6. (20 points) Palindromes

Definition. A palindrome is a sequence that has the same elements in normal and reverse order.

(a) (3 pt) Implement `pal`, which takes a positive integer `n` and returns a positive integer with the digits of `n` followed by the digits of `n` in reverse order.

Important: You may not write `str`, `repr`, `list`, `tuple`, `[`, or `]`.

```
def pal(n):
    """Return a palindrome starting with n.

    >>> pal(12430)
    1243003421
    """
    m = n

    while m:

        n, m = n * 10 + m % 10, m // 10

    return n
```

(b) (4 pt) Implement `contains`, which takes non-negative integers `a` and `b`. It returns whether all of the digits of `a` also appear in order among the digits of `b`.

Important: You may not write `str`, `repr`, `list`, `tuple`, `[`, or `]`.

```
def contains(a, b):
    """Return whether the digits of a are contained in the digits of b.

    >>> contains(357, 12345678)
    True
    >>> contains(753, 12345678)
    False
    >>> contains(357, 37)
    False
    """
    if a == b:

        return True

    if a > b:

        return False

    if a % 10 == b % 10:

        return contains(a // 10, b // 10)

    else:

        return contains(a, b // 10)
```

4. **(10 points) Editor**

Definitions. An *edit* is a pure function that takes a non-negative integer and returns a non-negative integer. An *editor* for a non-negative integer `n` is a function that takes an *edit*, applies it to `n`, displays the result, and then returns an editor for the result.

(a) **(3 pt)** Implement `make_editor`, which takes a non-negative integer `n` and a one-argument function `pr`. It returns an *editor* for `n` that uses `pr` to display the result of each *edit*.

(b) **(5 pt)** Implement `insert`, which takes a single digit `d` (from 0 to 9) and a non-negative position `k`. It returns an *edit* that inserts `d` into its argument `n` at position `k`, where `k` counts the number of digits from the end of `n`. **Assume that `k` is not larger than the number of digits in `n`. Your solution must be recursive.**

(c) **(2 pt)** Implement `delete`, which takes a non-negative integer `k` and returns an *edit* that deletes the last `k` digits of its argument `n`. You may use `pow` or `**` in your solution.

