LINKED LISTS, MUTABLE TREES AND MIDTERM REVIEW Meta

COMPUTER SCIENCE MENTORS 61A

October 28 – November 1, 2024

Recommended Timeline

- Linked Lists Mini-Lecture: 5 min
- Linked Lists Problems
  - WWPD - Easy
  - Combine Two - Medium
  - Middle Node - Medium
  - Insert At - Hard
- Trees (Class) Mini-Lecture / Overview: 5 min
- Trees (Class) Problems
  - Delete Path Duplicates - Medium
  - Replace Leaves Sum - Medium
  - Tulip Mania - Medium
  - Contains N - Hard
- Higher Order Functions Problems
  - Make Digit Remover - Medium
  - Compound - Hard
- Lists / Mutable Lists Problems
  - Incredibles - Medium
  - Duplicate List - Medium
  - Add Up - Medium
- Iterators & Generators
  - Num Elems - Medium

This is a very long worksheet – It essentially acts as a MT2 review problem bank. The topics will also not only be limited to Linked Lists & Mutable Trees. Feel free to ignore the bulk of these problems during your section. This worksheet is more of a resource for the student than teaching material (tbh).

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

1 Linked Lists

For each of the following problems, assume linked lists are defined as follows:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Linked lists are a recursive data structure for representing sequences. They consist of a series of "links," each of which has two attributes: `first` and `rest`. The `first` attribute contains the value of the link (which can hold any type of data, even another linked list!). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just a "None" type value.

For example, `Link(1, Link(2, Link(3)))` is a linked list representation of the sequence $1, 2, 3$.

Like trees, linked lists naturally lend themselves to recursive problem solving. Consider the following example, in which we double every value in linked list. We double the value of the current link and then recursively double the rest.

```
def double_values(link):
     if link is not Link.empty:
        link.first *= 2 # we mutate the value inside of the link
        double_val(link.rest) # we mutate the values in the rest
                              # of the linked list
    # if the link is empty then do nothing
```

Teaching Tips

- Try to draw box and pointer diagrams.
- Make clear that the pointer *points* to a linked list if we have nested linked lists.
- Try to experiment with going over various ways to mutate and create linked lists.
- We have a great visualizer on https://code.cs61a.org/ where you can call draw(lst) to visualize a list!
- Try using PythonTutor as well!

1. What will Python output? Draw box-and-pointer diagrams along the way.

```
>>> a = Link(1, Link(2, Link(3)))
```

```
+---+---+   +---+---+   +---+---+
| 1 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

```
>>> a.first
```

```
1
```

```
>>> a.first = 5
```

```
+---+---+   +---+---+   +---+---+
| 5 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

```
>>> a.first
```

5

```
>>> a.rest.first
```

2

```
>>> a.rest.rest.rest.rest.first
```

Error: tuple object has no attribute rest (Link.empty has no rest)

```
>>> a.rest.rest.rest = a
```

```
   +---+---+   +---+---+   +---+---+
+->| 5 | --|->| 2 | --|->| 3 | --|--+
|  +---+---+   +---+---+   +---+---+  |
|                                     |
+-------------------------------------+
```

```
>>> a.rest.rest.rest.rest.first
```

```
2
```

```
>>> repr(Link(1, Link(2, Link(3, Link.empty))))
```

```
"Link(1, Link(2, Link(3)))"
```

```
>>> Link(1, Link(2, Link(3, Link.empty)))
```

```
Link(1, Link(2, Link(3)))
```

```
>>> str(Link(1, Link(2, Link(3))))
```

```
'<1 2 3>'
```

```
>>> print(Link(Link(1), Link(2, Link(3))))
```

```
<<1> 2 3>
```

Teaching Tips

- For assignment statements, Python will not print anything but still have them draw out what the linked list will look like
- Note that we are doing mutation here, so we are actually altering the object that was created in the first assignment.
  - Some students may have minimal exposure to mutating objects so try to emphasize this and make it obvious through diagrams.
- For the error, walk-through how to keep track of which rest corresponds to which object in the box and pointer diagram. **Make sure they understand why calling rest a fourth time will give us an error (look back at the class definition)**
  - Abstraction:
    * our last .rest is set to Link.empty
    * Link.empty is not a Link objects — they do not have a .rest attribute

