

UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2017 EXAMINATIONS CSC148H1F

Duration: 3 hours
Instructors: Diane Horton and David Liu.
No Aids Allowed

Name:

Student Number:

Please read the following guidelines carefully.

- Please print your name and student number on the front of the exam.
 - This examination has **10** questions. There are a total of **22** pages, **DOUBLE-SIDED**.
 - You may always write helper functions/methods unless explicitly asked not to.
 - Docstrings are *not* required unless explicitly asked for.
 - You must earn a grade of **at least 40% on this exam to pass this course**.
-

Take a deep breath.

This is your chance to show us
How much you've learned.
We **WANT** to give you the credit
That you've earned.
A number does not define you.

It's been a real pleasure
teaching you this term.

Good luck!

Question	Grade	Out of
Q1		7
Q2		9
Q3		12
Q4		5
Q5		9
Q6		11
Q7		5
Q8		11
Q9		6
Q10		5
Total		80

1. [7 marks] Short answer questions. Point-form is acceptable; you do not need to write a lot for full marks.

(a) [1 mark] Suppose we have a function with the following header, and we know it does no input or output:

```
def mystery(number: int, bunch: tuple(str, float, str)) -> None:
```

It is impossible to write a body for this function that would make this function useful. Explain why.

(b) [2 marks] Here's the beginning of a method inside a class `MyClass`:

```
def silly(self, number: int) -> None:
    # Some code omitted
    this = that + number + self.thing
```

What must be true when we reach the assignment statement so that this method runs without raising an error?

(c) [4 marks] Consider this *incorrect* implementation of method `__getitem__` for class `LinkedList`:

```
class LinkedList:
    def __getitem__(self, index: int) -> object:
        """Return the item at position <index> in this list.
        Raise an IndexError if <index> is out of bounds."""
        curr = self._first
        curr_index = 0
        while curr is not None and curr_index < index:
            curr = curr.next
            curr_index += 1

        if curr_index == index:
            return curr.item
        else:
            raise IndexError
```

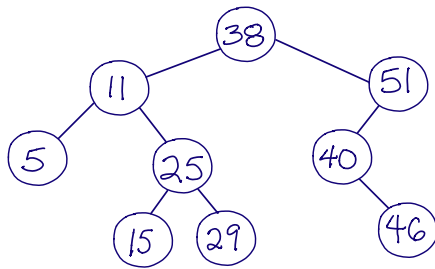
What do you know will be true immediately after the while loop? Express it as an executable assertion.

Give a doctest example that this method would fail.

Describe the problem with this method.

2. [9 marks] Short answer questions. Point-form is acceptable; you do not need to write a lot for full marks.

(a) [2 marks] Consider this tree:



Suppose we traverse the tree using post-order traversal, and we push each value onto a stack as we go. If we pop those values out of the stack, write down the items in the order they will come out.

(b) [2 marks] Explain the meaning of the term *composition*.

Give one example of composition we have used in this course (*e.g.*, from lecture, a lab, exercise, assignment).

(c) [2 marks] Explain the meaning of the term *abstract class*.

Give one example of an abstract class we have used in this course (*e.g.*, from lecture, a lab, exercise, assignment).

(d) [3 marks] How do we indicate that an instance attribute is private?

What restriction on client code does that imply?

Give two advantages of making an instance attribute private.

1.

2.

3. [12 marks] Suppose you are working on a replacement for ROSI/ACORN. One small part of it is keeping records about undergraduate and graduate students. They are similar, but have several differences:

- An undergrad needs a grade of at least 50 to earn credit for a course, but a graduate student needs at least 70.
- A grad student has a supervisor, as well as a committee of professors (including the supervisor) who help guide his or her research.