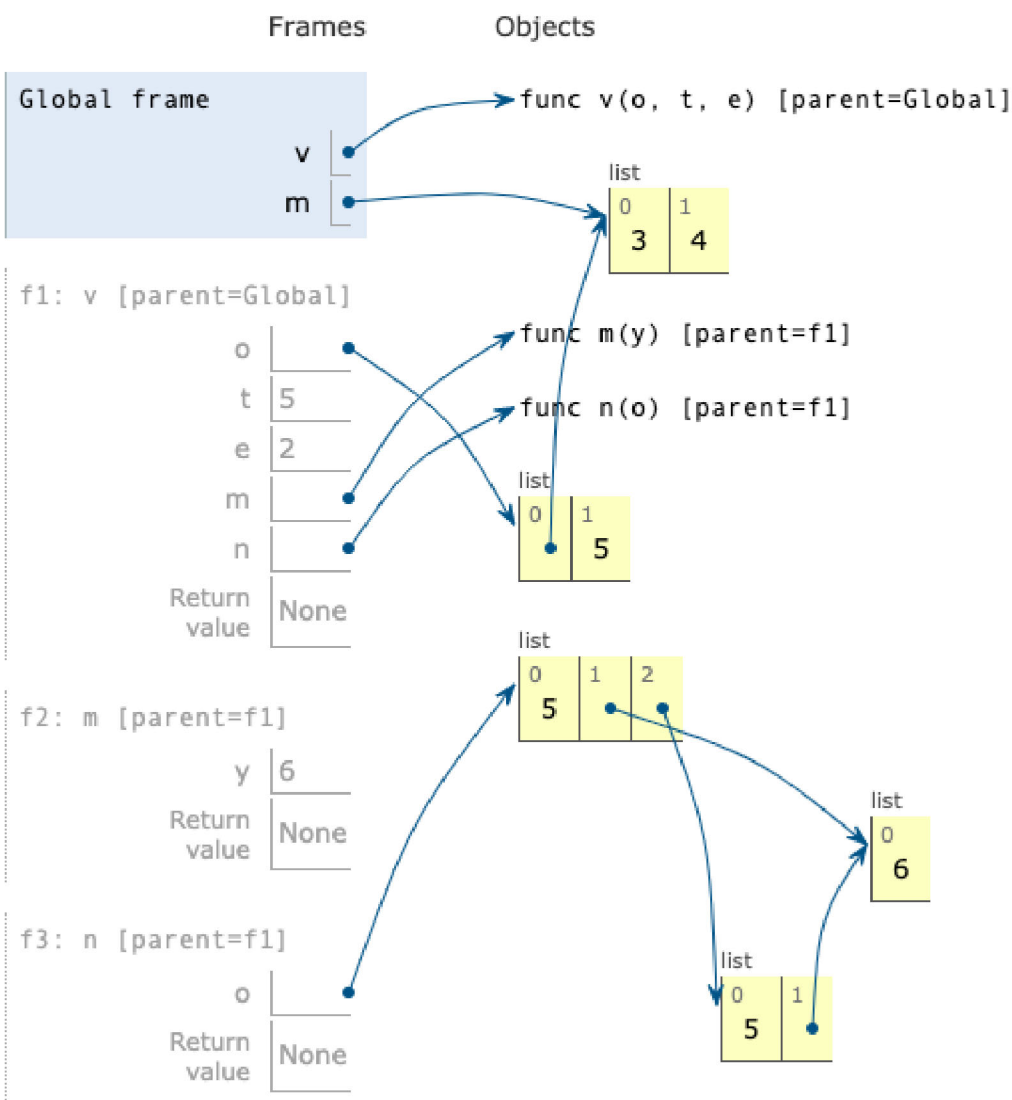
```
def make_editor(n, pr):
    """Return an editor for N.

    >>> f = make_editor(2018, lambda n: print('n is now', n))
    >>> f = f(delete(3))     # delete the last 3 digits from the end of 2018
    n is now 2
    >>> f = f(insert(4, 0))  # insert digit 4 at the end of 2 (position 0)
    n is now 24
    >>> f = f(insert(3, 1))  # insert digit 3 in the middle of 24 (position 1)
    n is now 234
    >>> f = f(insert(1, 3))  # insert digit 1 at the start of 234 (position 3)
    n is now 1234
    >>> f = make_editor(123, print)(delete(10))  # delete 10 digits from the end of 123
    0
    """
    def editor(edit):

        result = edit(n)

        pr(result)

        return make_editor(result, pr)

    return editor

def insert(d, k):

    def edit(n):

        if k == 0:

            return d + 10 * n

        else:

            return n % 10 + 10 * insert(d, k-1)(n//10)

    return edit

delete = lambda k: lambda n: n // 10 ** k
```

1. **(8.0 points) Political Environment**

Fill in each blank in the code example below so that executing it would generate the following environment diagram.

RESTRICTIONS. You must use all of the blanks. Each blank can only include one statement or expression.

Click here to open the diagram in a new window/tab




```
def v(o, t, e):
    def m(y):

        _________
           (a)

        _________
           (b)

    def n(o):

        o.append(_________)
```

```
                   (c)

        o.append(_________)
                   (d)

     m(e)
     n([t])
     e = 2

m = [3, 4]

v(m, 5, 6)
```

(a) **(2.0 pt)** Fill in blank (a). **You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.**

nonlocal o

(b) **(2.0 pt)** Fill in blank (b). **You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.**

o = [o, t]

(c) **(2.0 pt)** Fill in blank (c). **You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.**

[e]

(d) **(2.0 pt)** Which of these could fill in blank (d)? **Check all that apply.**

