# 6.101 Exam 1

Spring 2023

Name: **Answers**

Kerberos/Athena Username:

4 questions 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, but you may use one 8.5 × 11 sheet of paper (both sides) as a reference.
- You may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- 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 finishing their exams.
- You may not discuss the details of the exam with anyone other than course staff until final exam grades have been assigned and released.

# 1 The Bacon Connection

Bacon number assumes Kevin Bacon is the most connected actor, but is he? Let's find out!

The lab used a database of the form `[(actor_id1, actor_id2, film_id), ...]`. For this question, you can assume this data has already been transformed into a dictionary that maps an actor's ID number to the set of all other actor ID numbers they have acted with, not including themselves, and that all sets are non-empty.

One example of this `transformed_data` actor dictionary is shown below:

```
{
    9675: {8555},
    8555: {9675, 4724},
    4724: {8555, 2876},
    2876: {1532, 4724},
    1532: {2876},
}
```

You may assume you have access to a working `actor_path(transformed_data, start_actor, goal_test_function)` function, whose docstring is included below. You can also assume that `transformed_data` contains at least two actors, that there is a path from every actor to every other actor in the given `transformed_data`, and that all actors IDs are unique.

```
def actor_path(transformed_data, start_actor_id, goal_test_function):
    """
    Find a path between the given actor to any actor who satisfies the goal test

    Parameters:
        transformed_data: a dictionary of ids from actor -> set of actors
        start_actor_id: int unique ID of a specific actor to start from
        goal_test_function: takes a single actor ID as input, returns True if
            that actor is a valid ending location for the path, False otherwise.

    Returns:
        A list containing actor IDs, representing the shortest possible path
        from the given actor ID to any actor that satisfies the goal-test.
    """
    # some working code here
```

**Part a: Most Actors Acted With**

For this part, fill in the definition of the function below to efficiently find and return the ID of the actor who has directly acted with the most actors, or in other words has the largest number of length 2 actor paths. In the case of ties, choose the actor with the largest ID number. In the case of the example `transformed_data`, this function would return `8555`.

```
# solution 1:
def actor_most_acted_with(transformed_data):
    key_func = lambda a: (len(transformed_data[a]), a)
    return max(transformed_data, key=key_func)

# solution 2:
def actor_most_acted_with(transformed_data):
    best_actor = None
    best_num = -float("inf")
    for actor, others in transformed_data.items():
        if best_num < len(others):
            best_num = len(others)
            best_actor = actor
        elif best_num == len(others) and actor > best_actor:
            best_actor = actor
    return best_actor
```

**Part b: Lowest Average Actor Path**

For this part, fill in the definition of the function below to return the ID of the actor with the lowest average shortest path length to all other actors. In the case of ties, choose the actor with the largest ID number. In the case of the example `transformed_data`, this function would return `4724`.