- Actual implementation:
    * our last .rest is set to Link.empty
    * Link.empty is not a Link objects — they do not have a .rest attribute
- Reassigning the last .rest to point back at the front always trips students up.
  - Make it clear that a is a pointer that points to the linked list. So we are trying to assign the last rest of a to point at what a points to, which is the beginning of the list. **To test their understanding ask what would be different if we instead had**:
    * a.rest.rest.rest = a.rest
  - a way to explain the assignment for this problem is to emphasize the "evaluation" of the RHS and the LHS
  - what is the value of a (a pointer). Really emphasize the implications of pointers here.
  - where are we putting a into? (the box that represents a.rest.rest.rest)
  - same for a.rest.rest.rest = a.rest. what is the value of a.rest? (still a pointer!)
  - Mention that this creates a cycle in the list

2. Write a function `combine_two`, which takes in a linked list of integers `lnk` and a two-argument function `fn`. It returns a new linked list where every two elements of `lnk` have been combined using `fn`.

```
def combine_two(lnk, fn):
    """
    >>> lnk1 = Link(1, Link(2, Link(3, Link(4))))
    >>> combine_two(lnk1, add)
    Link(3, Link(7))
    >>> lnk2 = Link(2, Link(4, Link(6)))
    >>> combine_two(lnk2, mul)
    Link(8, Link(6))
    """
    if ______________________________________:

        return ______________________________

    elif ____________________________________

        return ______________________________

    combined = ______________________________

    return __________________________________
```

```
def combine_two(lnk, fn):
    if lnk is Link.empty:
        return Link.empty
    elif lnk.rest is Link.empty:
        return Link(lnk.first)
    combined = fn(lnk.first, lnk.rest.first)
    return Link(combined, combine_two(lnk.rest.rest, fn))
```

3. Write a function `middle_node` that takes as input a linked list `lst`. `middle_node` should return the middle node of the linked list. If there are two middle nodes, return the second middle node.

```
def middle_node(lst):
    """
    >>> head = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> middle_node(head)
    Link(3, Link(4, Link(5))) # The middle node of the list is node 3
    >>> head = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
    Link(4, Link(5, Link(6))) # Since the list has two middle nodes with
       values 3 and 4, we return the second one
    """
    list_iter, middle = _______________, ________________

    length = _________________________________

    while _________________________________:

        length = _________________________________

        list_iter = _________________________________

    for _________________________________:

        middle = _________________________________

    if length % 2 == 1:

        middle = _________________________________

    return middle
```

Challenge version **(Optional)**:

```
def middle_node(lst):

    list_iter, middle = _______________, ________________

    while __________________ and __________________:

        list_iter = _________________________________

        middle = _________________________________

    return middle
```

```python
def middle_node(lst):
    list_iter, middle = lst, lst
    length = 0

    while list_iter:
        length = length + 1
        list_iter = list_iter.rest

    for i in range(length // 2):
        middle = middle.rest

    if length % 2 == 1:
        middle = middle.rest

    return middle
```

In this solution, we first calculate the length of the linked list, and then finding the middle node based on that length.

Challenge version

```python
def middle_node(lst):
    list_iter, middle = lst, lst

    while list_iter and list_iter.rest:
        list_iter = list_iter.rest.rest
        middle = list_iter.rest

    return middle
```

In this solution, we iterate through the linked list with two pointers at different speeds. One pointer, `list_iter`, moves through the list one node at a time, while the other pointer, `middle`, moves through the list at half the speed of `list_iter`.

Emphasize drawing out this problem with a box-and-pointer diagram, illustrating the different steps of our function in a slow and digestible way if students are confused as to functionality. Illustrate how the function works for the doctests.

If students find it easier to think about solving and logic-ing the problem without the skeleton, feel free to tell students to disregard it.

4. Write a recursive function `insert_all` that takes as input two linked lists, `s` and `x`, and an index `index`. `insert_all` should return a new linked list with the contents of `x` inserted at index `index` of `s`.

```
def insert_all(s, x, index):
    """
    >>> insert = Link(3, Link(4))
    >>> original = Link(1, Link(2, Link(5)))
    >>> insert_all(original, insert, 2)
    Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> start = Link(1)
    >>> insert_all(original, start, 0)
    Link(1, Link(1, Link(2, Link(5))))
    """
    if ___________________ and ___________________:
        _____________________________________________
    if ___________________ and ___________________:
        _____________________________________________
    _________________________________________________
```

```
def insert_all(s, x, index):
    """
    >>> insert = Link(3, Link(4))
    >>> original = Link(1, Link(2, Link(5)))
    >>> insert_all(original, insert, 2)
    Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> start = Link(1)
    >>> insert_all(original, start, 0)
    Link(1, Link(1, Link(2, Link(5))))
    >>> insert_all(original, insert, 3)
    Link(1, Link(2, Link(5, Link(3, Link(4)))))
    """
    if s is Link.empty and x is Link.empty:
        return Link.empty
    if x is not Link.empty and index == 0:
        return Link(x.first, insert_all(s, x.rest, 0))
    return Link(s.first, insert_all(s.rest, x, index - 1))
```

All of our return statements should return a new linked list.