The following code (on the left), and its expected output (on the right), demonstrate some features that we need:

Code:

```
# Construct two undergrads and a grad student.
# They start out with no courses.
s1 = Undergrad('Jacqueline Smith', 1234)
s2 = Undergrad('Paul Gries', 2345)
# Jen's supervisor is named Zemel.
s3 = Grad('Jen Campbell', 5555, 'Zemel')

# Record that they completed some courses.
s1.complete_course('CSC108', 80)
s1.complete_course('BI0350', 40)
s1.complete_course('BI0350', 90) # Second try.
s3.complete_course('CSC2128', 80)
s3.complete_course('CSC3534', 60)

# Give Jen more committee members than just Zemel.
s3.add_committee_member('Urtasun')
s3.add_committee_member('Hinton')
s3.add_committee_member('Dickinson')

# Print everyone's basic info and their average.
# All grades count towards a student's average.
# Paul has completed no courses yet,
# so his average is reported as 0.0.
for s in [s1, s2, s3]:
    print(f'{str(s)}:')
    print(f'    {s.num_credits()} credits and ' +
          f'an average of {s.average()}')
# Print Jen's basic info and committee size.
print(f'{str(s3)} has {s3.committee_size()} members')
```

Output:

```
Jacqueline Smith (1234):
    2 credits and an average of 70.0
Paul Gries (2345):
    0 credits and an average of 0.0
Jen Campbell (5555):
    1 credits and an average of 70.0
Jen Campbell (5555) has 4 members
```

On the next two pages, write the **documentation** for the classes, including their methods, that are necessary for this code to run and produce the expected output. Your class design should use inheritance effectively to capture common attributes and methods.

Write excellent docstrings for classes and methods, but do not write doctests. **Do not spend time writing any method bodies**, except that if a method should be abstract, clearly mark it as such by raising the appropriate error in the method body. If a method overrides an inherited method, the docstring in the subclass method does not need to repeat information from the superclass method's docstring; it should just briefly explain why the method is being overridden.

Remember to include in each class:

- a class docstring describing instance attributes and their representation invariants
- type declarations for all attributes
- the header (including type contracts) and docstring for each public method in the class

Don't forget: Except for abstract methods, **do not spend time writing any method bodies**.

Space for your class design. (Continue your answer on the next page.)

Space for your class design.

4. [5 marks] Short answer questions. Point-form is acceptable; you do not need to write a lot for full marks.

- (a) [2 marks] Suppose we have a Python list with 1000 items, and a linked list with 1000 items. We want to insert a new item at position 500 in each list. Is doing this faster in the Python list, faster in the linked list, or roughly the same amount of time in each? Circle one:

faster in the Python list

faster in the linked list

roughly the same

Explain your answer.

- (b) [3 marks] Here is a function which copies a list. **Hint:** It is quite similar to mergesort.

```
def copy(lst: list) -> list:
    """Return a new list with the same elements as <lst>.

    >>> lst1 = [13, 4, 8, 7, 11, 25]
    >>> lst2 = copy(lst1)
    >>> lst2
    [13, 4, 8, 7, 11, 25]
    >>> lst2 is lst1
    False
    """
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left = copy(lst[:mid])
        right = copy(lst[mid:])
        for item in right:
            left.append(item)
        return left
```

What is the worst-case Big-Oh running time for this algorithm, in terms of n , the length of the input list?

Explain your answer.