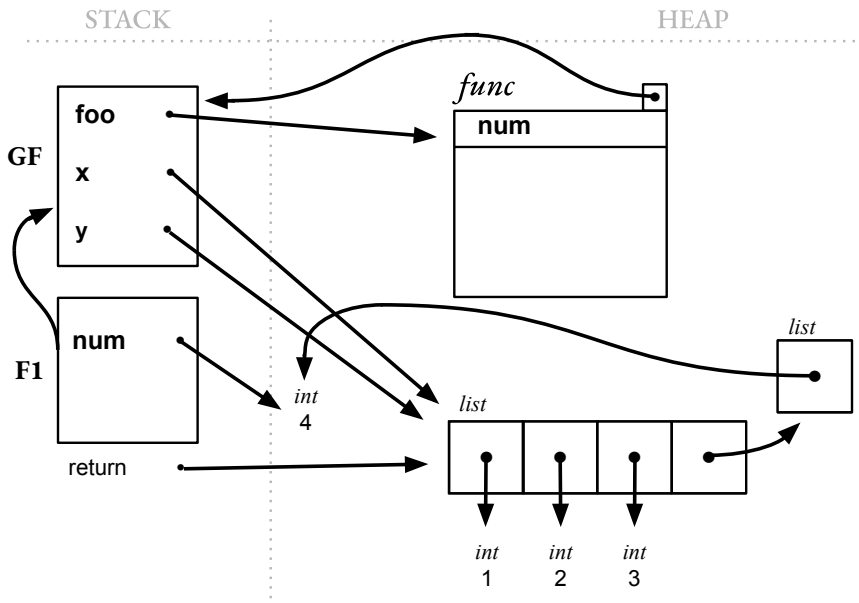
```
# solution 1
def actor_lowest_average_path_length(transformed_data):
    avg = lambda path_lens: sum(path_lens) / len(path_lens)
    key_func = lambda a: (avg(calc_path_lengths(a)), -a)
    def calc_path_lengths(ac):
        return [len(actor_path(transformed_data, ac, lambda x: x==goal))
                for goal in transformed_data]
    return min(transformed_data, key=key_func)

# solution 2
def actor_lowest_average_path_length(transformed_data):
    def calc_avg(actor):
        path_lengths = []
        for actor2 in transformed_data:
            goal = lambda x: x == actor2
            path_lengths.append(len(actor_path(transformed_data, actor, goal)))
        return sum(path_lengths) / len(path_lengths)
    best_avg = float("inf")
    best_actor = None
    for actor in transformed_data:
        avg = calc_avg(actor)
        if avg < best_avg:
            best_actor = actor
            best_avg = avg
        elif avg == best_avg and actor > best_actor:
            best_actor = actor
    return best_actor
```

## 2 Environment Diagrams

For each environment diagram, indicate whether there is a way to fill in the blanks such that executing the code would produce the diagram. If so, write the code in the box. If not, explain why.

**Diagram a.**




Is there a way to fill in the blanks in the code below to match the diagram? Circle one: **Yes** **No**

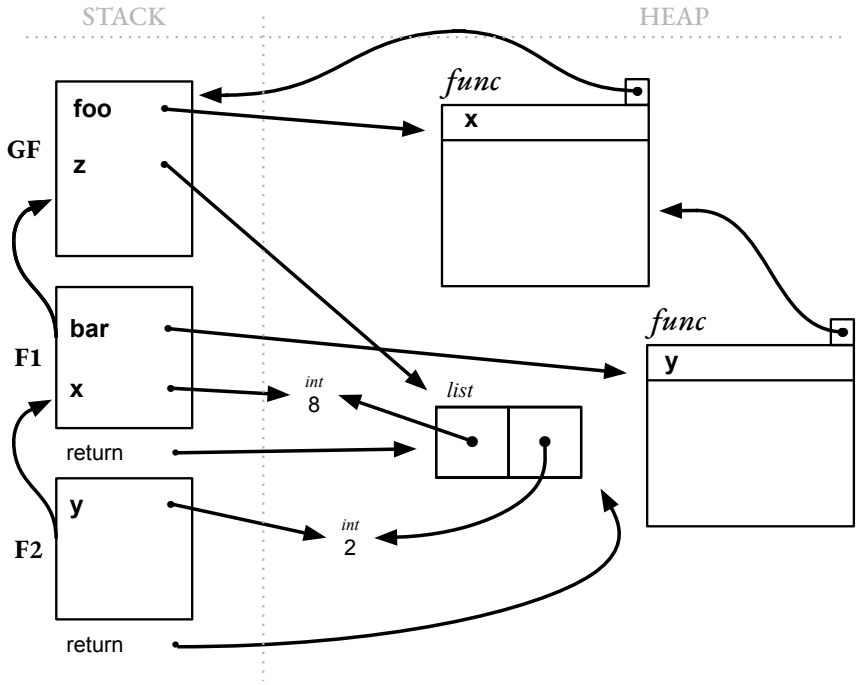
In the box below, if you indicated yes, fill in the blanks in the code. If you indicated no, write a brief justification instead.

```
x = [1, 2, 3]
def foo(num):
```

```
    x.append([num])
    return x
```

```
y = foo(4)
```

**Diagram b.**




Is there a way to fill in the blanks in the code below to match the diagram? Circle one: **Yes** **No**