Our base case should be the simplest possible version of the problem: when both `x` and `s` are empty, clearly the result is just the empty list.

We can now think of ways to break down this problem even further. Note that when the index to be inserted at is `0`, the problem is relatively easy: we just have to put all of the elements of `x` followed by all the elements of `s`. So the first element of the new list should `x.first`, and the rest of the new list should be `x.rest` concatenated with `s`, or `insert_all(s, x.rest, 0)`. Since we are using `x.first` and `x.rest`, we must check that `x` is nonempty to ensure that we do not error.

Finally, when the index to be inserted at is nonzero, we know that we're going to have some elements of `s`, then the elements of `x`, and then the rest of the elements from `s`. So the first element of the new list should be `s.first`. Then we can get the rest of the new list by inserting the contents of `x` at index `index - 1` of `s.rest`, reducing the index by 1 to account for the fact that we have removed the first element of `s`.

There's one issue we glossed over here: what if `x` is empty but `s` is not? Then we want to return the contents of `s`. But because the problem requires that we return a new linked list, we must recursively reconstruct `s` instead of simply returning it. You could add another base case to handle this, but as it turns out the second recursive case will handle this just fine since `Link(s.first, insert_all(s.rest, x, index - 1))` is just equivalent to `Link(s.first, s.rest)` when `x` is empty. Since the `x is not Link.empty` condition for the first recursive case will direct all situations where `x` is empty but `s` is not to the second recursive case, it turns out that we do not need to add anything else to this solution.

Convincing yourself that this problem works requires that you eventually reach a base case. Note that in either recursive call, we either reduce `s` or `x` by one element. So the base case will always eventually be reached, and the solution is valid.

Despite being just a few lines and exercising a familiar concepts with lists, I've found that this problem is quite difficult, so one thing I would emphasize is to draw out this problem with a box-and-pointer diagram and illustrate the different steps of our function. Illustrate how the function works for the doctests, which should cover all possible cases of inserting a new linked list into the beginning, middle, and end of the original linked list

If students are lost, which they most likely will be, here are some leading questions you could ask:

- When do we know that we are done inserting items into the list?
- What should the parameters be equal to if we are going to start inserting `x`, what if we are not currently inserting `x`?
- How do ensure to add all elements of `x` into `x`?

You should tell your students that they should feel free to disregard the provided skeleton, because it is quite difficult to think of the solution to this problem when you are trying to fit everything into the skeleton.

2 Trees

For the following problems, use this definition for the `Tree` class:

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return self.branches == []

    # Implementation ommitted
```

Here are a few key differences between the `Tree` class and the Tree abstract data type, which we have previously encountered:

- Using the constructor: Capital T for the `Tree` class and lowercase t for tree ADT `t = Tree(1)` vs. `t = tree(1)`
- In the class, `label` and `branches` are instance variables and `is_leaf()` is an instance method. In the ADT, all of these were globally defined functions.

  `t.label` vs. `label(t)`

  `t.branches` vs. `branches(t)`

  `t.is_leaf()` vs. `is_leaf(t)`
- A `Tree` object is mutable while the tree ADT is not mutable. This means we can change attributes of a `Tree` instance without making a new tree. In other words, we can solve tree class problems non-destructively and destructively, but can only solve tree ADT problems non-destructively.

  `t.label = 2` is allowed but `label(t)= 2` would error.

Apart from these differences, we can take the same general approaches we used for the tree ADT and apply them to the `Tree` class!

Feel free to not spend too much time on this section! Your students already covered immutable trees when practicing ADTs.

1. Define `delete_path_duplicates`, which takes in `t`, a tree with non-negative labels. If there are any duplicate labels on any path from root to leaf, the function should mutate the label of the occurrences deeper in the tree (i.e. farther from the root) to be the value `-1`.

