CS 61A Summer 2026

Rebecca Dang

Midterm

Solutions last updated: Tuesday, July 14, 2026

Your Name: _____

Your Student ID: _____

Room you are taking the exam in: _____

Name and SID of the person to your **left**: _____

Name and SID of the person to your **right**: _____

You have 110 minutes. There are 8 questions of varying credit. (60 points total)

Question	1	2	3	4	5	6	7	8	Total
Points	9	10	6	6	12	11	6	0	60

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (Completely unfilled)
- ◉ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares
- ☑ (Don't do this)

Make sure to **write your student ID at the top of each page** in the provided blank.

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam. I will not use any other unauthorized materials or devices during this exam. I will not communicate with anyone other than the proctors during this exam. I will not share any information about the exam with anyone until after the end of the exam period.

Acknowledge that you have read and agree to the honor code above and **sign your name below**:

Q1 *Do you speak for the Trees? (WWPD)* **(9 points)**

Consider the following code that was executed in a Python interpreter. For each of the lines below, write exactly what Python would display as output in the interpreter, except in these cases:

1. If a function would be displayed, write `Function`.
2. If an error would be raised, write `Error`.
3. If nothing would be displayed, write `Nothing`.

For all subparts, if there are multiple lines of output, write all of them and clearly indicate when a new line begins.

Q1.1 (3 points) What would be displayed by evaluating the following code?

```
>>> sixty_seven = lambda x: lambda y: (print(x + y) and 67)
>>> seventy_six = lambda x: lambda y: (76 and print (x + 2*y))
>>> print(sixty_seven(7)(6) or seventy_six(6)(7))
```

```
13
20
None
```

The question continues on the next page (this space is intentionally blank).

```
t = tree(1, [
        tree(3, [tree(4)]),
        tree(5, [tree(6)])
    ])

def count_paths(t, f):
    """
    Takes in a tree `t` and a function `f`.
    It returns the number of paths from the root to any node in the tree
    for which calling `f` on the sum of the labels on the path
    returns a truthy value. You may assume it is implemented correctly.
    >>> count_paths(t, lambda x: x == 4) # 1 + 3 == 4 for the path 1 -> 3
    1
    """
    def helper(t, running_sum):
        total_sum = running_sum + label(t)
        if f(total_sum):
            count = 1
        else:
            count = 0
        for b in branches(t):
            count += helper(b, total_sum)
        return count
    return helper(t, 0)

six = lambda x: (x == 6 and print(x))
```

Q1.2 (2 points) What would be displayed by evaluating the following code, assuming the tree ADT was already previously defined? (The tree ADT is provided in your reference sheet.)

```
>>> count_paths(t, lambda x: (x * x) % 10 == 6)
```

2

Q1.3 (4 points) What would be displayed by evaluating the following code?

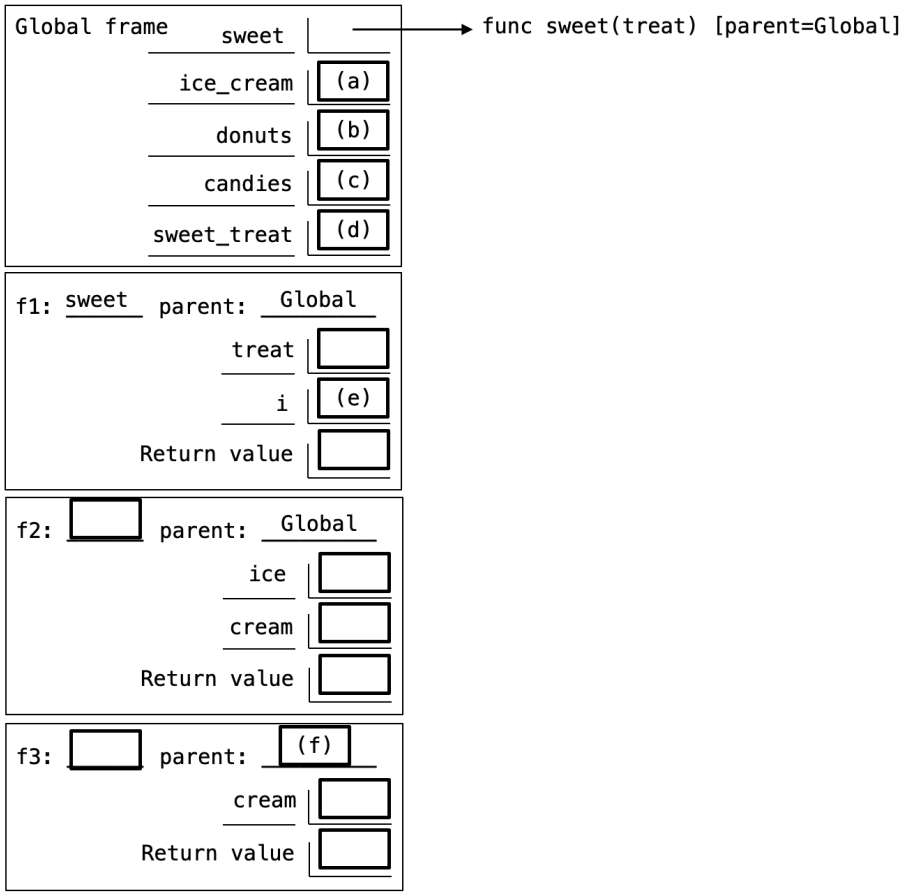
```
>>> (count_paths(t, six) - 1) and count_paths(t, lambda x: x % 2 == 0)
```