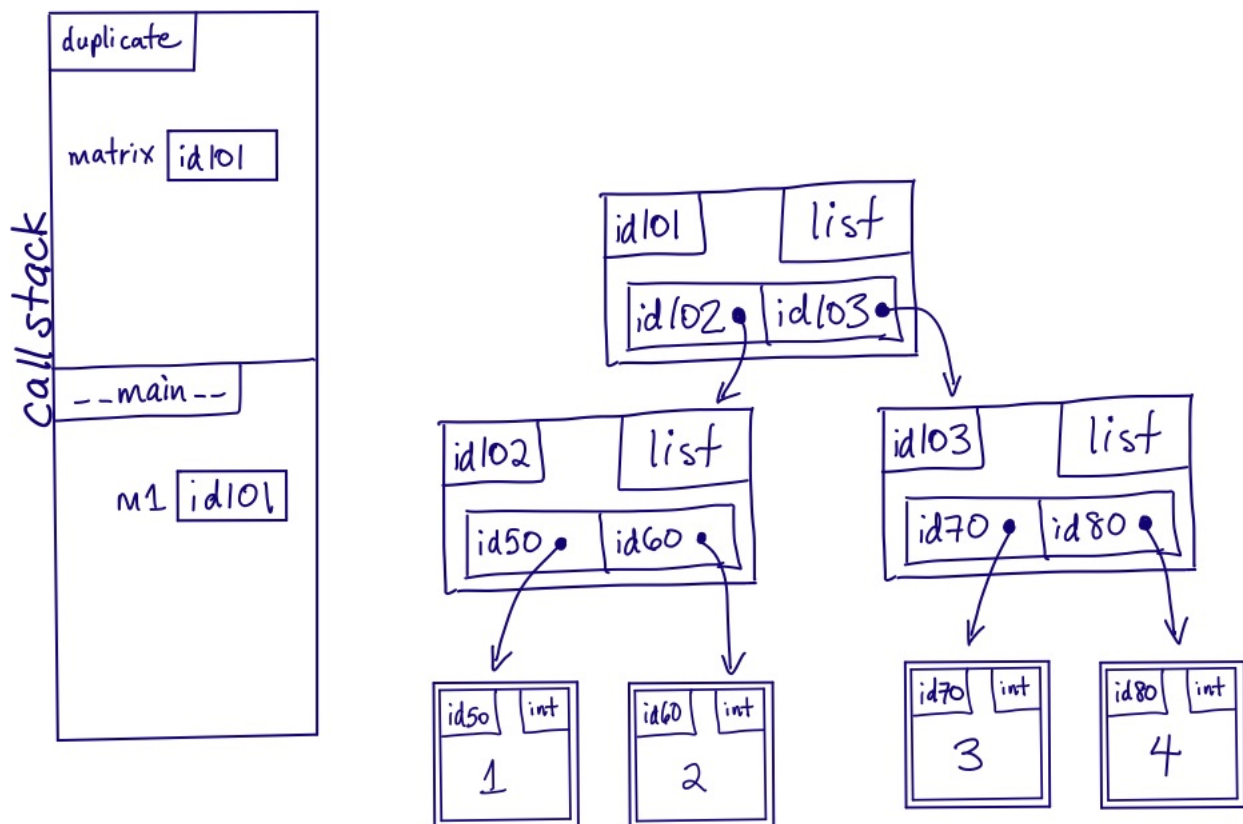
Why is the algorithm used by this implementation of `copy` a poor choice for solving the problem?

5. [9 marks] The code below defines and calls a function.

```
def duplicate(matrix: List[List[int]]) -> List[List[int]]:
    answer = []
    for item in matrix:
        answer.append(item)
    return answer

if __name__ == '__main__':
    m1 = [[1, 2], [3, 4]]
    m2 = duplicate(m1)
```

- (a) [2 marks] Below, we have drawn the state of memory when we call `duplicate(m1)`, right before the first line of the function body, `answer = []`, has been executed. Modify the diagram to show the state of memory after the function has returned and `m2` has been initialized. You may show references using arrows, id values, or both.



- (b) [2 marks] For each assertion below, circle YES or NO to indicate whether it would succeed if we ran it immediately after the line `m2 = duplicate(m1)`. Explain your answers.

`assert m1 == m2`

YES

NO

Explain:

`assert m1 is m2`

YES

NO

Explain:

- (c) [2 marks] Suppose that after `m2` is initialized, we change `m2` by running the line of code:

`m2[1][1] = 99`

For each assertion below, circle YES or NO to indicate whether it would succeed if we ran it immediately after running `m2[1][1] = 99`. Explain your answers.

`assert m2 == [[1, 2], [3, 99]]`

YES

NO

Explain:

`assert m1 == [[1, 2], [3, 4]]`

YES

NO

Explain:

- (d) [3 marks] Suppose we call `duplicate` on an input list of that contains `n` lists, where each inner list contains `m` items. Use Big-Oh notation to describe how long this function takes to run, in terms of `n` and/or `m`. Do not assume anything about the relationship between `n` and `m`.