```
def delete_path_duplicates(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(1), Tree(1)])])
    >>> delete_path_duplicates(t)
    >>> t
    Tree(1, [Tree(2, [Tree(-1), Tree(-1)])])
    >>> t2 = Tree(1, [Tree(2), Tree(2, [Tree(2, [Tree(1, [Tree(5)])])])])
    >>> delete_path_duplicates(t2)
    >>> t2
    Tree(1, [Tree(2), Tree(2, [Tree(-1, [Tree(-1, [Tree(5)])])])])
    """
    def helper(_______________, _______________):

        if ________________________________:

            ________________________________

        else:

            ________________________________

        for _______ in ____________________:

            ________________________________

    ________________________________________
```

```
    def helper(t, seen_so_far):
        if t.label in seen_so_far:
          t.label = -1
        else:
            seen_so_far = seen_so_far + [t.label]
        for b in t.branches:
            helper(b, seen_so_far)
    return helper(t, [])
```

Teaching Tips

- To clarify, the problem is asking to delete *path* duplicates, and not *tree* duplicates. As illustrated in the last doctest, it is acceptable to keep two identical labels if they appear on different branches.
- Draw out the doctest Tree and walk through how you would delete path duplicates by hand. Then, ask your students, "how would we write this in code?"
- Recap with your students the core properties for trees such as label and branches.
- We don't need to use the is_leaf() function because our for loop will not run if there are no branches (which only occurs if the tree is a leaf). But, you can write in this base case to start with.

- Make sure to point out the reason why we can't use `seen_so_far.append(t.label)` in the else case. (The reason is that we need to create a new list in each frame, rather than mutating the same one. If append is used, `seen_so_far` would contain everything seen in the tree so far, not just the current branch.)

2. Given a tree `t`, mutate the tree so that each leaf's label becomes the sum of the labels of all nodes in the path from the leaf node to the root node.

```
def replace_leaves_sum(t):
    """
    >>> t = Tree(1, [Tree(3, [Tree(2), Tree(8)]), Tree(5)])
    >>> replace_leaves_sum(t)
    >>> t
    Tree(1, [Tree(3, [Tree(6), Tree(12)]), Tree(6)])
    """
    def helper(______________ , _________________):

        if t.is_leaf():

            _____________________________________

        for b in t.branches:

            _________________________________

    _____________________________________________
```

```
def replace_leaves_sum(t):
    def helper(t, total):
        if t.is_leaf():
            t.label = total + t.label
        else:
            for b in t.branches:
                helper(b, total + t.label)
    helper(t, 0)
```

3. Write a function that returns `True` if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

   Hint: recall that the built-in function ***any*** *takes in an iterable and returns* `True` *if any of the iterable's elements are truthy.*

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ____________________________________________:

        return ______________________________________

    elif ____________________________________________:

        return ______________________________________

    else:

        return ______________________________________
```

```
    if n == 0:
        return True
    elif t.is_leaf():
        return n == 1 and t.label == elem
    elif t.label == elem:
        return any([contains_n(elem, n - 1, b) for b in
          t.branches])
    else:
        return any([contains_n(elem, n, b) for b in
          t.branches])
```

Base cases: The simplest case we have is when `n == 0`, or when we want at least 0 instances of `elem` in `t`. In this case, we always return `True`. The other simple case we consider is when the tree is only a leaf — there is nothing left to recurse on. In that case, we simply check to see that both `n == 1` and that `t.label == elem`, meaning that we have one element left to satisfy, and the leaf label satisfies the final element we are looking for. If we have more elements to search for (ie. n ¿ 1), then we will not satisfy that many elements at the leaf node; conversely, if we have fewer (ie. `n == 0`), then the case would already be covered by the first base case.

Recursive cases: If the current node isn't a leaf, then there's two different cases we should consider. Either the label of the current node is equal to `elem` or the label is not equal to `elem`. For the former, we would have to search for `n` more `elems` in each branch of `t` and return `True` if any of the branches contain `n` elems. For the latter, we would have `(n - 1)` elements remaining, so we would search for `(n - 1)` more `elems` in each branch of `t` and return `True` if any of the branches contain `(n - 1) elems`. Since there is not room to do a for loop, we can use a list comprehension to recursively call the function on each branch. Thus, our two list comprehension statements would be `[contains_n(elem, n, b) for b in t.branches]` and `[contains_n(elem, n - 1, b) for b in t.branches]`. To determine if any of the branches contain either `n` elems or `(n - 1)` elems, we can check if there's a `True` element in the respective lists.