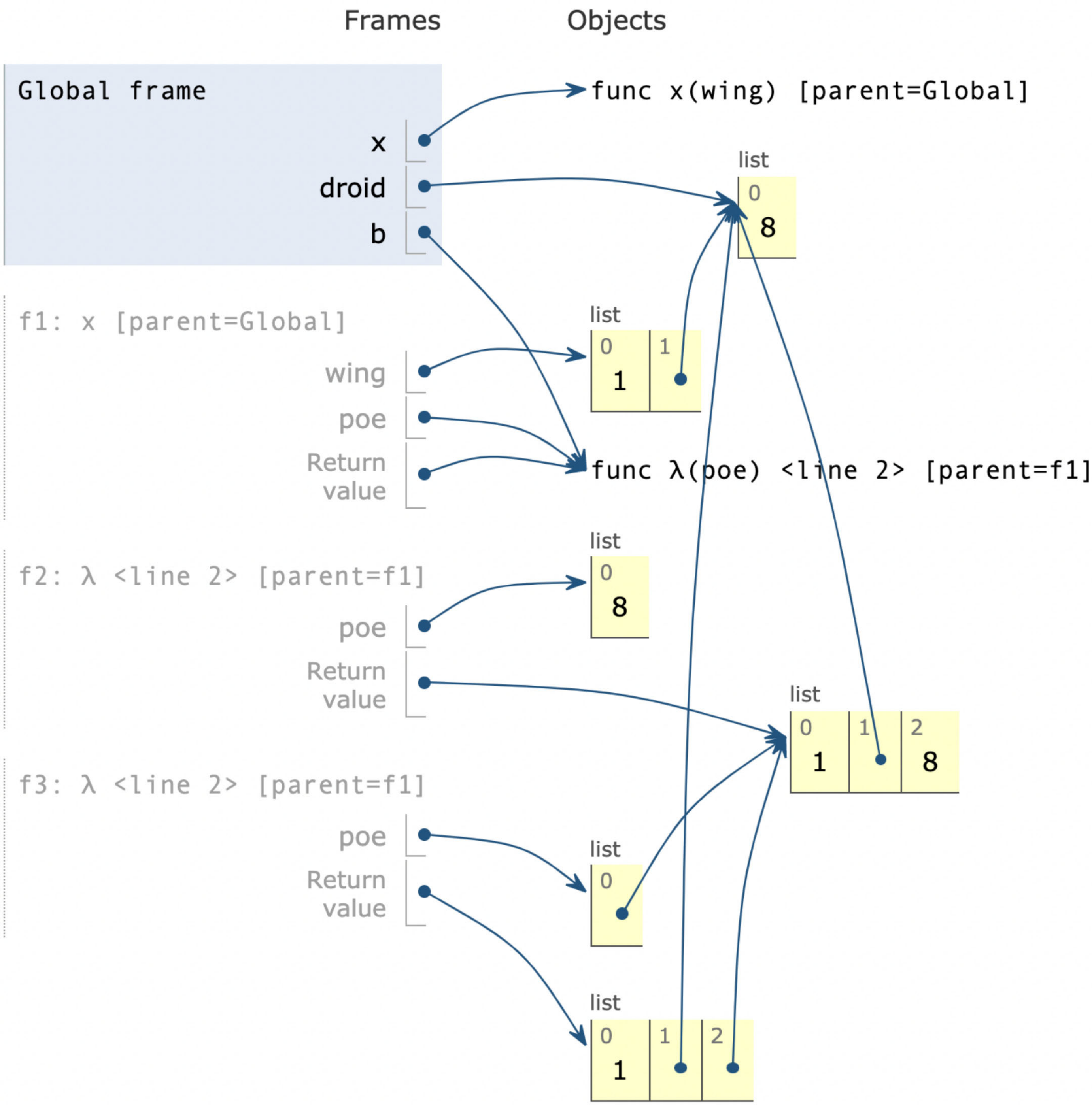
- ☐ `o`
- ☐ `[o]`
- ■ `list(o)`
- ☐ `[list(o)]`
- ☐ `list([o])`
- ■ `o + []`
- ■ `[o[0], o[1]]`
- ■ `o[:]`

1. (6 points) The Droids You're Looking For

Fill in each blank in the code example below so that executing it would generate the following environment diagram on tutor.cs61a.org.

RESTRICTIONS. You must use all of the blanks. Each blank can only include one statement or expression.

Click here to open the diagram in a new window/tab




```
def x(wing):

    poe = lambda poe: _________
```

```
    #                     (a)

    wing.append(_________)
    #              (b)

    return _________
    #         (c)

droid = [8]

b = x([1])

_________
#  (d)
```

(a) **(1 pt)** Which of these could fill in blank (a)? Check all that apply.

- [x] `wing + poe`
- [ ] `wing.extend(poe)`
- [ ] `wing.append(poe)`
- [ ] `list(wing).extend(poe)`
- [ ] `list(wing).append(poe)`

(b) **(1 pt)** Fill in blank (b).

droid

(c) **(1 pt)** Which of these could fill in blank (c)?

