FINAL EXAM REVIEW Solutions

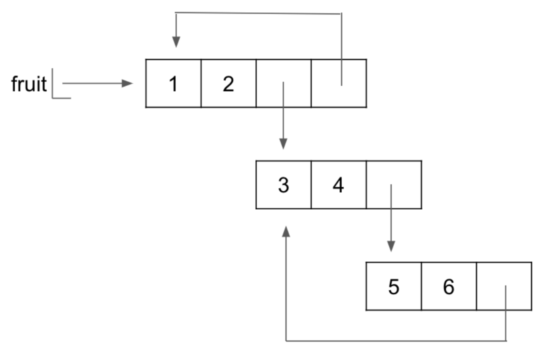
COMPUTER SCIENCE MENTORS 61A

December 9 – December 13, 2024

1 Environment Diagrams

1. Fill in each blank in the code example below so that its environment diagram is the following. You do not need to use all the blanks.

```
fruit = [1, 2, [3, 4]]
fruit.____________________________
fruit[3][2].______________________
fruit[2][2].______________________
fruit[3][3][2][2][2][1] = ____
```




```
fruit = [1, 2, [3, 4]]
fruit.append(fruit)
fruit[3][2].append([5, 6])
fruit[2][2].append(fruit[2])
fruit[3][3][2][2][2][1] = 4
```

2 Iterators

2. Define a **non-decreasing path** as a path from the root where each node's label is greater than or equal to the previous node along the path. A **subpath** is a path between nodes X and Y, where Y must be a descendent of X (ex: Y is a branch of a branch of X).

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

(a) Write a generator function `root_to_leaf` that takes in a tree `t` and yields all non-decreasing paths from the root to a leaf node, in any order. Assume that `t` has at least one node.

```
def root_to_leaf(t):
    """
    >>> t1 = Tree(3, [Tree(5), Tree(4)])
    >>> list(root_to_leaf(t1))
    [[3, 5], [3, 4]]
    >>> t2 = Tree(5, [Tree(2, [Tree(7), Tree(8)]), Tree(5, [Tree(6)])])
    [[5, 5, 6]]
    """

    if ____________________:

        _________________________

    for _________________________:

        if __________________________:

            for __________________________:

                ______________________________
```

```
def root_to_leaf(t):
    if t.is_leaf():
        yield [t.label]
    for b in t.branches:
        if t.label <= b.label:
            for path in root_to_leaf(b):
                yield [t.label] + path
```

The easiest way to approach this is to notice the two blocks of code that are provided: first an if statement, probably referring to a base case, and a for loop, which will probably be the recursive case. From the doctests, we can see that giving the function a tree that just has one node, or in other words `is_leaf()`, returns a list containing just that node.

In our recursive case we want to do two things. First, we want to check if the next branch value really is non-decreasing. Then, if it is, we want to append the result of calling `root_to_leaf` on the branch to the value of our current tree to create a complete path. So we recurse through each of the branches in `t` (`for b in t.branches`), then check if it is nondecreasing (`t.label <= b.label`), then yield our tree's label appended to the recursive call (the last two lines).

(b) Write a generator function `subpaths` that takes in a tree `t` and yields all non-decreasing subpaths that end with a leaf node, in any order. You may use the `root_to_leaf` function above, and assume again that `t` has at least one node.

```
def subpaths(t):

    yield from ____________________

    for b in t.branches:

        __________________________
```

```
def subpaths(t):
    yield from root_to_leaf(t)
    for b in t.branches:
        yield from subpaths(b)
```

We can split this problem into two steps – yielding all subpaths for the current tree that we have, then yielding all subpaths for all other trees within this tree. It is important to realize that each node in the tree is merely a subtree of the original tree to solve this problem.

To yield all non-decreasing subpaths for our current tree (that is all non-decreasing subpaths that start at our current node and end at the leaf nodes), we can just yield from our previous function, `root_to_leaf`, called on that node. For the rest of the subpaths, we want to recursively call `subpaths` on all our child nodes. This will give us all paths that end on the leaf nodes (because `root_to_leaf` ends on the leaf nodes) that start from any child on this tree. It is important to realize that the base case in this situation is implicit. If a leaf node is passed in and reaches the for loop, the for loop finds no items in `t.branches`, and will just terminate without calling the clause inside.