Teaching Tips

- To understand what the function is doing, try walking through a test case on `t2`.
- Ask students if there are any more base cases (lot of **`elif`** statements).
- If we are at some node with label `elem`, what does this mean about the amount of instances of `elem` we have yet to search for?
- Where could we find more instances of `elem`? The branches; so we should recurse on them.
- Remind students about the **`any`** and **`all`** functions for dealing with lists of booleans.

3 Higher Order Functions

1. Write a function, `make_digit_remover`, which takes in a single digit `i`. It returns another function that takes in an integer and, scanning from right to left, removes all digits from the integer up to and including the first occurrence of `i`, starting from the ones place. If `i` does not occur in the integer, the original number is returned.

```
def make_digit_remover(i):
    """
    >>> remove_two = make_digit_remover(2)
    >>> remove_two(232018)
    23
    >>> remove_two(23)
    0
    >>> remove_two(99)
    99
    """
    def remove(________):

        removed = _______________________

        while _______________________ > 0:

            _____________________________

            removed = removed // 10

            if __________________________:

                _________________________

        return __________________________

    return __________________________
```

```
def make_digit_remover(i):
    def remove(n):
        removed = n
        while removed > 0:
            digit = removed % 10
            removed = removed // 10
            if digit == i:
                return removed
        return n
    return remove
```

2. Implement `compound`, which takes in a single-argument function `base_func` and returns a two-argument compounder function `g`. The function `g` takes in an integer `x` and positive integer `n`.

   Each call to `g` will print the result of calling `f` repeatedly 0,1,.., n-1 times on `x`. That is, `g(x, 2)` prints `x`, then `f(x)`. Then, `g` will return the next two-argument compounder function.

```
def compound(base_func, prev_compound=lambda x: x):
    """
    >>> add_one = lambda x: x + 1
    >>> adder = compound(add_one)
    >>> adder = adder(3, 2)
    3        # 3
    4        # f(3)
    >>> adder = adder(4, 4)
    6        # f(f(4))
    7        # f(f(f(4)))
    8        # f(f(f(f(4))))
    9        # f(f(f(f(f(4)))))
    """
    def g(x, n):
      new_comp = ____________________________

      while n > 0:
        print(____________________________)

        new_comp = (lambda save_comp: \
                    ________________________)(____________)

        ________________________________________________

      return ______________________________________________

    return ____________________________________
```

```
def compound(base_func, prev_compound=lambda x : x):
  def g(x, n):
    new_comp = prev_compound
    while n > 0:
      print(new_comp(x))
      new_comp = (lambda save_comp: \
                  lambda x: base_func(save_comp(x)))(new_comp)
      n -= 1
    return compound(base_func, new_comp)
  return g
```

4 Lists

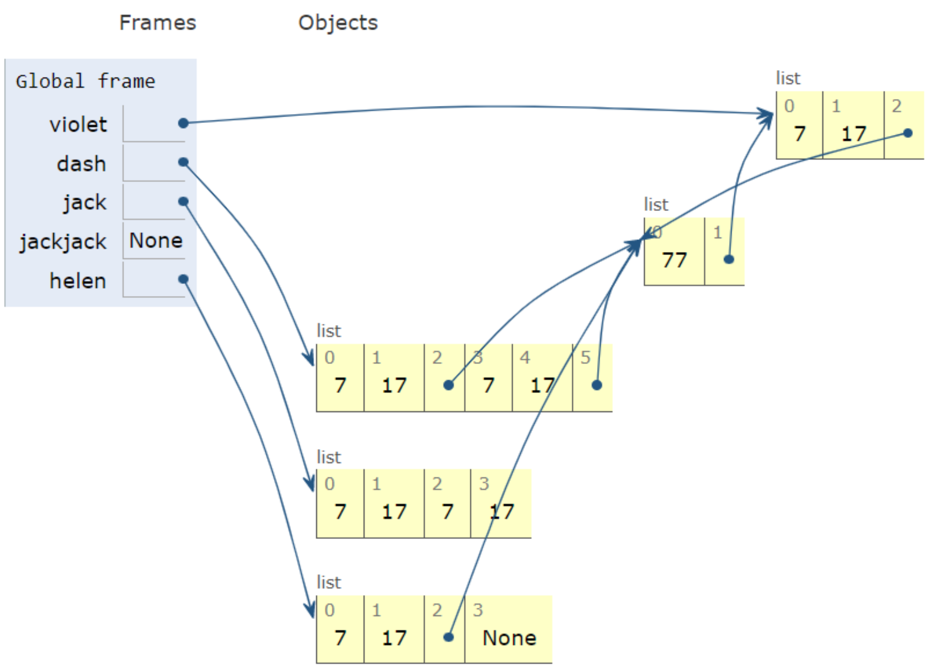
1. Draw the box-and-pointer diagram.

```
>>> violet = [7, 77, 17]
>>> violet.append([violet.pop(1)])

>>> dash = violet * 2
>>> jack = dash[3:5]
>>> jackjack = jack.extend(jack)

>>> helen = list(violet)
>>> helen += [jackjack]
>>> helen[2].append(violet)
```