Explain your answer.

6. [11 marks] Consider the following linked list method:

```
class LinkedList:
    def bisect(self, i: int) -> LinkedList:
        """Remove from this linked list the nodes at position i and beyond,
        and return them in a new linked list.

        The nodes in the new linked list should be in the same order as they
        were in <self>.

        Raise an IndexError if i < 0 or i >= the length of this linked list.

        Note: this is a mutating method because it modifies the contents
        of <self>.
        """
```

- (a) [3 marks] Write two doctests examples for this method, using `str` to display the values of the linked list. The examples must illustrate *different* cases for the method behaviour. We have started the first doctest for you.

Doctest 1

```
>>> linky1 = LinkedList([1, 2, 3, 4])
>>> str(linky1)
'[1 -> 2 -> 3 -> 4]'
```

Doctest 2:

- (b) [2 marks] You will need a `while` loop to iterate to the `i`-th node in the linked list (or to discover that `i` is out of bounds). Suppose we have a counter `index` that begins at 0, and a variable `curr` that begins by referencing the first node of the linked list.

Under what conditions should the loop stop, in terms of `index` and `curr`?

Under what conditions should the loop continue, in terms of `index` and `curr`?

(c) [6 marks] Implement the `bisect` method, including the `while` loop as described above.

You may not use any `LinkedList` or `_Node` methods other than their initializers; you may access all attributes (public or private) of both the `LinkedList` and `_Node` classes.

```
def bisect(self, i: int) -> LinkedList:
    """Remove from this linked list the nodes at position i and beyond,
    and return them in a new linked list.

    The nodes in the new linked list should be in the same order as they
    were in <self>.

    Raise an IndexError if i < 0 or i >= the length of this linked list.

    Note: this is a mutating method because it modifies the contents
    of <self>.
    """
```

7. [5 marks] Recall from Assignment 2 the definition of the Block class. We've provided the full docstring as a reference, but you won't need most of it for this question.

```
class Block:
    """A square block in the Blocky game.

    === Public Attributes ===
    position:
        The (x, y) coordinates of the upper left corner of this Block.
        Note that (0, 0) is the top left corner of the window.
    size:
        The height and width of this Block. Since all blocks are square,
        we needn't represent height and width separately.
    colour:
        If this block is not subdivided, <colour> stores its colour.
        Otherwise, <colour> is None and this block's subblocks store their
        individual colours.
    level:
        The level of this block within the overall block structure.
        The outermost block, corresponding to the root of the tree,
        is at level zero. If a block is at level i, its children are at
        level i+1.
    max_depth:
        The deepest level allowed in the overall block structure.
    highlighted:
        True iff the user has selected this block for action.
    children:
        The blocks into which this block is subdivided. The children are
        stored in this order: upper-right child, upper-left child,
        lower-left child, lower-right child.
    parent:
        The block that this block is directly within.

    === Representation Invariations ===
    - len(children) == 0 or len(children) == 4
    - If this Block has children,
      - their max_depth is the same as that of this Block,
      - their size is half that of this Block,
      - their level is one greater than that of this Block,
      - their position is determined by the position and size of this Block,
        as defined in the Assignment 2 handout, and
      - this Block's colour is None
    - If this Block has no children,
      - its colour is not None
    - level <= max_depth
    """
    position: Tuple[int, int]
    size: int
    colour: Optional[Tuple[int, int, int]]
    level: int
    max_depth: int
    highlighted: bool
    children: List['Block']
    parent: Optional['Block']
```

Implement the new `Block` method `colour_valid` according to its docstring. Your solution must be recursive; you may not use *any* `Block` methods, but you may access all `Block` attributes.

```
def colour_valid(self) -> bool:
    """Return whether this Block and all Blocks contained in this block
    satisfy these representation invariants:

        - If a Block has children, it has exactly four children and its own colour is None.
        - If a Block has no children, its colour is not None.

    Note: normally we assume that representation invariants are always satisfied.
    But the purpose of this method is to write an explicit check for these invariants,
    so don't simply write 'return True'.
    """
```

8. [11 marks] Please refer to the documentation for the `BinarySearchTree` class for this question.