3 Data Abstraction

3. In the following problem, we will represent a bookshelf object using dictionaries.

   In the first section, we will set up the format. Here, we will directly work with the internals of the Bookshelf, so don't worry about abstraction barriers for now. Fill in the following functions based on their descriptions (the constructor is given to you):

```
def Bookshelf(capacity):
    """ Creates an empty bookshelf with a certain max capacity. """
    return {'size': capacity, 'books': {}}

def add_book(bookshelf, author, title):
    """
    Adds a book to the bookshelf. If the bookshelf is full,
    print "Bookshelf is full!" and do not add the book.
    >>> books = Bookshelf(2)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Daniel Kleppner', 'An Introduction to Mechanics
       5th Edition')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    Bookshelf is full!
    """
    if ________________________________:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            _____________________________________
        else:
            _____________________________________
```

```
    if len(bookshelf['books']) == bookshelf['size']:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            bookshelf['books'][author].append(title)
        else:
            bookshelf['books'][author] = [title]
```

```
def get_all_authors(bookshelf):
    """
    Returns a list of all authors who have at least one book in the
       bookshelf.
    >>> books = Bookshelf(10)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Sheldon Axler', 'Linear Algebra Done Right')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    >>> get_all_authors(books)
    ['Jane Austen', 'Sheldon Axler', 'Kurt Vonnegut']
    """
    return ____________________________________


        return list(bookshelf['books'].keys())
```

Now, complete the function `most_popular_author` **without breaking the abstraction barrier**. In other words, you are not allowed to assume anything about the implementation of a Bookshelf object, or use the fact that it is a dictionary. You can only use the methods above and their stated return values.

```
def most_popular_author(bookshelf):
    """
    Returns the author with the greatest number of books on this bookshelf.
    You can assume that the bookshelf is not empty.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', 'Xenocide')
    >>> add_book(books, 'Orson Scott Card', 'Children of the Mind')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> most_popular_author(bookshelf)
    'Orson Scott Card'
    """
    return max(__________________________________________________,

        key=_________________________________________________________)


        return max(get_all_authors(bookshelf), key=lambda x:
           len(get_author_books(x)))
```

4. Find the $\Theta(\cdot)$ runtime bound for `hiya(n)`. Remember that Python strings are immutable: when we add two strings together, we need to make a copy.

```
def hiii(m):
    word = "h"
    for i in range(m):
        word += "i"
    return word

def hiya(n):
    i = 1
    while i < n:
        print(hiii(i))
        i *= 2
```

$\Theta(n^2)$.

Solution: We can determine the efficiency by approximately counting the number of characters we have to store upon a call to `hiya(n)`. First, let us determine the efficiency of a call `hiii(m)`. Within `hiii`'s for loop:

- When `i` is `1`, we store the string "hi", which is 2 characters.
- When `i` is `2`, we store the string "hii", which is 3 characters.

  ...
- When `i` is `m`, we store `m + 1` characters.

Adding up these values, we see that calling `hiii(m)` causes us to store on the order of m^2 characters. (The exact value is $\frac{m(m+3)}{2} = \frac{m^2}{2} + \frac{3}{2}m$, but we really only care about the highest order term.)

Now, when we make a call `hiya(n)`, we will make calls to `hiii(1)`, `hiii(2)`, `hiii(4)`, ..., `hiii(4)`. This will store approximately $1^2 + 2^2 + 4^2 + 8^2 + ... + n^2$ characters. Calculating out the partial sums of this sequence shows that

$$1^2 = 1$$

$$1^2 + 2^2 = 5 < 2 \cdot 2^2$$

$$1^2 + 2^2 + 4^2 = 21 < 2 \cdot 4^2$$

$$1^2 + 2^2 + 4^2 + 8^2 = 85 < 2 \cdot 8^2$$

At some point, we are reasonably convinced that this pattern holds. Thus the value of $1^2 + 2^2 + 4^2 + 8^2 + ... + n^2$ is approximately n^2, within a constant factor. So we store about n^2 characters upon a call to `hiya(n)`, which means the efficiency is $\Theta(n^2)$.

5 OOP

Let's use OOP design to help us create a supermarket chain (think Costco)! There are many different ways to implement such a system, so there is no concrete answer.

5. What classes should we consider having? How should each of these classes interact with each other?

   There are many ways of approaching this, but one way is to have a Supermarket class to represent the entire store, an Item class to represent a certain item, a Food class to represent an item that is a food (inherits from Item), and maybe a Customer class to represent someone buying items from that store.

6. For each class, what instance and class variables would it have?

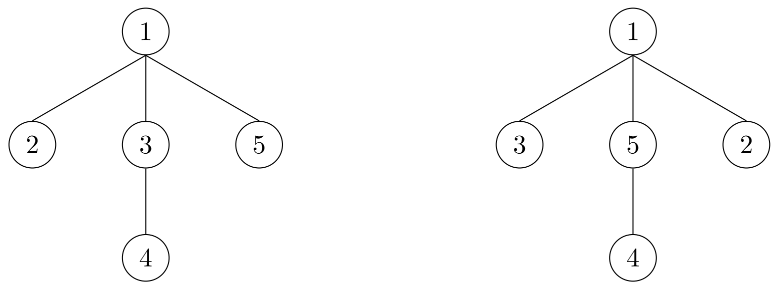
   1. Supermarket – we might have instance variables such as profit, store name, location, and a list of the items in that store along with their quantity. Note that we prefer to store the quantity inside the Supermarket, since an Item might belong to multiple Supermarkets, and each Supermarket will have a separate quantity. We might even have a price associated with each item, since specific supermarkets may mark up prices in different areas.
   2. Item – we might have instance variables such as the name and the base price.
   3. Food – we will have it inherit of all the instance variables of the Item, and also whether it is yummy, maybe the food group it is in or the expiration date.
   4. Customer – we might have some personal information, the supermarket that they're buying from, and the history of their
   5. There are some details that have been missed as well! For example, not just food items expire. Feel free to just discuss this.