- (●) `poe`
- ( ) `poe(droid)`
- ( ) `poe(wing)`
- ( ) `poe(b)`

(d) **(3 pt)** Fill in blank (d).

b([b([8])])

7. (10 points) Summer Camp

(a) (6 pt) Implement `sums`, which takes two positive integers `n` and `k`. It returns a list of lists containing all the ways that a list of `k` positive integers can sum to `n`. Results can appear in any order.

```
def sums(n, k):
    """Return the ways in which K positive integers can sum to N.

    >>> sums(2, 2)
    [[1, 1]]
    >>> sums(2, 3)
    []
    >>> sums(4, 2)
    [[3, 1], [2, 2], [1, 3]]
    >>> sums(5, 3)
    [[3, 1, 1], [2, 2, 1], [1, 3, 1], [2, 1, 2], [1, 2, 2], [1, 1, 3]]
    """
    if k == 1:
        return [[n]]
    y = []
    for x in range(1, n): # range(1, n-k+1) is OK
        y.extend([s + [x] for s in sums(n-x, k-1)])
    return y

    OR

    if n == 0 and k == 0:
        return [[]]
    y = []
    for x in range(1, n+1):
        y.extend([s + [x] for s in sums(n-x, k-1)])
    return y
```

(b) (4 pt) *Why so many lines?* Implement `f` and `g` for this alternative version of the `sums` function.

```
f = lambda x, y: (x and [[x] + z for z in y] + f(x-1, y)) or []

def sums(n, k):
    """Return the ways in which K positive integers can sum to N.

    >>> sums(2, 2)
    [[1, 1]]
    >>> sums(4, 2)
    [[3, 1], [2, 2], [1, 3]]
    >>> sums(5, 3)
    [[3, 1, 1], [2, 2, 1], [2, 1, 2], [1, 3, 1], [1, 2, 2], [1, 1, 3]]
    """
    g = lambda w: (w and f(n, g(w-1))) or [[]]
    return [v for v in g(k) if sum(v) == n]
```

2. (10.0 points) Yield, Fibonacci!

(a) (4.0 points)

Implement `fibs`, a generator function that takes a one-argument pure function `f` and yields all Fibonacci numbers `x` for which `f(x)` returns a true value.

The Fibonacci numbers begin with 0 and then 1. Each subsequent Fibonacci number is the sum of the previous two. Yield the Fibonacci numbers in order.

```
def fibs(f):
    """Yield all Fibonacci numbers x for which f(x) is a true value.

    >>> odds = fibs(lambda x: x % 2 == 1)
    >>> [next(odds) for i in range(10)]
    [1, 1, 3, 5, 13, 21, 55, 89, 233, 377]
    >>> bigs = fibs(lambda x: x > 20)
    >>> [next(bigs) for i in range(10)]
    [21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
    >>> evens = fibs(lambda x: x % 2 == 0)
    >>> [next(evens) for i in range(10)]
    [0, 2, 8, 34, 144, 610, 2584, 10946, 46368, 196418]
    """

    n, m = 0, 1

    while _________:
             (a)

        if _________:
              (b)

            _________
               (c)

        _________
           (d)
```

i. (1.0 pt) Which of these could fill in blank (a)?

- ○ `f(n)`
- ○ `f(m)`
- ○ `f(n) or f(m)`
- ○ `f(n) and f(m)`
- ● `True`
- ○ `False`

ii. (1.0 pt) Which of these could fill in blank (b)?

- ● `f(n)`
- ○ `f(m)`
- ○ `f(n) or f(m)`
- ○ `f(n) and f(m)`
- ○ `True`
- ○ `False`

iii. (1.0 pt) Fill in blank (c).

yield n

iv. (1.0 pt) Fill in blank (d).

n, m = m, n + m

5. (7 points) Do You Yield?

(a) (6 pt) Implement `partitions`, which is a generator function that takes positive integers `n` and `m`. It yields strings describing all partitions of `n` using parts up to size `m`. Each partition is a sum of non-increasing positive integers less than or equal to `m` that totals `n`. The `partitions` function yields a string for each partition exactly once.

