DS C88C Computational Structures in Data Science
Spring 2023

FINAL

INSTRUCTIONS

- **Do NOT open the exam until you are instructed to do so!**
- You **must not** collaborate with anyone inside or outside of C88C.
- You **must not** use any internet resources to answer the questions.
- If you are taking an online exam, at this point you should have started your Zoom / screen recording. If something happens during the exam, focus on the exam! Do not spend more than a few minutes dealing with proctoring.
- When a question specifies that you must rewrite the completed function, you should **not** recopy the doctests.
- The exam is closed book, closed computer, closed calculator, except your hand-written 8.5" x 11" cheat sheets of your own creation and the official C88C Reference Sheet

Full Name	
Student ID Number	
Official Berkeley Email (@berkeley.edu)	`_______________________________@berkeley.edu`
What room are you in?	
Name of the person to your left	
Name of the person to your right	
By my signature, I certify that all the work on this exam is my own, and I will not discuss it with anyone until exam session is over. **(please sign)**	

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
  - **You must include all answers within the boxes.**
  - **Online Exams: You may start your exam as soon as you are given the password.**
    - **You may have a digital version of the C88C Reference Sheet, but no other files.**
    - Exam Clarifications: https://tinyurl.com/clarifications-sp23
    - Reference Sheet: https://tinyurl.com/mt-reference

1. **(5.0 points) ConceptMan**

ConceptMan is a superhero who answers multiple choice questions on the Spring 2023 CS 88 Final. Help ConceptMan answer these multiple choice questions on the Spring 2023 CS 88 Final!

(a) (1.0 pt)

```
d = {1:2}
try:
    print(d[1])
except NameError as e:
    print('nyaoooooooooooooooo')
except KeyError as e:
    print('donkey')
except ZeroDivisionError as e:
    print('Wowzas!')
```

○ `Wowzas!`

○ `donkey`

○ `nyaoooooooooooooooo`

● `2`

○ `None`

(b) (1.0 pt)

```
d = {1:2}
try:
    print(d[2])
except NameError as e:
    print('nyaoooooooooooooooo')
except KeyError as e:
    print('donkey')
except ZeroDivisionError as e:
    print('Wowzas!')
```

● `donkey`

○ `nyaoooooooooooooooo`

○ `Wowzas!`

○ `None`

○ `2`

(c) (1.0 pt)

```
d = {1:2}
try:
    print(d[1/0])
except NameError as e:
    print('nyaoooooooooooooo')
except KeyError as e:
    print('donkey')
except ZeroDivisionError as e:
    print('Wowzas!')
```

○ `None`

○ `2`

○ `donkey`

● `Wowzas!`

○ `nyaoooooooooooooo`

(d) (1.0 pt) Consider this implementation of a Link from lecture. No other methods are implemented.

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

Does the following code follow the programming principles we learned in class?

```
x = Link(1, Link(2))
x.rest.rest = 3
```

● No, because this would result in an invalid `Link` object.

○ Yes, because the code is syntactically correct.

○ Yes, because the code uses the attributes of the class.

○ No, because while directly setting the data value of the first Link is acceptable, doing so for later Links requires a setter method.

(e) (1.0 pt) Which SQL keyword is used to filter the groups created by GROUP BY?