6
4

Q2 *Too Sweet!* **(10 points)**

Complete the environment diagram below and then answer the questions that follow. There is one question for each labeled blank in the diagram. The blanks with no labels have no questions associated with them and are not scored. If a blank contains an arrow to a function, write the function as it would appear in the diagram.

```
1  def sweet(treat):
2      i = 0
3      while i < len(donuts):
4          if donuts[i] % 3 == 0:
5              treat.append(donuts.pop(i))
6          i += 1
7      return treat[:]
8
9  def ice_cream(ice, cream):
10     ice.insert(2, (lambda cream: [1, donuts[:3]])(ice))
11     return cream
12
13 donuts = list(range(1, 10))
14 candies = []
15
16 sweet_treat = sweet(candies)
17 ice_cream = ice_cream(sweet_treat, sweet_treat[:])
```



Q2.1 (1 point) Fill in blank (a).

```
[3, 6, 9]
```

Q2.2 (2 points) Fill in blank (b).

```
[1, 2, 4, 5, 7, 8]
```

Q2.3 (1 point) Fill in blank (c).

```
[3, 6, 9]
```

Q2.4 (2 points) Fill in blank (d).

```
[3, 6, [1, [1, 2, 4]], 9]
```

Q2.5 (1 point) Fill in blank (e).

7

Q2.6 (1 point) Fill in blank (f).

- ○ `Global`
- ○ `f1`
- ● `f2`

Q2.7 (2 points) In one sentence, what does the **`sweet`** function do? (Anything you write beyond one sentence will not be graded.)

Removes (non-consecutive) multiples of 3 from **`donuts`**, adds them to **`treat`**, and returns a (shallow) copy of **`treat`**

Q3 *Bugs Begone* **(6 points)**

CS 61A course staff are trying to complete a coding assignment but are struggling. The assignment is to write a function `digit_filter`, which takes in `predicate`, a one-argument function, and `n`, a positive integer (e.g. `n > 10`), and returns the number of digits in `n` for which `predicate(digit)` returns `True`. Here's some doctests showing the intended behavior:

```
>>> is_even = lambda x: x % 2 == 0
>>> digit_filter(is_even, 2111)
1
>>> digit_filter(is_even, 9843)
2
>>> digit_filter(lambda x: x > 3, 123456789)
6
```

In **all** subparts below, you don't need to indent your answer when writing code. Each buggy implementation must be made correct by changing **exactly one line**.

Q3.1 (2 points) Sriya has decided to use a while loop to complete the assignment, but it's not working!

```
1 def digit_filter(predicate, n):
2     total = 0
3     while n >= 0:
4         if predicate(n % 10):
5             total += 1
6         n //= 10
7     return total
```

Which line number is Sriya's mistake on?

○ 1 ○ 2 ● 3 ○ 4

○ 5 ○ 6 ○ 7

What should that line be replaced with?

```
n > 0
```

Solution: The while loop condition being `while n >= 0` causes an infinite loop because we are repeatedly floor diving `n` by `10`, which means that eventually after processing the final digit, `n` will become 0. Thus, to fix the bug, the condition must exclude the case when `n == 0`.