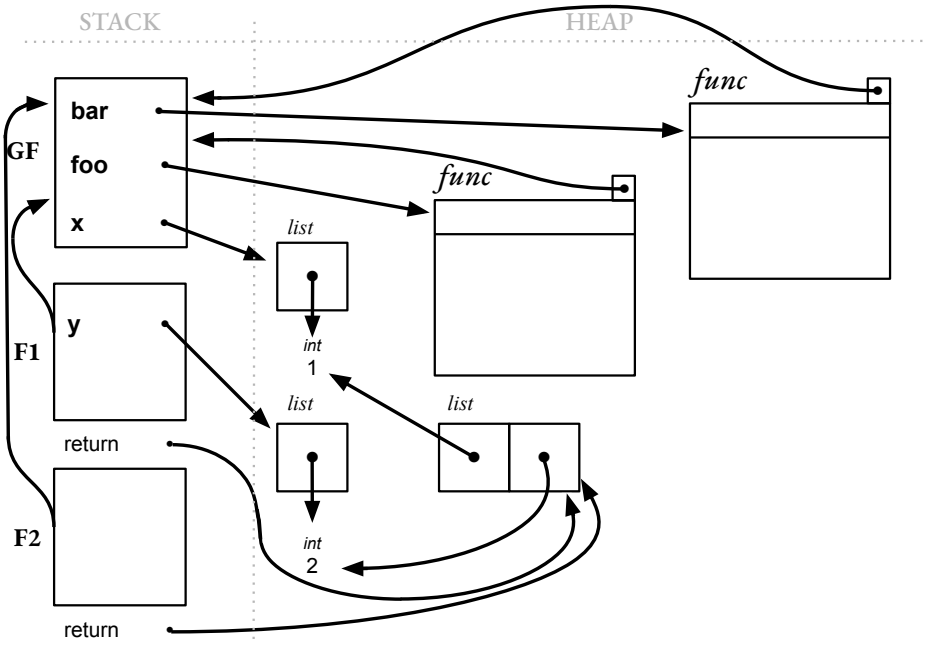
In the box below, if you indicated yes, fill in the blanks in the code. If you indicated no, write a brief justification instead.

```
def foo(x):
```

No. The diagram is impossible because the `bar` function object's enclosing frame must be a frame, not a function object.

```
z = foo(8)
```

**Diagram c.**




Is there a way to fill in the blanks in the code below to match the diagram? Circle one: **Yes** **No**

In the box below, if you indicated yes, fill in the blanks in the code. If you indicated no, write a brief justification instead.

```
def foo():
    y = [2]
    return bar()

def bar():
```

No. The call to `bar` creates F2. F2's parent frame points to the global frame and therefore it can't access `y` in F1.

Note: because of the way that integers are stored in Python's heap, `bar` could refer to the `int 2` without accessing `y`, so the following is also an acceptable answer:

```
return [x[0], 2]
```

```
x = [1]
foo()
```

# 3 Zipping Together Linked Lists

Ben Bitdiddle and his friends are learning about recursion in their programming class 1.016. Their instructor has given them an assignment to write a function called `zip_lists`, which combines elements in two linked lists into one linked list of tuples containing the elements from both lists.

Each linked list has zero or more elements. For the purposes of this function, an empty linked list is represented as an empty list, a linked list with a single element is a list of length one, and a linked list with more than one element is represented as a list containing `[element, rest]`, where rest is a linked list.

Ben's instructor provided the following test cases to verify the correctness of Ben and his friends' submissions, shown below:

```
def test1():
    assert zip_lists([], [6, [7, [8, [9]]]]) == []
    assert zip_lists([6, [7, [8, [9]]]], []) == []

def test2():
    assert zip_lists([1], [2]) == [(1, 2)]

def test3():
    linked1 = [1, [2, [3, [4]]]]
    linked2 = [6, [7, [8, [9]]]]
    expected = [(1, 6), [(2, 7), [(3, 8), [(4, 9)]]]]
    assert zip_lists(linked1, linked2) == expected

def test4():
    linked1 = [1, [2, [3, [4]]]]
    linked2 = [9, [8, [7, [6, [5]]]]]
    expected = [(1, 9), [(2, 8), [(3, 7), [(4, 6)]]]]
    assert zip_lists(linked1, linked2) == expected

def test5():
    linked1 = [[1], [[2], [[3], [[4]]]]]
    linked2 = [1, [2, [3, [4]]]]
    expected = [([1], 1), [([2], 2), [([3], 3), [([4], 4)]]]]
    assert zip_lists(linked1, linked2) == expected
```

Unfortunately, the automatic grading infrastructure on the course website broke and the instructor has asked you to manually grade the submissions. For each of the following implementations of `zip_lists`, circle the test cases that would pass.