○ `WITH`

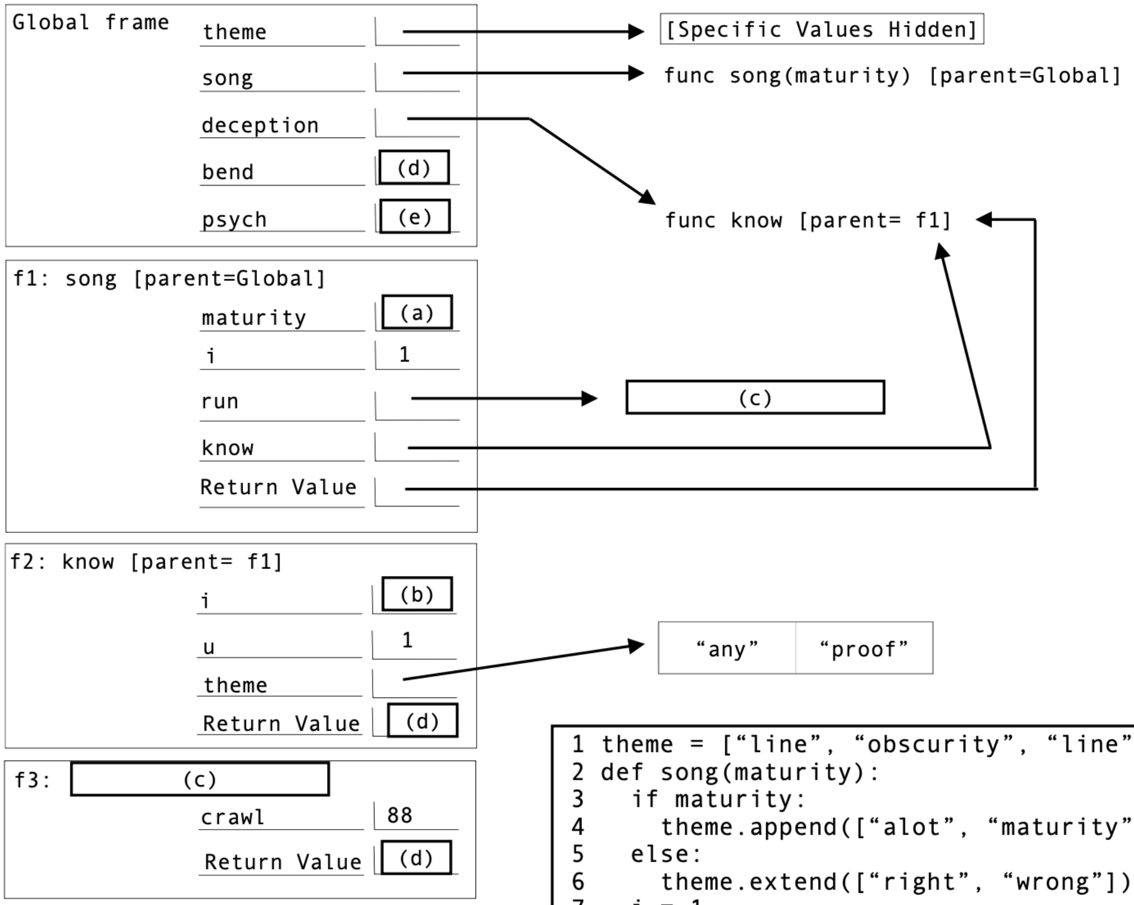
● `HAVING`

○ `SELECT`

○ `WHERE`

2. (10.0 points) Psych!

Fill in the blanks to complete the environment diagram. All the code used is in the box to the right, and the code runs to completion. Boxes with the same label will have the same value.



```
theme = [“line”, “obscurity”, “line”]
def song(maturity):
  if maturity:
    theme.append([“alot”, “maturity”])
  else:
    theme.extend([“right”, “wrong”])
  i = 1
  run = lambda crawl: theme[1]
  def know(i, u):
    theme = [“any”, “proof"]
    if i and u:
      return run(88)
    else:
      return run
  return know

deception = song(8 % 4)
bend = deception(-1, 1)
psych = len(theme)
```

(a) (2.0 pt) What is the value of the variable `maturity` in frame `f1`? (box a)

0

(b) **(2.0 pt)** What is value of bend at the end of the environment diagram (box d)?

○ `["alot", "maturity"]`

○ `"alot"`

● `"obscurity"`

○ `"proof"`

○ `"any"`

○ `"wrong"`

○ `"line"`

(c) **(2.0 pt)** What is value of `i` in frame 2 at the end of the environment diagram (box b)?