(a) [3 marks] Here is the code for the `BinarySearchTree` method `height`.

```
1  def height(self) -> int:
2      """Return the height of this BST.
3      """
4      if self.is_empty():
5          return 0
6      else:
7          return 1 + max(self._left.height(), self._right.height())
```

What would happen if either `self._left` or `self._right` were `None` at line 7?

How do we know that neither of them can be `None` when we reach this line? Be specific.

(b) [2 marks] Suppose we have a binary search tree with n items and height h . What is the Big-Oh running time of this method, in terms of n and/or h ? Be as specific as possible, and explain your reasoning.

- (c) [6 marks] The `BinaryTree` class is the same as the `BinarySearchTree` class, except that its instances do not necessarily satisfy the BST property. You are given the following documentation for the class:

```
class BinaryTree:
    # === Private Attributes ===
    _root: Optional[object]
    _left: Optional['BinaryTree']
    _right: Optional['BinaryTree']

    def is_empty(self) -> bool:
        """Return whether this binary tree is empty."""

    def min_max(self) -> Tuple[int, int]:
        """Return the minimum and maximum value stored in this binary tree.
        The returned tuple contains (min value, max value), in that order.

        Precondition: this binary tree contains only integers.
        """
```

Implement the following `BinaryTree` method according to its docstring. You may access all `BinaryTree` attributes, but the only `BinaryTree` methods you may use are `is_empty` and `min_max` (assume they're implemented correctly already). Your solution must be recursive.



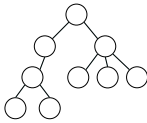
```
def is_bst(self) -> bool:
    """Return whether this binary tree is a binary *search* tree.

    An empty binary tree satisfies the binary search tree property.
    """
```

9. [6 marks] The following `Tree` method contains multiple errors.

```
class Tree:
    def num_at_or_below(self, depth: int) -> int:
        """Return the number of items in this tree at depth >= the given depth.
        The tree's root has depth 1.
        """
        sum = 0
        if depth <= 1:
            # Count the root
            sum = sum + 1
        for subtree in self._subtrees:
            # Count the relevant items for each subtree
            subtree.num_at_or_below(depth - 1)
        return sum
```

- (a) [2 marks] For each test case in the table below, write what the method *should* return (according to its docstring) for the given tree and value of `depth`. Note that we have drawn only the tree structure; the actual values stored in the tree do not affect the behaviour of this method.

Tree	Expected result for <code>depth = 1</code>	Expected result for <code>depth = 3</code>
empty tree	Test case 1	Test case 2
	Test case 3	Test case 4
	Test case 5	Test case 6
	Test case 7	Test case 8

- (b) [4 marks] Identify two different code errors in the given implementation of `num_at_or_below`, and for each one explain how it would cause one of the test cases from the previous part to have an incorrect *actual* result.

Problem 1:

Problem 2:

10. [5 marks] At McDonalds, it is possible to buy chicken nuggets (“McNuggets”) in packs of 4, 6, or 25. Consider the following function, which takes a number and returns whether it is possible to buy *exactly* that many chicken nuggets from McDonalds.

```
def buyable(n: int) -> bool:
    """Return whether one can buy exactly <n> McNuggets.

    It is considered possible to buy exactly 0 McNuggets.

    Precondition: n >= 0

    >>> buyable(6)
    True
    >>> buyable(35)
    True
    >>> buyable(5)
    False
    >>> buyable(13)
    False
    """
```

- (a) [1 mark] Suppose you know that it's possible to buy exactly 1231 McNuggets. Is it possible to buy 1235 McNuggets? How do you know?
- (b) [4 marks] Implement the `buyable` function. Your solution must be recursive, and you may not write any helper functions for this question.

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question*.