The question continues on the next page (this space is intentionally blank).

Q3.2 (2 points) Emma has decided to use recursion instead, but it's also not working!

```
1 def digit_filter(predicate, n):
2     if n < 10:
3         return 0
4     if predicate(n % 10):
5         return 1 + digit_filter(predicate, n // 10)
6     return digit_filter(predicate, n // 10)
```

Which of the following changes, when made only to the specified line number **independently of the other answer options**, would **fix** Emma's mistake? Select all that apply.

Hint: `True` and `False` are internally stored as `1` and `0`, respectively, in Python. This means you can perform arithmetic on them, e.g. `True == 1` and `True + 3 == 4`.

- ☐ Line 2, `if n < 10 and predicate(n % 10):`
- ■ Line 2, `if n == 0:`
- ■ Line 3, `return predicate(n)`
- ■ Line 3, `return predicate(n % 10)`
- ☐ Line 4, `elif predicate(n % 10):`
- ☐ Line 4, `if predicate(n // 10):`
- ☐ Line 5, `return digit_filter(predicate, n // 10)`
- ☐ Line 6, `return 1 + digit_filter(predicate, n // 10)`

Solution: The 1st option is incorrect because if `n < 10` that means `n` is a single digit number, and we may want to return `1` or `0` depending on if `predicate` of that digit returns `True` (but in Line 3, we always `return 0`).

The 2nd option is correct because if `n == 0` that means we have already recursively processed all the digits and have reached the true base case. In this case, we do not want to change the running total number of digits for which the `predicate` returns `True`, so we should `return 0` (which is already on Line 3).

See the hint and explanation for the 1st option for why the 3rd option is correct. This is an example of arms-length recursion, which we generally do not recommend since it obfuscates the real base case. However, it is still a valid solution.

The 4th option is correct because it is equivalent to the third option. This is because `n` is a single digit number, so `n % 10` is equal to `n`.

The 5th option is incorrect because it is logically equivalent to the existing buggy code.

The 6th option is incorrect because `predicate` expects a single digit as input but `n // 10` may evaluate to number with multiple digits.

The 7th option is incorrect because we need to add 1 to our running recursive sum if `predicate` of the current digit we're looking at (`n % 10`) returns `True`, as the problem states.

The 8th option is incorrect because we only execute Line 6 if `predicate(n % 10)` was falsy, which means that we should *not* add 1 to our running recursive sum.

Q3.3 (2 points) Amy has taken an unconventional approach and has decided to use a helper function - and it's still wrong!

```
1 def digit_filter(predicate, n):
2     def helper(x, total):
3         if x > n:
4             return total
5         last = (n // x) % 10
6         return helper(10 ** x, total + predicate(last))
7     return helper(1, 0)
```

Which line number is Amy's mistake on?

○ 1 ○ 2 ○ 3 ○ 4

○ 5 ● 6 ○ 7

What should that line be replaced with?

```
return helper(x * 10, total + predicate(last))
```

Solution: This recursive approach uses a `helper` function to store the value of intermediate state, specifically the value of `x` and the `total` running sum of the number of digits for which `predicate` returned `True`. The existing code on lines 3 and 5 expect `x` to be consecutive powers of 10 (e.g. $10^0, 10^1, 10^2$, etc.), but when line 6 makes the recursive call with `10 ** x`, `x` quickly blows out of proportion. To fix this, we can pass in `x * 10` to perform the consecutive powers of 10 computation at each recursive call.

Q4 *Skipper* **(6 points)**

Definitions. Given a positive integer `n` and a positive step size `k`, a *skipper* is a subsequence of digits in `n` that are each `k` indices apart and all digits in the subsequence are equal to some *skipper digit* `d`. Digits in `n` outside of the skipper (e.g. the numbers between the skipper digits) do not need to be equal to `d`. A skipper's *length* is the number of skipper digits in the subsequence.