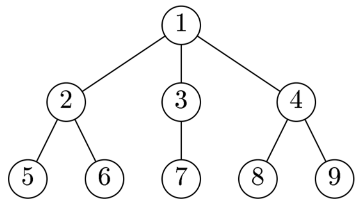
You may **not** use `lambda`, `if`, `and`, `or`, lists, tuples, or dictionaries in your solution (other than what already appears in the template).

```
def partitions(n, m):
    """Yield all partitions of N using parts up to size M.

    >>> list(partitions(1, 1))
    ['1']
    >>> list(partitions(2, 2))
    ['2', '1 + 1']
    >>> list(partitions(4, 2))
    ['2 + 2', '2 + 1 + 1', '1 + 1 + 1 + 1']
    >>> for p in partitions(6, 4):
    ...     print(p)
    4 + 2
    4 + 1 + 1
    3 + 3
    3 + 2 + 1
    3 + 1 + 1 + 1
    2 + 2 + 2
    2 + 2 + 1 + 1
    2 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 1 + 1
    """

    if n == m:

        yield str(m)

    if n > 0 and m > 0:

        for p in partitions(n - m, m):

            yield str(m) + ' + ' + p

        yield from partitions(n, m - 1)
```

(b) (1 pt) Circle the order of growth of the value of `len(list(partitions(n, 2)))` as a function of `n`. For example, the value of `len(list(partitions(3, 2)))` is 2, because there are 2 partitions of 3 using parts up to size 2: $2 + 1$ and $1 + 1 + 1$. Assume `partitions` is implemented correctly.

Constant   Logarithmic   Linear   Quadratic   Exponential   None of these

5. (13 points) Level-Headed Trees A *level-order traversal* of a tree, T, traverses the root of T (level 0), then the roots of all the branches of T (level 1) left to right, then all the roots of the branches of the nodes traversed in level 1, (level 2) and so forth. Thus, a level-order traversal of the tree



visits nodes with labels 1, 2, 3, 4, 5, 6, 7, 8, 9 in that order.

(a) (9 pt) Fill in the following generator function to yield the labels of a given tree in level order. All trees are of the class `Tree`, defined on page 2 of the Midterm 2 Study Guide. The strategy is to use a helper function that yields nodes at one level, and then to call this function with increasing levels until a level does not yield any labels. You may not need all the lines.

```
def level_order(tree):
    """Generate all labels of tree in level order.

    >>> list(level_order(Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])))
    [1, 2, 5, 3, 4]
    """
    def one_level(tree, k):
        """Generate the labels of tree at level k."""

        if  k == 0:

            yield tree.label

        else:

            for child in tree.branches:


                yield from one_level(child, k-1)

    level, count = 0, True

    while count:

        count = 0



        for label in one_level(tree, level):


            count += 1


            yield label


        level += 1
```

5. (6 points) To-Do Lists

Implement the `TodoList` and `Todo` classes. When a `Todo` is `complete`, it is removed from all the `TodoList` instances to which it was ever added. Track both the number of completed `Todo` instances in each list and overall so that printing a `TodoList` instance matches the behavior of the doctests below. Assume the `complete` method of a `Todo` instance is never invoked more than once.

```
class TodoList:
    """A to-do list that tracks the number of completed items in the list and overall.

    >>> a, b = TodoList(), TodoList()
    >>> a.add(Todo('Laundry'))
    >>> t = Todo('Shopping')
    >>> a.add(t)
    >>> b.add(t)
    >>> print(a)
    Remaining: ['Laundry', 'Shopping'] ; Completed in list: 0 ; Completed overall: 0
    >>> print(b)
    Remaining: ['Shopping'] ; Completed in list: 0 ; Completed overall: 0
    >>> t.complete()
    >>> print(a)
    Remaining: ['Laundry'] ; Completed in list: 1 ; Completed overall: 1
    >>> print(b)
    Remaining: [] ; Completed in list: 1 ; Completed overall: 1
    >>> Todo('Homework').complete()
    >>> print(a)
    Remaining: ['Laundry'] ; Completed in list: 1 ; Completed overall: 2
    """
    def __init__(self):
        self.items, self.complete = [], 0
    def add(self, item):
        self.items.append(item)

        item.lists.append(self)
    def remove(self, item):


        self.complete += 1


        self.items.remove(item)

    def __str__(self):
        return ('Remaining: ' + str([t.task for t in self.items]) +
                ' ; Completed in list: ' + str(self.complete) +


                ' ; Completed overall: ' + str(Todo.done))
class Todo:
    done = 0
    def __init__(self, task):
        self.task, self.lists = task, []
    def complete(self):

        Todo.done += 1
        for t in self.lists:
            t.remove(self)
```