○ `"proof"`

● `"obscurity"`

○ `"any"`

○ `"line"`

○ `"alot"`

○ `["alot", "maturity"]`

○ `"wrong"`

(d) **(2.0 pt)** What function is being called in the f3 frame? (box c)

○ `func know [parent = f2]`

○ `func song [parent = global]`

○ `func lambda <line 8> [parent = f2]`

● `func lambda <line 8> [parent = f1]`

○ `func know [parent = f1]`

(e) **(2.0 pt)** What is value of `psych` at the end of the environment diagram (box e)?

5

3. (8.0 points) What Would Python Do (WWPD)

For each expression below, select the output displayed by the interactive Python interpreter when the expression is evaluated. If an error occurs, select "Error".

```
>>> func = lambda x, y: add(mystery(x), y)

>>> def add(x, y):
...     return x + y

>>> def cond(x):
...     return x < (lambda y: y / x)(18)

>>> def mystery(x):
...     arr = [1, 4, 5, 2, 3]
...     return (lambda x: (arr[-1] * x))(x)
...     x += 1
```

(a) (1.5 pt)

```
>>> mystery(0)
```

○ Error

○ 12

○ 6

● 0

(b) (1.5 pt)

```
>>> mystery(2)
```

● 6

○ 0

○ 12

○ Error

(c) (1.5 pt)

```
>>> cond(0)
```

○ True

● Error

○ False

(d) (1.5 pt)

```
>>> cond(9)
```

○ Error

○ True

● False

(e) (2.0 pt)

```
>>> func(4, -2)
```

○ `18`

○ `Error`

○ `14`

● `10`

4. (4.0 points) Bugging Out

We are writing the function `cumulative_link` which accepts a linked list `lnk` as input and *mutates* it such that each node is the sum of all nodes to the right in the old `lnk`. An example of the desired behavior is below:

```
>>> lnk = Link(1, Link(2, Link(3)))
>>> cumulative_lnk(lnk)
>>> lnk
Link(6, Link(5, Link(3)))
```

Below is a buggy implementation of `cumulative_link`. Answer the following questions with this implementation in mind. **Hint:** It can be helpful to draw out the linked lists to follow the execution of the code. The inputs are small enough that this should be possible!

```
1.  def cumulative_link(lnk):
2.      if lnk.rest is Link.empty:
3.          return
4.      else:
5.          cumulative_link(lnk.rest)
6.          lnk.first = 1 + lnk.rest.first
```

(a) **(1.0 pt)** Select the value of `lnk` such that buggy `cumulative_link(lnk)` mutates `lnk` **correctly** and **does not error**.

● `lnk = Link(1, Link(1, Link(5)))`

○ `lnk = Link("s", Link("u", Link("m")))`

○ `lnk = Link("h", Link("i", Link(88)))`

(b) **(1.0 pt)** Select the value of `lnk` such that buggy `cumulative_link(lnk)` **causes an error**.

○ `lnk = Link("h", Link("i", Link(88)))`

○ `lnk = Link(1, Link(1, Link(5)))`

● `lnk = Link("s", Link("u", Link("m")))`

(c) **(1.0 pt)** Select the value of `lnk` such that buggy `cumulative_link(lnk)` mutates `lnk` **incorrectly** and **does not error**.

○ `lnk = Link(1, Link(1, Link(5)))`

○ `lnk = Link("s", Link("u", Link("m")))`

● `lnk = Link("h", Link("i", Link(88)))`

(d) **(2.0 pt)** The bug in this function can be fixed by modifying one line of code. Indicate which line you are changing, and how you will replace it.