7. For each class, what class methods would they have? How would they interact with each other?

   1. Once again, these are just suggestions:
   2. Supermarket
      - `check_quantity(Item)`: looks up the available quantity of that item
      - `checkout_items(Customer)`: returns the total sum of items in a customer's shopping cart, and clears their shopping cart
   3. Item
      - `check_quantity(Supermarket)`: calls `supermarket.check_quantity(self)`
   4. Food
      - `time_to_expire()`: returns an integer representing how many days before this item expires
      - `is_yummy()`: returns a boolean value of whether this item is yummy or not!
   5. Customer
      - `enter(Supermarket)`: create a shopping cart for customer in this supermarket, if it doesn't already exist
      - `leave(Supermarket)`: clear customer's shopping cart
      - `buy_item(Item)`: add item to customer's shopping cart
      - `checkout_items()`: calls `supermarket.checkout_items(Customer)`

8. Implement `rotate`, which takes in a tree and rotates the labels at each level of the tree by one to the left destructively. This rotation should be modular (That is, the leftmost label at a level will become the rightmost label after running rotate). You do NOT need to rotate across different branches.

   For example, given tree `t` on the left, `rotate(t)` should mutate `t` to give us the right.



```
def rotate(t):
  """
  >>> t1 = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(5)])
  >>> rotate(t1)
  >>> t1
  Tree(1, [Tree(3), Tree(5, [Tree(4)]), Tree(2)])
  >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]),
                    Tree(5, [Tree(6)])])
  >>> rotate(t2)
  >>> t2
  Tree(1, [Tree(5, [Tree(4), Tree(3)]),
                    Tree(2, [Tree(6)])])
  """
  branch_labels = ________________________________________

  n = len(t.branches)

  for ___________________________________________:

    _____________________________________________________

    _____________________________________________________

    _____________________________________________________
```

```python
def rotate(t):
  branch_labels = [b.label for b in t.branches]
  n = len(t.branches)
  for i in range(n):
    branch = t.branches[i]
    branch.label = branch_labels[(i + 1) % n]
    rotate(branch)
```

7 Scheme Lists

9. Star-Lord is cruising through space and can't afford to crash into any asteroids along the way. Let his path be represented as a (possibly nested) list of integers, where an asteroid is denoted with a 0, and stars and planets otherwise. Every time Star-lord sees (visits) an asteroid (0), he merges the next planet/star with the asteroid. In other words, construct a NEW list so that all asteroids (0s) are replaced with a list containing the planet followed by the asteroid (e.g. (planet 0)). You can assume that the last object in the path is not an asteroid (0).

```scheme
;Doctests
scm> (collision (list 1 2 3 0 4))
(1 2 3 (4 0))
scm> (collision (list 4 3 (list 0 1) 2))
(4 3 ((1 0)) 2)
scm> (collision (list 1 -2 0 -3 4 0 -5 6))
(1 -2 (-3 0) 4 (-5 0) 6)
scm> (collision (list 1 0 0 2 3))
(1 (0 0) 2 3)

;Asteroids can merge with other asteroids too

(define (collision lst)

  (cond ((_____________________) lst)

    ((_________________________)

     _______________________________)

    ((_________________________)

      (cons __________________________

        _______________________________))

    (else ______________________________)
  )
)
```

```
(define (collision lst)
  (cond ((null? lst) nil)
    ((list? (car lst))
      (cons (collision (car lst)) (collision (cdr lst))))
    ((and (equal? (car lst) 0) (not (null? (cdr lst))))
      (cons (list (car (cdr lst)) (car lst))
        (collision (cdr (cdr lst)))))
    (else(cons (car lst) (collision (cdr lst))))
  )
)

#Alternate solution (No cond form)

(define (collision lst)
  (if (null? lst)
    lst
    (if (list? (car lst))
      (cons (collision (car lst)) (collision (cdr lst)))
      (if (equal? (car lst) 0)
        (cons (list (cadr list) (car lst)) (collision (cddr lst)))
        (cons (car lst) (collision (cdr lst)))
      )
    )
  )
)
```