5. (8 points) Midterm Elections

(a) (6 pt) Implement the `Poll` class and the `tally` function, which takes a choice `c` and returns a list describing the number of votes for `c`. This list contains pairs, each with a name and the number of times `vote` was called on that `choice` at the `Poll` with that name. Pairs can be in any order. Assume all `Poll` instances have distinct names. *Hint*: the dictionary `get(key, default)` method (MT 2 guide, page 1 top-right) returns the value for a `key` if it appears in the dictionary and `default` otherwise.

```
class Poll:
    s = []
    def __init__(self, n):

        self.name = n

        self.votes = {}

        Poll.s.append(self)

    def vote(self, choice):

        self.votes[choice] = self.votes.get(choice, 0) + 1

def tally(c):
    """Tally all votes for a choice c as a list of (poll name, vote count) pairs.

    >>> a, b, c = Poll('A'), Poll('B'), Poll('C')
    >>> c.vote('dog')
    >>> a.vote('dog')
    >>> a.vote('cat')
    >>> b.vote('cat')
    >>> a.vote('dog')
    >>> tally('dog')
    [('A', 2), ('C', 1)]
    >>> tally('cat')
    [('A', 1), ('B', 1)]
    """

    return [(p.name, p.votes[c]) for p in Poll.s if c in p.votes]
```

(b) (2 pt) Implement the `vote` method of the `Crooked` class, which only records every other `vote` call for each `Crooked` instance. Only odd numbered calls to `vote` are recorded, e.g., first, third, fifth, etc.

```
class Crooked(Poll):
    """A poll that ignores every other call to vote.

    >>> d = Crooked('D')
    >>> for s in ['dog', 'cat', 'dog', 'cat', 'cat']:
    ...     d.vote(s)
    >>> d.votes
    {'dog': 2, 'cat': 1}
    """
    record = True
    def vote(self, choice):
        if self.record:

            Poll.vote(self, choice)

        self.record = not self.record
```

Trees

4. (11 points) Tree Time

Definition. A *runt node* is a node in a tree whose label is smaller than all of the labels of its siblings. A sibling is another node that shares the same parent. A node with no siblings is a runt node.

(a) (7 pt) Implement `runts`, which takes a `Tree` instance `t` in which *every label is different* and returns a list of the labels of all runt nodes in `t`, in any order. Also implement `apply_to_nodes`, which returns nothing and is part of the implementation. Do **not** mutate any tree. The `Tree` class is on the Midterm 2 Guide.

```
def runts(t):
    """Return a list of the labels of all smallest siblings in any order.

    >>> sorted(runts(Tree(9, [Tree(3), Tree(4, [Tree(5, [Tree(6)]), Tree(7)]), Tree(2)])))
    [2, 5, 6, 9]
    """
    result = [ ]

    def g(node):

        if not node.is_leaf():

            result.append(min([b.label for b in node.branches]))

    apply_to_nodes(g, t)

    return [t.label] + result


def apply_to_nodes(f, t):
    """Apply a function f to each node in a Tree instance t."""


    f(t)


    for b in t.branches:


        apply_to_nodes(f, b)
```

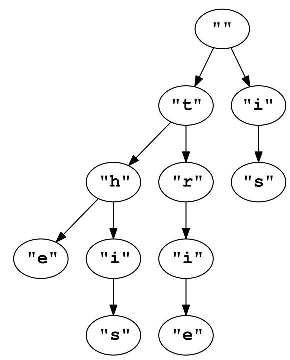
(b) (4 pt) Implement `max_label`, which takes a `Tree t` and returns its largest label. Do **not** mutate any tree.

```
def max_label(t):
    """Return the largest label in t.

    >>> max_label(Tree(4, [Tree(5), Tree(3, [Tree(6, [Tree(1), Tree(2)])])]))
    6
    """
    def f(node):

        nonlocal t

        t = max(t, node, key=lambda n: n.label)

    apply_to_nodes(f, t)

    return t.label
```

6. (15 points) Trie this

A Trie is a Tree where every node in the tree contains a single letter except for the root which is always the empty string. Every path from the root to a leaf forms a word. You may assume no words are substrings of other words in the trie (e.g., "hi" and "him"). The figure below is a trie generated by storing the words ["this", "is", "the", "trie"]. The `Tree` class is defined on Page 2 of the Midterm 2 Study Guide.