Line 6, `lnk.first = lnk.first + lnk.rest.first`

5. (7.0 points) Multi-Caller

(a) **(7.0 pt)** You're writing a function for people who are too lazy to call functions themselves. Write the `multi_caller` function, which will take in `functions` (a list of two-argument functions) and return a function `apply`. This `apply` function takes in `arguments` (a list of two-element lists) and returns a new list containing the outputs from calling each function in `functions` on its respective arguments in `arguments`.

The first and second arguments for the function at `functions[i]` will be the first and second elements of the list at `arguments[i]` respectively. You can assume that `functions` and `arguments` will always have the same length.

```
def multi_caller(functions):
    """
    >>> add = lambda a, b: a + b
    >>> sub = lambda a, b: a - b
    >>> mul = lambda a, b: a * b
    >>> functions = [add, sub, mul]
    >>> apply = multi_caller(functions)
    >>> arguments = [[2, 5], [10, 5], [11, 8]]
    >>> apply(arguments) # [add(2,5), sub(10, 5), mul(11, 8)]
    [7, 5, 88]
    """
    def apply(arguments):
        result = ______________________________
        for i in range(______________________________)
            func = ______________________________
            args = ______________________________
            output = ______________________________
            ______________________________
        return ______________________________
    return ______________________________
```

```
def multi_caller(functions):
    def apply(arguments):
        result = []
        for i in range(len(arguments)):
            func = functions[i]
            args = arguments[i]
            output = func(args[0], args[1])
            result.append(output)
        return result
    return apply
```

6. (6.0 points) Product of List

(a) **(6.0 pt)** Given a nested list, return the product of all the elements using recursion. For an empty list, the product returned should be 1.

Hint: The `isinstance` function will be useful!

To review, `isinstance` takes in two arguments: an instance and the name of a class. `isinstance` returns `True` if the first argument is an instance of the class of the second argument, and `False` otherwise. Lists belong to the `list` class.

```
def nested_product(lst):
    """
    >>> lst = [2, 3, [4, 1, [5], 1]]
    >>> nested_product(lst)
    120 # 2 * 3 * 4 * 1 * 5 * 1

    >>> lst2 = [1, [2, [3]]]
    >>> nested_product(lst2)
    6

    >>> nested_product([])
    1
    """
    if not lst:
        return ______________________________
    if ______________________________:
        return ___________________ * ____________________
    else:
        return ___________________ * ____________________
```

```
def nested_product(lst):
    if not lst:
        return 1
    if isinstance(lst[0], list):
        return nested_product(lst[0]) * nested_product(lst[1:])
    else:
        return lst[0] * nested_product(lst[1:])
```

7. (4.0 points) Test Distribution

(a) **(4.0 pt)** Given a list of test scores that range from 0 to 100, create a function that returns a dictionary `distribution` that contains bins of test scores in 10s, and the percentage of scores in those bins. Your dictionary should have keys 0, 10, 20, 30, . . . , 90, each corresponding to one bin in the distributionogram.

Bin 0 should hold the percentage of test scores that were between 0-10 (excluding 10), bin 10 should hold the percentage of test scores that were between 10-20 (excluding 20), etc.

You do not need to include bins with no test scores, and order of bins do not matter.

```
def dist_dictionary(scores):
    """
    >>> test_scores = [11.2, 13.4, 6, 78.5, 92.6]
    # 11.2 and 13.4 belong in the 10 bin as they fall between 10-20 (excluding 20)
    # 6 belongs in the 0 bin as it falls between 0-10 (excluding 10)
    # 78.5 belongs in the 70 bin as it falls between 70-80 (excluding 80)
    # 92.5 belongs in the 100 bin as it falls between 90-100 (excluding 100)

    >>> dist = dist_dictionary(test_scores)
    >>> dist
    {0:0.2, 10:0.4, 70:0.2, 90:0.2}
    """
    distribution = {}
    for score in scores: # getting counts of scores
        bin = ______________________________
        if bin in distribution:
            ______________________________
        else:
            ______________________________

    for bin in distribution: # making every value into a decimal value
        ______________________________
    return distribution
```

```
def dist_dictionary(scores):
    distribution =
    for score in scores:
        bin = (score // 10) * 10
        if bin in distribution:
            distribution[bin] += 1
        else:
            distribution[bin] = 1
    for bin in distribution:
        distribution[bin] = distribution[bin] / len(scores)
    return distribution
```

