_repetitions, nextix
```

```
    elif char == '{':
```

```
inner = ''
ix += 1  # don't include the '{'
while s[ix] != '}':
    subexpr, ix = expand_helper(s, ix)
    inner += subexpr
return inner, ix + 1
```

```
    else:
```

```
return char, ix+1
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

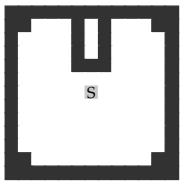
Worksheet (intentionally blank)

Worksheet (intentionally blank)

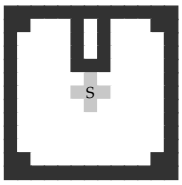
Worksheet (intentionally blank)

Worksheet (intentionally blank)

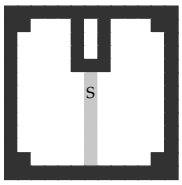
Options for Flood-Fill Question



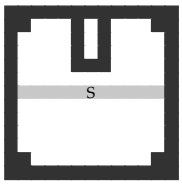
A
(1 grey pixel)



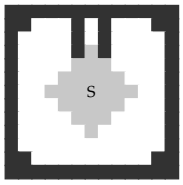
B
(5 grey pixels)



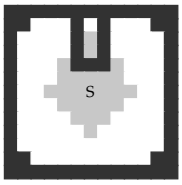
C
(7 grey pixels)



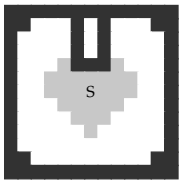
D
(11 grey pixels)



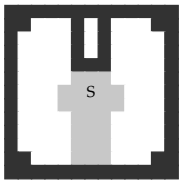
E
(25 grey pixels)



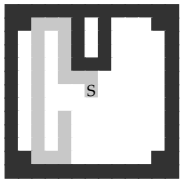
F
(25 grey pixels)



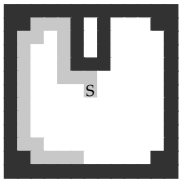
G
(25 grey pixels)



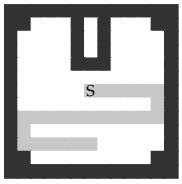
H
(25 grey pixels)



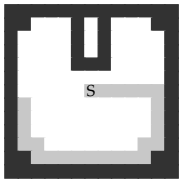
I
(25 grey pixels)



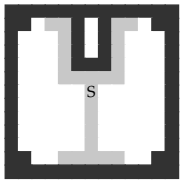
J
(25 grey pixels)



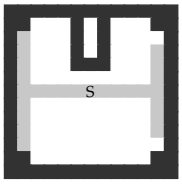
K
(25 grey pixels)



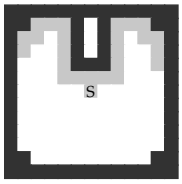
L
(25 grey pixels)



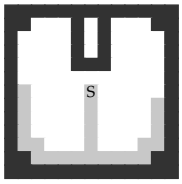
M
(25 grey pixels)



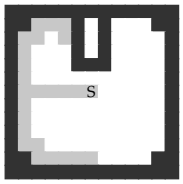
N
(25 grey pixels)



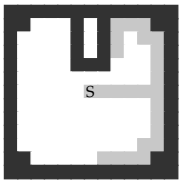
O
(25 grey pixels)



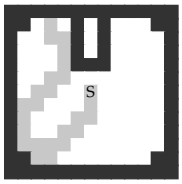
P
(25 grey pixels)



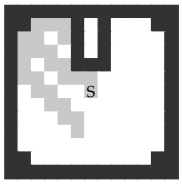
Q
(25 grey pixels)



R
(25 grey pixels)



S
(25 grey pixels)