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question*.

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question*.

List methods

```
lst = [1, 2, 3]
len(lst)      # 3
lst[0]        # 1
lst[0:2]      # [1, 2]
lst[0] = 'howdy' # lst == ['howdy', 2, 3]
lst.append(29) # lst == ['howdy', 2, 3, 29]
lst.pop()     # lst == ['howdy', 2, 3], returns 29
lst.pop(1)    # lst == ['howdy', 3], returns 2
lst.insert(1, 100) # lst == ['howdy', 100, 3]
lst.extend([4, 5]) # lst == ['howdy', 100, 3, 4, 5]
3 in lst      # returns True
```

Dictionary methods

```
d = {'hi': 4, 'bye': 100}
d['hi']      # 4
d[100]       # raises KeyError!
'hi' in d    # True
4 in d       # False
d['howdy'] = 15 # adds new key-value pair
d['hi'] = -100 # changes a key-value pair
```

Classes

```
class Point:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

    def size(self) -> float:
        return (self.x ** 2 + self.y ** 2) ** 0.5

p = Point(3, 4) # initializer
p.x            # attribute access: returns 3
p.size()       # method call: returns 5.0

class MyWeirdClass(Point): # inheritance
    pass
```

Exceptions

```
class MyCustomError(Exception):
    pass

raise MyCustomError
```

Stack and Queues

```
s = Stack()
s.is_empty()
s.push(10)
s.pop() # Raises an EmptyStackError if the stack is empty.

q = Queue()
q.is_empty()
q.enqueue(10)
q.dequeue() # Returns None if the Queue is empty
```

Linked List

```
class _Node:
    """A node in a linked list.

    === Attributes ===
    item: The data stored in this node.
    next: The next node in the list, or None if there are
           no more nodes in the list.
    """
    item: object
    next: Optional['_Node']

    def __init__(self, item: object) -> None:
        """Initialize a new node storing <item>,
        with no 'next' node.
        """

class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # === Private Attributes ===
    # _first:
    #     The first node in the linked list,
    #     or None if the list is empty.
    _first: Optional['_Node']

    def __init__(self, items: list) -> None:
        """Initialize a linked list with the given items.

        The first node in the linked list contains the
        first item in <items>.
        """
```

Tree

```
class Tree:
    # === Private Attributes ===
    # The item stored at this tree's root, or None if the tree is empty.
    _root: Optional[object]
    # The list of all subtrees of this tree.
    _subtrees: List['Tree']

    # === Representation Invariants ===
    # - If self._root is None then self._subtrees is an empty list.
    #   This setting of attributes represents an empty Tree.
    # - self._subtrees may be empty when self._root is not None.
    #   This setting of attributes represents a tree consisting of just one
    #   node.

    def __init__(self, root: object, subtrees: List['Tree']) -> None:
        """Initialize a new Tree with the given root value and subtrees.

        If <root> is None, the tree is empty.
        Precondition: if <root> is None, then <subtrees> is empty.
        """

    def is_empty(self) -> bool:
        """Return True if this tree is empty."""
```

BinarySearchTree

```
class BinarySearchTree:
    # === Private Attributes ===
    # The item stored at the root of the tree, or None if the tree is empty.
    _root: Optional[object]
    # The left subtree, or None if the tree is empty
    _left: Optional['BinarySearchTree']
    # The right subtree, or None if the tree is empty
    _right: Optional['BinarySearchTree']

    # === Representation Invariants ===
    # - If _root is None, then so are _left and _right.
    #   This represents an empty BST.
    # - If _root is not None, then _left and _right are BinarySearchTrees.
    # - (BST Property) All items in _left are <= _root,
    #   and all items in _right are >= _root.

    def __init__(self, root: Optional[object]) -> None:
        """Initialize a new BST with the given root value and no children.

        If <root> is None, make an empty tree, with subtrees that are None.
        If <root> is not None, make a tree with subtrees are empty trees.
        """

    def is_empty(self) -> bool:
        """Return True if this BST is empty."""
```