8. (5.0 points) Cherry Tree Blossom

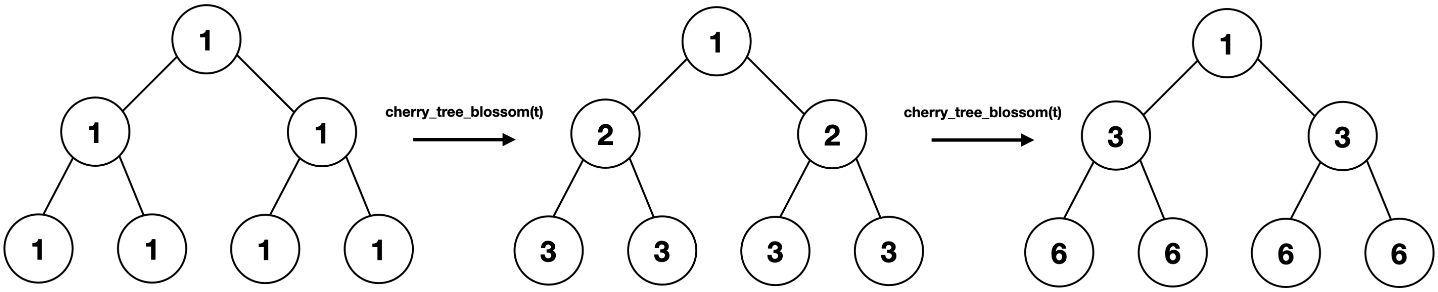
In memory of the full blossom of the cherry trees on campus, you decided to write a function that simulates this process.

A cherry tree is a `Tree` instance where each node contains an integer value.

When a tree blossoms, start from the top of the tree and increment every node's value by its parent's value. The root node of the tree will not be incremented.

Implement `cherry_tree_blossom`, which takes in a `Tree` instance and makes it blossom by mutating the tree given.

The doctests are depicted here:



Note: The two bottom lines given are meant to be indented so that they are inside the for loop. You cannot use more lines than given.

```
def cherry_tree_blossom(t):
    """
    >>> t = Tree(1, [Tree(1, [Tree(1), Tree(1)]), Tree(1, [Tree(1), Tree(1)])])
    >>> print(t)
    1
      1
        1
        1
      1
        1
        1
    >>> cherry_tree_blossom(t)
    >>> print(t)
    1
      2
        3
        3
      2
        3
        3
    >>> cherry_tree_blossom(t)
    >>> print(t)
    1
      3
        6
        6
      3
        6
        6
    """
    for _______________________________________________:
        _______________________________________________
        _______________________________________________
```

(a) (4.0 pt)

300.0px0.75

```
def cherry_tree_blossom(t):
    """
    »> t = Tree(1, [Tree(1, [Tree(1), Tree(1)]), Tree(1, [Tree(1),
Tree(1)])])
    »> print(t)
    1
      1
        1
        1
      1
        1
        1
    »> cherry_tree_blossom(t)
    »> print(t)
    1
      2
        3
        3
      2
        3
        3
    »> cherry_tree_blossom(t)
    »> print(t)
    1
      3
        6
        6
      3
        6
        6
    """
    for b in t.branches:
        b.value += t.value
        cherry_tree_blossom(b)
```

9. (8.0 points) Out of Order

(a) **(8.0 pt)** You are very picky about your linked lists and need them to be sorted. Given a linked list `lnk` containing integers, write the function `out_of_order` that returns the number that breaks the sorted property in `lnk`. In addition, the function should mutate `lnk` to skip over the value. If `lnk` is already sorted, return `None` without mutating `lnk`.