(a) (7 pt) Implement `add_word` which takes a Trie and a word and adds the word to the trie.




```
def make_trie(words):
    """ Makes a tree where every node is a letter of a word.
        All words end as a leaf of the tree.
        words is given as a list of strings.
    """
    trie = Tree('')
    for word in words:
        add_word(trie, word)
    return trie


def add_word(trie, word):

    if word == '':
        return

    branch = None

    for b in trie.branches:

        if b.label == word[0]:

            branch = b

    if not branch:

        branch = Tree(word[0])

        trie.branches.append(branch)

    add_word(branch, word[1:])
```

(b) (8 pt) Implement `get_words`, which takes a Trie and returns a list of all the words the Trie is storing.

```
def get_words(trie):
    """
    >>> get_words(make_trie(['this', 'is', 'the', 'trie']))
    ['this', 'the', 'trie', 'is']
    """
    if trie.is_leaf():

        return [trie.label]

    return sum([[trie.label + word for word in get_words(branch)] for branch in trie.branches], [])
```

3. (21 points) College Party

In a US presidential election, each state has a number of electors.

Definition: For some collection of states `s`, a *win by at least* `k` is a (possibly empty) subset `w` of `s` such that the total number of electors for the states in `w` is at least `k` more than the total number of electors for the states not in `w` but in `s`.

For example, in the `battleground` states below, Arizona (`AZ`), Pennsylvania (`PA`), and Michigan (`MI`) have a total of $11 + 20 + 16 = 47$ electors. The remaining states have a total of $6 + 16 + 10 = 32$ electors. So, the subset `<AZ PA MI>` is a win by 47 - 32 = 15.

```
class State:
    electors = {}
    def __init__(self, code, electors):
        self.code = code
        self.electors = electors
        State.electors[code] = electors

battleground = [State('AZ', 11), State('PA', 20), State('NV', 6),
                State('GA', 16), State('WI', 10), State('MI', 16)]
```

The total number of electors for an empty set of states is 0.

The `print_all` function prints all elements of an iterable.

```
def print_all(s):
    for x in s:
        print(x)
```

(a) (8 points)

Implement `wins`, a generator function that takes a list of `State` instances `states` and an integer `k`. For every possible *win by at least* `k` among the `states`, it yields a **linked list containing strings** of the two-letter codes for the states in that win.

Any order of the wins and any order of the states within a win is acceptable.

A linked list is a `Link` instance or `Link.empty`. The `Link` class appears on the Midterm 2 Study Guide.

```
def wins(states, k):
    """Yield each linked list of two-letter state codes that describes a win by at least k.

    >>> print_all(wins(battleground, 50))
    <AZ PA NV GA WI MI>
    <AZ PA NV GA MI>
    <AZ PA GA WI MI>
    <PA NV GA WI MI>
    >>> print_all(wins(battleground, 75))
    <AZ PA NV GA WI MI>
    """

    if _________:
        # (a)

        yield Link.empty

    if states:

        first = states[0].electors
```

```
    for win in wins(states[1:], _________):
        #                           (b)

        yield Link(_________, win)
        #             (c)

    yield from wins(states[1:], _________)
    #                              (d)
```

i. (2 pt) Which of the these could fill in blank (a)?

- ○ `k >= 0`
- ○ `k <= 0`
- ○ `k == 0`
- ○ `not states`
- ○ `k >= 0 and not states`
- ● `k <= 0 and not states`
- ○ `k == 0 and not states`

ii. (2 pt) Which of the these could fill in blank (b)?

- ● `k - first`
- ○ `k + first`
- ○ `k`
- ○ `-k`
- ○ `0`
- ○ `first`
- ○ `min(k, first)`
- ○ `max(k, first)`

iii. (2 pt) Fill in blank (c).

states[0].code

iv. (2 pt) Which of the these could fill in blank (d)?

- ○ `k - first`
- ● `k + first`
- ○ `k`
- ○ `-k`
- ○ `0`
- ○ `first`
- ○ `min(k, first)`
- ○ `max(k, first)`

(b) (7 points)

Implement `must_win`, which takes a list of `State` instances `states` and an integer `k`. It returns a list of two-letter state codes (strings) for all states that appear in every *win by at least k* among the `states`. Assume `wins` is implemented correctly.

```
def must_win(states, k):
    """List all states that must be won in every scenario that wins by k.

    >>> must_win(battleground, 50)
    ['PA', 'GA', 'MI']
    >>> must_win(battleground, 75)
    ['AZ', 'PA', 'NV', 'GA', 'WI', 'MI']
    """
    def contains(s, x):
        """Return whether x is a value in linked list s."""

        return (_________) and (_________)
        #          (a)              (b)

    return [_________ for s in states if _________([_________ for w in wins(states, k)])]
    #          (c)                             (d)        (e)
```

i. (1 pt) Which of these could fill in blank (a)?