Implement a function `skipper`, which returns the length of the longest skipper in `n` with step size `k`, or `0` if there is no skipper in `n`.

```
def skipper(n: int, k: int) -> int:
    """
    >>> skipper(111, 1)
    3
    >>> skipper(10101, 2) # two skippers with three 1s and two 0s
    3
    >>> skipper(1, 3) # no skipper
    0
    >>> skipper(7667, 2) # no skipper
    0
    >>> skipper(414243, 2) # skipper with three 4s
    3
    >>> skipper(13232, 2) # two skippers with two 3s and two 2s
    2
    >>> skipper(88008, 4) # digits in between may be equal to skipper digit d = 8
    2
    """
    longest = 0
    while n:
        curr = n
        d = ___(a)___
        length = 0
        while ___(b)___ and curr > 0:
            length += 1
            ___(c)___
        if ___(d)___:
            ___(e)___
        n = n // 10
    return longest
```

Q4.1 (1 point) Fill in blank (a).

```
n % 10
```

Solution: Alternate solution: `curr % 10`

Q4.2 (1 point) Fill in blank (b).

```
curr % 10 == d
```

Q4.3 (2 points) Fill in blank (c).

```
curr = curr // (10 ** k)
```

Q4.4 (1 point) Fill in blank (d).

- ◯ `length > 0`
- ◯ `length > 1`
- ◯ `longest > length`
- ◯ `length > longest`
- ◯ `length > 0 and longest > length`
- ◯ `length > 0 and length > longest`
- ◯ `length > 1 and longest > length`
- ● `length > 1 and length > longest`

Q4.5 (1 point) Fill in blank (e).

```
longest = length
```

Q5 *Higher Order Minions* **(12 points)**

Q5.1 (6 points) Gru is the boss of creatures called minions. He wants a way to know who the top performing minions are using a function which will help him display the exact number of minions he asks for.

Write the function `leaderboard` which takes in a possibly infinite **generator object** (not a generator function) `next_best` and returns a function `top_minion`. `top_minion` takes in an integer `n` and outputs a list of the first `n` minions given by `next_best`. Previous calls to `top_minion` should not impact the current output.

IMPORTANT: A fully correct solution MUST use list mutation.

Even if you're not sure of your entire implementation, you may receive partial credit for certain aspects of the solution. You may not need all lines, but you may not use more than the lines provided and each line must contain up to one Python statement. Make sure you clearly indent when necessary.

```
def leaderboard(next_best):
  """
  Returns a function which outputs the first n outputs of next_best.

  >>> def minion_gen():
  ...   while True:
  ...     yield from ['Kevin', 'Stuart', 'Bob']
  >>>
  >>> top_minion = leaderboard(minion_gen())
  >>> top_minion(5)
  ['Kevin', 'Stuart', 'Bob', 'Kevin', 'Stuart']
  >>> top_minion(3)
  ['Kevin', 'Stuart', 'Bob']
  """
  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------

  ------------------------------------------------------------------------
```

Solution:

```
values = []
def top_minion(n):
  for i in range(n):
    values.append(next(next_best))
  return values[:n]
return top_minion
```

Note that the staff solution does not call `next` the minimum number of times. A more efficient solution would loop `n - len(values)` times instead.

Common errors/misconceptions with this problem:

- Not creating the list outside the helper function, resulting in previous generator elements not being saved in parent scope.
- Calling `iter(next_best)` with the intention of creating a copy of the generator. This does not work; calling `iter` on an iterator just returns a reference to the exact same iterator object that was passed in.
- Calling `list(next_best)`, because `next_best` could possibly be infinite and doing this would result in an infinite loop

Q5.2 (6 points) Kevin, Stuart, and Bob are minions trying to guess the code to the banana vault.

Write a function, `banana_vault`, which takes in a number representing the correct vault code. `banana_vault` should return a one-argument function. This returned function takes in a guessed code. If the guessed code is correct, it returns the number of incorrect guesses made before the correct guess. If the guessed code is incorrect, it returns another guessing function that remembers one additional incorrect guess has been made.

IMPORTANT: Your solution MUST NOT use lists, dictionaries, or object mutation.