In order for `lnk` to follow the sorted property, all of its values must be larger than or equal to the value before it.

You can assume that `lnk` will have a maximum of one value that breaks the sorted property. Further, you can assume that after you remove the unsorted value, `lnk` will be sorted.

```
def out_of_order(lnk):
    """
    >>> lnk1 = Link(1, Link(2, Link(2, Link(0))))
    >>> out_of_order(lnk1)
    0
    >>> lnk1
    Link(1, Link(2, Link(2)))
    >>> lnk2 = Link(5, Link(3, Link(7, Link(9))))
    >>> out_of_order(lnk2)
    3
    >>> lnk2
    Link(5, Link(7, Link(9)))
    >>> lnk3 = Link(6, Link(7, Link(10, Link(11))))
    >>> out_of_order(lnk3) # lnk is already sorted, returns None
    >>> lnk3 # Not mutated
    Link(6, Link(7, Link(10, Link(11))))
    """
    if ________________________________________:
        return ________________________________________
    if ________________________________________:
        unsorted_value = ________________________________________
        ________________________________________ = ________________________________________
        return ________________________________________
    return ________________________________________
```

```
def out_of_order(lnk):
    if lnk is Link.empty or lnk.rest is Link.empty:
        return None
    if lnk.first > lnk.rest.first :
        unsorted_value = lnk.rest.first
        lnk.rest = lnk.rest.rest
        return unsorted_value
    return out_of_order(lnk.rest)
```

10. (15.0 points) Final Countdown

To make writing exams easier, C88C has decided to use Object-Oriented Programming.

Specifically, we have defined a `Question` class. Unfortunately, the class is incomplete! Each `Question` instance should have the following instance attributes:

- `self.text`: Contains the question text
- `self.solution`: Contains the question solution
- `self.points`: Contains the number of points the question is worth
- `self.serial_no`: The unique question serial number

```
class Question:
    serial_no = 1000
    def __init__(self, text, solution, points):
        """
        >>> q1 = Question("True or False?", "False", 2)
        >>> q1.serial_no
        1000
        >>> q2 = Question("True or False?", "True", 1)
        >>> q2.serial_no
        1001
        >>> q1.grade("True")
        0
        >>> q1.grade("False")
        2
        """
        self.text = text
        self.solution = solution
        self.points = points
        self.serial_no = ____________________________
        ____________________________

    def grade(self, student_sol):
        if ______________________________:
            return ______________________________
        else:
            return 0
```

(a) **(2.0 pt)** To ensure each `Question` instance gets a unique serial number, we have a class attribute `Question.serial_no` which stores the serial number for the next instance to be created.

Complete the constructor of the `Question` class to populate the instance attribute `serial_no`. Remember to update the **class** attribute `serial_no` so that subsequent calls to the constructor do not use the same serial number.

```
class Question:
    serial_no = 1000
    def __init__(self, text, solution, points):
        self.text = text
        self.solution = solution
        self.points = points
        self.serial_no = Question.serial_no
        Question.serial_no += 1
```

(b) **(2.0 pt)** In addition to storing question information, we want to be able to grade questions. Implement the method `grade` which accepts `student_sol` as an argument. If `student_sol` is equal to the question's `solution`, return the `points` that the question is worth. Else, return `0`.

```
def grade(self, student_sol):
    if self.solution == student_sol:
        return self.points
    else:
        return 0
```