https://goo.gl/EAmZBW



Teaching Tips

- Draw out the box and pointer diagram for each part.
- Try to highlight when pointers change vs. when the values a pointer points to change.
- If students are very confused about this problem, try going over the PythonTutor!

2. Write a function `duplicate_list`, which takes in a list of positive integers and returns a new list with each element `x` in the original list duplicated `x` times.

```
def duplicate_list(lst):
    """
    >>> duplicate_list([1, 2, 3])
    [1, 2, 2, 3, 3, 3]
    >>> duplicate_list([5])
    [5, 5, 5, 5, 5]
    """
    ________________________________

    for ____________________________:

        for ____________________________:

            ________________________________

    ________________________________
```

```
new_list = []
for x in lst:
    for i in range(x):
        new_list = new_list + [x]
return new_list
```

Teaching Tips

1. If students have trouble arriving at the solution, walk through the intuition of nested for loops and discuss what each loop represents. For example, the first loop represents iterating over each element of the list and the second one represents repeating that element.
2. This is a good problem to emphasize how we can format our logic and approach to problems based on the skeleton code

3. Write a function that takes as input a number `n` and a list of numbers `lst` and returns `True` if we can find a subset of `lst` that sums to `n`.

```
def add_up(n, lst):
    """
    >>> add_up(10, [1, 2, 3, 4, 5])
    True
    >>> add_up(8, [2, 1, 5, 4, 3])
    True
    >>> add_up(-1, [1, 2, 3, 4, 5])
    False
    >>> add_up(100, [1, 2, 3, 4, 5])
    False
    """
    if n == 0:
        return True
    if lst == []:
         return False
    else:
        first, rest = lst[0], lst[1:]
        return add_up(n - first, rest) or add_up(n, rest)
```

5 Iterators and Generators

1. Write a generator function `num_elems` that takes in a possibly nested list of numbers `lst` and yields the number of elements in each nested list before finally yielding the total number of elements (including the elements of nested lists) in `lst`. For a nested list, yield the size of the inner list before the outer, and if you have multiple nested lists, yield their sizes from left to right.

```
def num_elems(lst):
    """
    >>> list(num_elems([3, 3, 2, 1]))
    [4]
    >>> list(num_elems([1, 3, 5, [1, [3, 5, [5, 7]]]]))
    [2, 4, 5, 8]
    """

    count = ________________

    for ____________________________:

        if _________________________:

            for __________________________________:

                yield ____________________________

            ________________________________

        else:

            ________________________________

    yield ____________________________
```

```
def num_elems(lst):
    count = 0
    for elem in lst:
        if isinstance(elem, list):
            for c in num_elems(elem):
                yield c
            count += c
        else:
            count += 1
    yield count
```

`count` refers to the number of elements in the current list `lst` (including the number of elements inside any nested list). Determine the value of `count` by looping through each element of the current list `lst`. If we have an element `elem` which is of type **`list`**, we want to yield the number of elements in each nested list of `elem` before finally yielding the total number of elements in `elem`. We can do this with a recursive call to `num_elems`. Thus, we yield all the values that need to be yielded using the inner for loop. The last number yielded by this inner loop is the total number of elements in `elem`, which we want to increase `count` by. Otherwise, if `elem` is not a list, then we can simply increase `count` by 1. Finally, yield the total count of the list.

Teaching Tips

- Double check with your students to make sure they understand the differences between iterables and iterators.
- When we call next(), we pick up from where the last yield statement ran.
- The `+= c` line may be tricky to get. It could be useful to tell students beforehand that the variable in a **`for`** loop persists after iteration as the last value it took on.
- Try walking through one of the doctests if students are confused by what the problem is asking for.
- Make sure they understand that nested lists are processed first; this implies some kind of recursion.