Hint: Pay close attention to the inputs and outputs of the 3 functions in the skeleton below.

```
def banana_vault(n):
  """Makes a banana_vault with code n.
  >>> guess0 = banana_vault(10)
  >>> guess1 = guess0(4)
  >>> guess2 = guess1(2)
  >>> guess3 = guess2(8)
  >>> guess3(10)
  3
  >>> guess2(10)
  2
  >>> banana_vault(7)(7)
  0
  >>> b = banana_vault(5)(1)(2)(3)(4)(5)
  >>> b
  4
  """
  def update(num_incorrect):

    def guess(code):

      if ____________________________________________________________________:

        return ____________________________________________________________

      else:

        return ____________________________________________________________

    return ______________________________________________________________

  return ________________________________________________________________
```

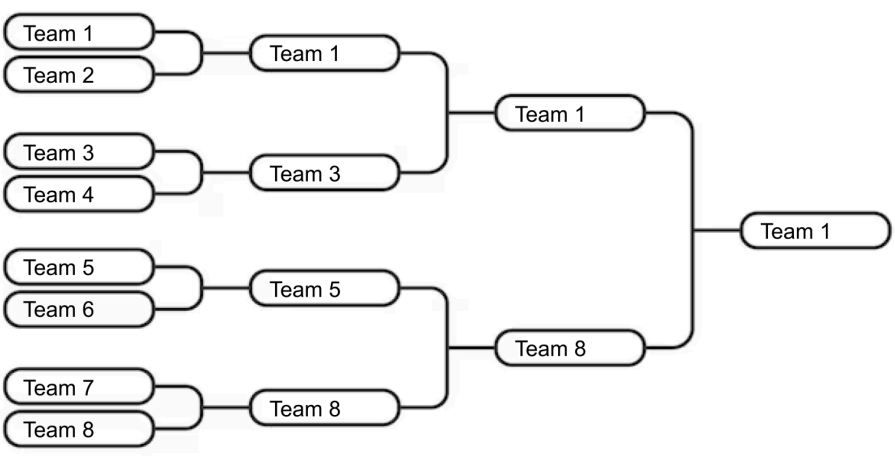
Solution:

```
def update(num_incorrect):
  def guess(code):
    if code == n:
      return num_incorrect
    else:
      return update(num_incorrect + 1)
  return guess
return update(0)
```

SID: _____

Q6 *Bracket Breaker* **(11 points)**

It's the World Cup and soccer/football teams from around the world are competing in a tournament! The results of a tournament can be represented in a *bracket*, such as the one pictured below. If you rotate the bracket counterclockwise 90 degrees, notice that it is a tree[1]!

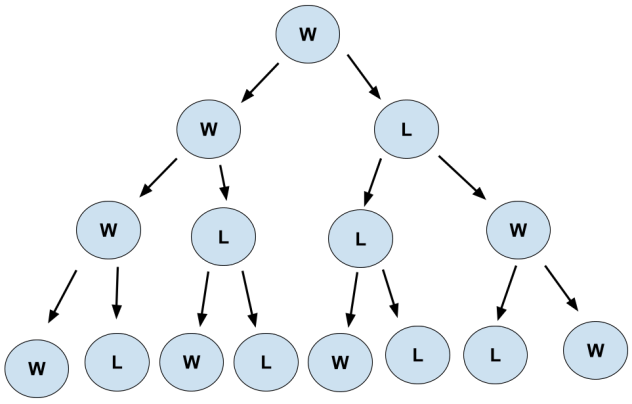


For this problem, we will represent a bracket as a tree.[2] Diagram 1 below is equivalent to the bracket pictured above, except instead of the team names, we have marked which team won (`"W"`) or lost (`"L"`) (ties are not allowed). The goal is to isolate the winning path.

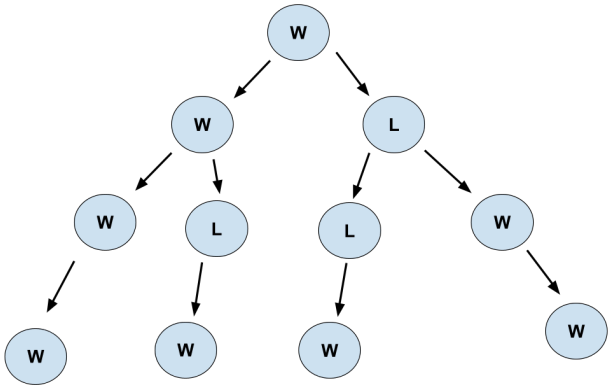
Implement `reveal_winner`, a generator function which should progressively yield simplified versions of the entire bracket. Each time `next` is called, this generator function should yield a **new tree** where every losing subtree at the **greatest depth**[3] is pruned.