Note: there is a detachable reference sheet containing a copy of the test cases at the end of the exam.

**Implementation a:**

```
def zip_lists(linked1, linked2):
    result = [(linked1[0], linked2[0])]
    if len(linked1) == 1 or len(linked2) == 1:
        return result
    result.append(zip_lists(linked1[1], linked2[1]))
    return result
```

**test1** **test2** **test3** **test4** **test5**

Fails test1 (length 0 lists not handled); passes all other tests.

**Implementation b:**

```
def zip_lists(linked1, linked2):
    if not linked1 or not linked2:
        return []
    start = [(linked1[0], linked2[0])]
    end = zip_lists(linked1[1:], linked2[1:])
    if end:
        start.append(end)
    return start
```

**test1** **test2** **test3** **test4** **test5**

Passes test1 and test2; fails otherwise.

**Implementation c:**

```
def zip_lists(linked1, linked2):
    if not linked1 or not linked2:
        return []
    return [(linked1[0], linked2[0]), zip_lists(linked1[1], linked2[1])]
```

**test1** **test2** **test3** **test4** **test5**

Passes test1, fails everything else.

**Implementation d:**

```
def zip_lists(linked1, linked2):
    start = [(linked1[0], linked2[0])]
    if len(linked1) == len(linked2) == 1:
        return start
    start.append(zip_lists(linked1[1], linked2[1]))
    return start
```

**test1** **test2** **test3** **test4** **test5**

Passes test2, test3, and test5 (does not handle length 0 lists or lists of non-equal lengths).

# 4 Name Chains

We gave Louis Reasoner a list of tuples that represent a person's first and last names, in `(first_name, last_name)` format, such that each person in the list has a unique combination of first and last names.

```
names = [ ('Amelia',  'Harper'),
          ('Bobby',   'Darwin'),
          ('Darby',   'Edwards'),
          ('Ellis',   'Roberts'),
          ('Harper',  'Jackson'),
          ('Harper',  'Kali'),
          ('Harper',  'Madison'),
          ('Kali',    'Harper'),
          ('Madison', 'Dempsey'),
          ('Madison', 'Bobby'),
          ('Bobby',   'Madison'),
        ]
```

We asked Louis to write code that, starting at a given person, would find the longest chain of names where the last name of the current link matches the first name of the next link, and each person on the chain can appear only once. An example three-link chain from this database might be:

```
 [ ('Amelia', 'Harper'),
   ('Harper', 'Madison'),
   ('Madison', 'Dempsey') ]
```

since Amelia Harper's last name is the same as Harper Madison's first name, and Harper Madison's last name is the same as Madison Dempsey's first name. To identify even longer chains, Louis started with a helper function to find all links that matched a given name, shown below. His code works as described.

```
def get_links(names, name_to_match):
    """
    Given a database and the name that needs to be matched,
    returns a list of tuples from people in the database
    with first names the same as name_to_match.
    """
    return [
            (first_name, last_name)
                for first_name, last_name in names
                    if first_name == name_to_match
           ]
```

**Part a:**
Louis then wrote `longest_chain_finder` on the next page. Here, he made **many** mistakes (sometimes more than one on a line!), except in the docstring. Help Louis out by fixing his code. Write your corrections on the next page, crossing out portions of Louis's code that you wish to change, and writing in your updates.

```
def longest_chain_finder(database, name):
    """
    Given three arguments, a database of names, a first_name
    string and a last_name string, returns a list of name
    tuples that is the longest valid chain of names starting at
    at the person in (first_name, last_name).
    """
    chains_found = [ (last_name, first_name) ]
    agenda = chains_found()
    while 0 < agenda:
        chain = agenda[0]
        last_link = chain[len(chain)]
        print(f'{last_link=')
        next_links = get_links(last_link)
        if next_link:
            for new_link in enumerate(next_links):
                if new_link in chain:
                    new_chain = chain()
                    new_chain[len(new_chain)] = new_link
                    agenda.extend(next_links)
                    print(f'added {new_links=}')
        else:
            agenda.append(chain)

    longest_chain = []
    max_length    = -1
    for chain in agenda:
        if len(chain) < max_len:
            longest_chain = chain
            max_length    = len(longest_chain) - 1
    return longest_chain
```