(c) (2.0 pt) Now that we have a `Question` class, let's put some questions together into an `Exam` class!

```
class Exam:
    def __init__(self, questions, submissions):
        """
        >>> q1 = Question("True or False?", "False", 2)
        >>> q2 = Question("True or False?", "True", 1)
        >>> final = Exam([q1, q2], [{1000: "False", 1001: "True"}, {1000: "False", 1001: "False"}])
        >>> final.scores # final.scores is empty initially
        []
        >>> final.total_score()
        3
        >>> final.grade_submission(final.submissions[0])
        3
        >>> final.scores # calling grade_submission does not modify final.scores
        []
        >>> final.grade_all()
        >>> final.scores # calling grade_all modifies final.scores
        [3, 2]
        """
        self.questions = questions
        self.submissions = submissions
        self.scores = []

    def total_score(self):
        return sum([______________ for q in __________________________________])

    def grade_submission(self, submission):
        score = _____________________________________________
        for ____________________________________________:
            score += ______________________.grade(__________________________)
        return score

    def grade_all(self):
        for ____________________________________________:
            _________________________.append(_________________________________)
```

Instances of the `Exam` class have the following instance attributes:

- `self.questions`: A list of `Question` instances that make up the exam
- `self.submissions`: A list of student submissions. Each submission is a dictionary mapping a question serial number to the student's answer.
- `self.scores`: A list containing student submission scores

Implement the method `total_score` which returns the total points that can be scored on the exam. The total score is equal to the sum of the `points` values of all questions on the `Exam`.

```
def total_score(self):
    return sum([q.points for q in self.questions])
```

(d) **(3.0 pt)** Implement the method `grade_submission` which take in a student's submission and returns the score the student scored. As mentioned previously, each submission is a dictionary mapping a question serial number to the student's answer. Each submission will contain all questions on the exam.

Each student's overall score is equal to the sum of their question scores. Recall: to get a student's score on a question we can use the `.grade()` method.

```
def grade_submission(self, submission):
    score = 0
    for q in self.questions:
        score += q.grade(submission[q.serial_no])
    return score
```

(e) **(3.0 pt)** Implement the method `grade_all` which adds the overall scores of all submissions to the `self.scores` list. The `grade_submission` method will be useful here!

```
def grade_all(self):
    for s in self.submissions:
        self.scores.append(self.grade_submission(s))
```

(f) **(1.0 pt)** Congratulations on implementing the `Question` and `Exam` classes! C88C TAs have one last question about the design of these classes. Currently, neither `Question` nor `Exam` inherit from each other. Should `Question` be a subclass of `Exam`?

○ No, the `Question` class implements methods that `Exams` should not have

● No, the `Exam` class implements methods that `Questions` should not have

○ Yes, a `Question` is a type of `Exam`

11. (5.0 points) Tree Sort

Write a generator function, `tree_sort`, that takes in a `Tree` instance whose values are integers and yields all values from the tree in ascending order. You may assume that all values in the tree are unique.

```
def tree_sort(t):
    """
    >>> t = Tree(1, [Tree(3, [Tree(4)]), Tree(2), Tree(6, [Tree(5)])])
    >>> for val in tree_sort(t):
    ...     print(val)
    ...
    1
    2
    3
    4
    5
    6
    """
    # create a list of generators, one for each branch
    sorted_branches = [tree_sort(b) for b in t.branches]
    # smallest values from each branch
    next_smallest = [next(gen) for gen in sorted_branches]
    value_yielded = False

    while len(next_smallest) > 0:
        # find the index of the smallest value from the branches
        min_index = min(range(len(next_smallest)), key=lambda i: ____(a)____)

        if not value_yielded and ____(b)____:
            yield t.value
            value_yielded = True
        yield ____(c)____

        try:
            # update the value at the yielded position
            next_smallest[min_index] = ____(d)____
        except StopIteration: # if no elements left in the generator
            next_smallest.____(e)____(min_index)
            sorted_branches.____(e)____(min_index)

    # yield the value of t if it is not yielded during the while loop
    if not value_yielded:
        yield t.value
```

(a) (1.0 pt) Which of the following can fill in blank (a)?