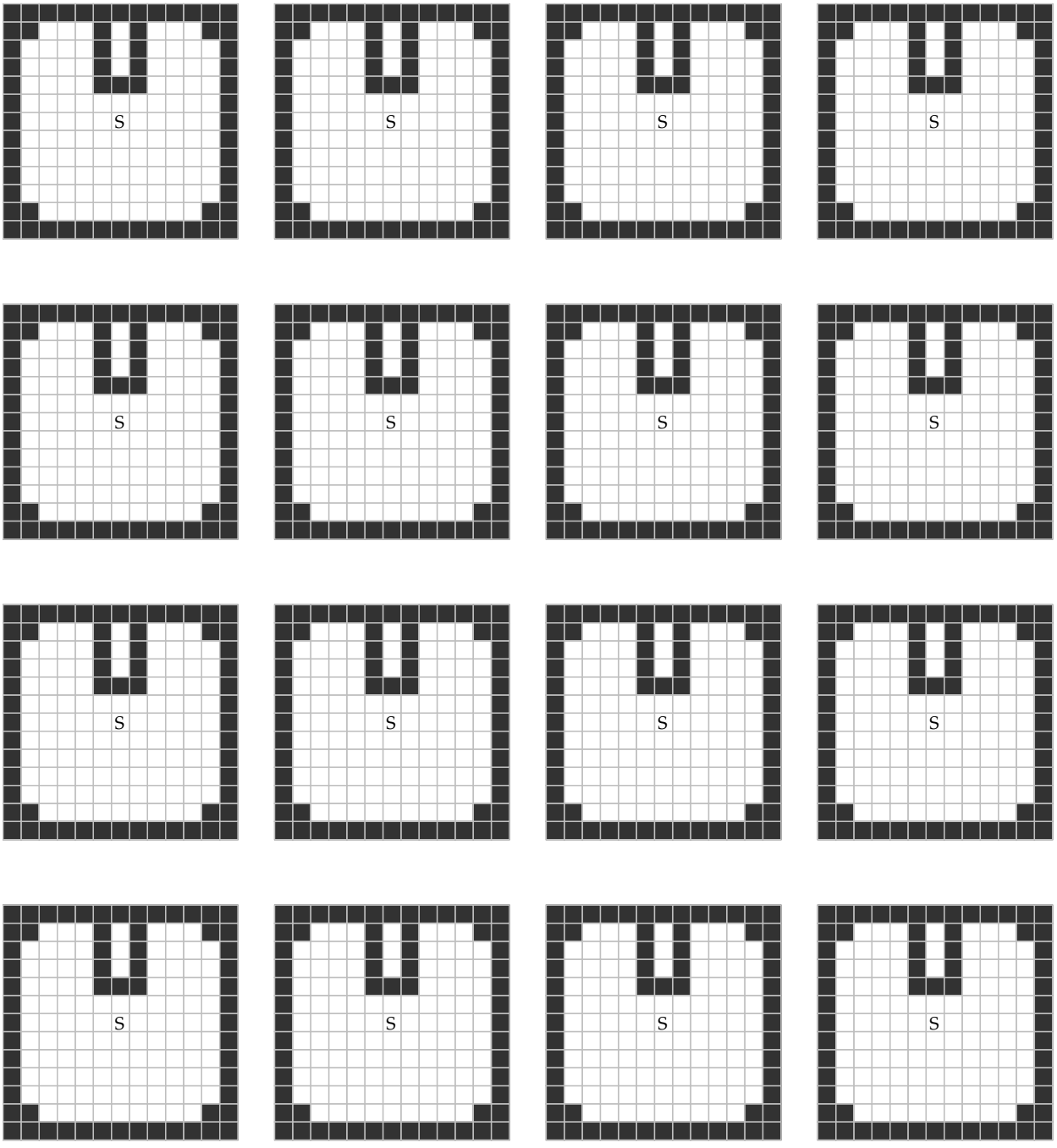


T
(25 grey pixels)

Worksheet (intentionally blank)

Optional Extra Workspace for Flood-Fill Question

Below are 16 copies of the original image from the flood-fill question, which you are welcome to use as you are working through the problem. Note that **your work on this page will not be graded**.



Worksheet (intentionally blank)

Dunder Method Summary

General:

- `print(a)` → `print(a.__str__())` or `print(a.__repr__())`
- `bool(a)` → `a.__bool__()`
- `if a:` → `if a.__bool__():`
- `a(arg1, arg2, …)` → `a.__call__(arg1, arg2, …)`

Arithmetic:

- `a + b` → `a.__add__(b)`, or `b.__radd__(a)`
- `a - b` → `a.__sub__(b)`, or `b.__rsub__(a)`
- `a * b` → `a.__mul__(b)`, or `b.__rmul__(a)`
- `a / b` → `a.__truediv__(b)`, or `b.__rtruediv__(a)`

Container Operations:

- `a[b] = c` → `a.__setitem__(b, c)`
- `a[b]` → `a.__getitem__(b)`
- `del a[b]` → `a.__delitem__(b)`
- `b in a` → `a.__contains__(b)`
- `for i in a:` → `for i in a.__iter__():`

Worksheet (intentionally blank)