# GENERAL

```
def longest_chain_finder(database, name):  first_name, last_name):
    """
    Given three arguments, a database of names, a first_name
    string and a last_name string, returns a list of name
    tuples that is the longest valid chain of names starting at
    at the person in (first_name, last_name).
    """
    chains_found = [[(last_name, first_name)]]   first  last
    agenda = chains_found[]   .copy or [:]
    while 0 < agenda:   or while 0 < len(agenda)
        chain = agenda[0]   .pop(0)  if pop() then DFS
        last_link = chain[len(chain)]   or chain[len(chain)-1]
        print(f'{last_link=}')
        next_links = get_links(last_link)   (database, last_link[1])   or -1
        if next_link:   s
            for new_link in enumerate(next_links):
                if new_link in chain:   not
                    new_chain = chain()   .copy()  or [:]
                    new_chain[len(new_chain)] = new_link   .append(new_link)
                    agenda.extend(next_links)   append(new_chain)
                    print(f'added {new_links=}')   chain
        else:
            agenda.append(chain)   chains_found   } or move to and delete else


    longest_chain = []
    max_length    = -1   0  -1 is ok
    for chain in agenda:   chains_found
        if len(chain) > max_len:   gth
            longest_chain = chain
            max_length    = len(longest_chain) - 1
    return longest_chain   will work if not removed
```

return max(chains_found, key=len)

or

return chains_found[-1]

COMPACT

```
def longest_chain_finder(database, name): first_name, last_name):
    """
    Given three arguments, a database of names, a first_name
    string and a last_name string, returns a list of name
    tuples that is the longest valid chain of names starting at
    at the person in (first_name, last_name).
    """
    chains_found = [[(last_name, first_name)]]
    agenda = chains_found()
    while 0 < agenda: for chain in chains_found:
        chain = agenda[0]
        last_link = chain[len(chain)-1]
        print(f'{last_link=}')
        next_links = get_links(last_link) (database, last_link[1])
        if next_link:
            for new_link in enumerate(next_links):
                if new_link not in chain:
                    new_chain = chain() + [new_link]
                    new_chain[len(new_chain)] = new_link
                    agenda.extend(next_links) chains_found.append(new_chain)
                    print(f'added {new_link=}') or new_chain
        else:
            agenda.append(chain)

    longest_chain = []
    max_length    = -1
    for chain in agenda:
        if len(chain) < max_len:
            longest_chain = chain
            max_length    = len(longest_chain) - 1
    return longest_chain chains_found[-1]
```

**Part b:**

Louis tries out the code you fixed to see what the longest chain from Amelia Harper is. Assuming your code works correctly and finds longest chains, what value should the following return?

```
longest_chain_finder(names, 'Amelia', 'Harper')
```

```
[('Amelia',  'Harper'),
 ('Harper',  'Kali'),
 ('Kali',    'Harper'),
 ('Harper',  'Madison'),
 ('Madison', 'Bobby'),
 ('Bobby',   'Madison'),
 ('Madison', 'Dempsey')]
```

**Part c:**

Your corrections on Louis's code implement a **BREADTH-FIRST** / **DEPTH-FIRST** approach (circle one).

Describe in detail the changes necessary to turn it into the other approach. You may use English or Python.

BREADTH-FIRST

Replace

```
        chain = agenda.pop(0)
```

with

```
        chain = agenda.pop()
```

The code (as corrected in Part A) adds `new_chain` values to be explored at the tail of `agenda`, and takes the next `chain` from the front, making it a BFS search. To turn it into a DFS search, the next `chain` needs to be taken from the same end of the `agenda` that `new_chain` values are added. (Or vice-versa, depending on how one has corrected the code in Part A.)