○ `next(sorted_branches[i])`

○ `i`

● `next_smallest[i]`

○ `sorted_branches[i]`

(b) (1.0 pt) Fill in blank (b)

`t.value < next_smallest[min_index]`

(c) **(1.0 pt)** Fill in blank (c)

```
next_smallest[min_index]
```

(d) **(1.0 pt)** Fill in blank (d)

```
next(sorted_branches[min_index])
```

(e) **(1.0 pt)** Which of the following can fill in blank (e)? The two blanks labeled (e) in the skeleton are supposed to have the same solution.

○ `append`

○ `extend`

○ `remove`

● `pop`

12. (4.0 points) JoJo's Bizarre Squared-venture

Jolyne and Johnny are trying to compare different ways of squaring a positive number `n`. Answer the following questions with the WORST-CASE time complexity of each function. Assume each basic operation (addition, multiplication, assignment) takes constant time.

(a) (1.0 pt)

```
def squared_platinum(n):
    res = 0
    for za_warudo in range(n):
        res += n
    return res
```

○ `O(1)`
● `O(n)`
○ `O(log(n))`
○ `O(n^2)`
○ `O(2^n)`

(b) (1.0 pt)

```
def squared-mit_purple(n):
    return n * n
```

○ `O(log(n))`
○ `O(2^n)`
○ `O(n)`
● `O(1)`
○ `O(n^2)`

(c) (1.0 pt)

```
def squared_experience(n):
    if n % 2 == 0:
        return 4 * square(n // 2)
    return n * n
```

○ `O(n)`
○ `O(n^2)`
○ `O(2^n)`
○ `O(1)`
● `O(log(n))`

(d) (1.0 pt)

```
def squared-y_diamond(n):
    res = 0
    for kira_queen in range(n):
        for j in range(n):
            res += 1
    return res
```

○ `O(log(n))`

● `O(n^2)`

○ `O(1)`

○ `O(n)`

○ `O(2^n)`

13. (10.0 points) Froggy Friends

You have quirky friends who give off very interesting vibes. You have a table `friends` that contains demographic information about your friends, including the column `animal` which is what type of `animal` energy you think they give off the most, e.g. golden retriever energy. You also have a table `animals` that describes each animal, the `animal` column in `friends` corresponds to the animal column in `animals`. Not all rows are shown for brevity, but your code should work even if there are more rows (no hardcoding to match output).

friends

name	age	year	animal
Jade	18	Freshman	Frog
Shuming	20	Junior	Prairie Dog
Jay	19	Sophomore	Capybara
Annie	19	Sophomore	Cat
Emma	20	Junior	Cat

animals

animal	sound	size	color
Frog	ribbit	tiny	green
Prairie Dog	bark	small	brown
Capybara	bark	large	brown
Cat	meow	small	black

(a) **(2.0 pt)** Write a query to output the `name` and `year` of all friends from `friends` that have "Frog" as their `animal`.

Intended output:

name	year
Jade	Freshman

```
SELECT name, year
FROM friends
WHERE animal = "Frog"
```

(b) **(3.0 pt)** Write a query to output the `animal` and the average `age` as `avg_age` of your friends for each `animal` and ordered by the `avg_age` descending. *Hint: Use AVG*

Intended output:

animal	avg_age
Prairie Dog	20
Cat	19.5
Capybara	19
Frog	18

```
SELECT animal, AVG(age) as avg_age
FROM friends
GROUP BY animal
ORDER BY avg_age DESC
```

(c) **(5.0 pt)** Write a query to output the `sound` and count of each `sound` corresponding to the `animal` of each friend in `friends`, ordered by the count ascending. *Hint: You will have to join the two tables and use COUNT(*).*

Intended output:

sound	COUNT
ribbit	1
bark	2
meow	2

```
SELECT a.sound, COUNT(*)
FROM friends as f, animals as a
WHERE f.animal = a.animal
GROUP BY a.sound
ORDER BY COUNT(*)
```

14. (0.0 points) End! (Optional)

(a) Draw your favorite staff member or tell us a joke!

No more questions.