- ○ `s is Link.empty`
- ● `s is not Link.empty`
- ○ `x in s`
- ○ `x not in s`
- ○ `x == s.first`
- ○ `x != s.first`

ii. (2 pt) Fill in blank (b).

s.first == x or contains(s.rest, x)

iii. (1 pt) Fill in blank (c).

s.code

iv. (1 pt) Fill in blank (d) with a single function name.

all

v. (2 pt) Fill in blank (e).

contains(w, s.code)

4. (11 points) Both Ways

(a) (4 pt) Implement `both`, which takes two *sorted* linked lists composed of `Link` objects and returns whether some value is in both of them. The `Link` class is defined on the midterm 2 study guide.
Important: You may **not** use `len`, `in`, `for`, `list`, slicing, element selection, addition, or list comprehensions.

```
def both(a, b):
    """Return whether there is any value that appears in both a and b, two sorted Link instances.

    >>> both(Link(1, Link(3, Link(5, Link(7)))), Link(2, Link(4, Link(6))))
    False
    >>> both(Link(1, Link(3, Link(5, Link(7)))), Link(2, Link(7, Link(9))))  # both have 7
    True
    >>> both(Link(1, Link(4, Link(5, Link(7)))), Link(2, Link(4, Link(5))))  # both have 4 and 5
    True
    """
    if a is Link.empty or b is Link.empty:

        return False

    if a.first > b.first:

        a, b = b, a

    return a.first == b.first or both(a.rest, b)
```

(b) (2 pt) Circle the Θ expression that describes the minimum number of comparisons (e.g., <, >, <=,==, or >= expressions) required to verify that two sorted lists of length n contain no values in common.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$ None of these

(c) (5 pt) Implement `ways`, which takes two values *start* and *end*, a non-negative integer k, and a list of one-argument functions `actions`. It returns the number of ways of choosing functions f_1, f_2, ..., f_j from `actions`, such that $f_1(f_2(...(f_j(\mathit{start}))))$ equals *end* and $j \leq k$. The same action function can be chosen multiple times. If a sequence of actions reaches *end*, then no further actions can be applied (see the first example below).

```
def ways(start, end, k, actions):
    """Return the number of ways of reaching end from start by taking up to k actions.

    >>> ways(-1, 1, 5, [abs, lambda x: x+2])   # abs(-1) or -1+2, but not abs(abs(-1))
    2
    >>> ways(1, 10, 5, [lambda x: x+1, lambda x: x+4])  # 1+1+4+4, 1+4+4+1, or 1+4+1+4
    3
    >>> ways(1, 20, 5, [lambda x: x+1, lambda x: x+4])
    0
    >>> ways([3], [2, 3, 2, 3], 4, [lambda x: [2]+x, lambda x: 2*x, lambda x: x[:-1]])
    3
    """
    if start == end:

        return 1

    elif k == 0:

        return 0

    return sum([ways(f(start), end, k - 1, actions) for f in actions])
```

3. (8 points) Seeing Double

Fill in the functions below to produce linked lists in which each item of the original list is repeated immediately after that item. Your solutions should be iterative, not recursive.

(a) (4 pt) The function `double1` is *non-destructive,* and produces a new list without disturbing the old.

```
def double1(L):
    """Returns a list in which each item in L appears twice in sequence.
    It is non-destructive.
    >>> Q = Link(3, Link(4, Link(1)))
    >>> double1(Q)
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    >>> Q
    Link(3, Link(4, Link(1)))
    >>> double1(Link.empty)
    ()
    """

    result = Link.empty
    last = None
    while L is not Link.empty:
        if last is None:
            result = Link(L.first, Link(L.first))
            last = result.rest
        else:
            last.rest = Link(L.first, Link(L.first))
            last = last.rest.rest
        L = L.rest
    return result
```

(b) (4 pt) The function `double2` is *destructive,* and reuses `Link` objects in the original list wherever possible.

```
def double2(L):
    """Destructively modifies L to insert duplicates of each item immediately
    following the item, returning the result.
    >>> Q = Link(3, Link(4, Link(1)))
    >>> double2(Q)
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    >>> Q
    Link(3, Link(3, Link(4, Link(4, Link(1, Link(1))))))
    """

    result = L
    while L is not Link.empty:
        L.rest = Link(L.first, L.rest)
        L = L.rest.rest
    return result
```