Refer to the diagrams below for examples. Diagram 1 refers to the original tree passed into `reveal_winner`, while Diagrams 2-4 refer to the trees that are yielded in that order by `reveal_winner`. Calling `next` after yielding the tree in Diagram 4 should cause a `StopIteration` exception.

1



2

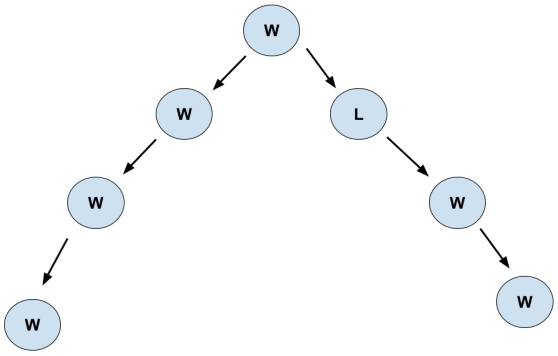


[1]Games are technically played from left to right for the bracket pictured above, but for trees in computer science, the nodes go from top to bottom, e.g. the reverse direction. Don't worry about that for this problem.

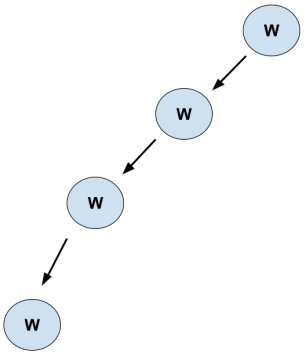
[2]The tree ADT definition can be found on your reference sheet.

[3]The *depth* of a node is its distance from the root (e.g. the root has depth 0).

3



4




```
def reveal_winner(t):
    """Doctests omitted for brevity, refer to problem description and diagrams."""
    def prune_lowest_losers(t):
        """
        Return (new_tree, changed), where:

        (1) new_tree is the pruned version of t,
            or None if t is removed
        (2) changed is True if t itself or a lowest losing subtree was pruned,
            and False otherwise
        """
        if is_leaf(t):
            if label(t) == __(a)__:
                return __(b)__, True
            return __(c)__, False

        new_branches = []
        changed = False

        for b in branches(t):
            new_b, child_changed = __(d)__
            if __(e)__:
                new_branches.__(f)__
            changed = changed or child_changed

        if label(t) == __(g)__ and __(h)__:
            return __(i)__, True

        return __(j)__

    current = t
    changed = True
    while changed:
        current, changed = prune_lowest_losers(current)
        if changed:
          yield __(k)__
```

Q6.1 (1 point) Fill in blank (a).

```
"L"
```

Q6.2 (1 point) Fill in blank (b).

● `None` ○ `tree(t)` ○ `branches(t)` ○ `tree(branches(t))`

Q6.3 (1 point) Fill in blank (c).

```
tree(label(t))
```

Q6.4 (1 point) Fill in blank (d).

```
prune_lowest_losers(b)
```

Q6.5 (1 point) Fill in blank (e).

```
new_b is not None
```

Q6.6 (1 point) Select all option(s) that could fill blank (f).

■ `append(new_b)` ☐ `extend(new_b)` ☐ `append([new_b])` ■ `extend([new_b])`

Q6.7 (1 point) Fill in blank (g).

```
"L"
```

Q6.8 (1 point) Fill in blank (h).

```
not changed
```

Q6.9 (1 point) Fill in blank (i).

```
None
```

Q6.10 (1 point) Fill in blank (j).

```
return tree(label(t), new_branches), changed
```

Q6.11 (1 point) Fill in blank (k).

○ `from current`

○ `all from current`

● `current`

○ `list(current)`

Solution: The helper function `prune_lowest_losers` returns two values: the pruned tree and whether a pruning step happened.

If the current tree is a leaf labeled `"L"`, it should be removed, so we return `None, True`. If it is a leaf labeled `"W"`, we keep it.

For a non-leaf tree, we recursively prune each branch. If a recursive call returns `None`, that branch is removed. Otherwise, the pruned branch is added to `new_branches`.

The variable `changed` tracks whether any lower losing subtree was removed. If the current node is labeled `"L"` and nothing below it changed, then this node is one of the lowest remaining losing subtrees, so we remove it.

The generator stores the current pruned tree in `current`, yields it after each successful pruning step, and stops once no more changes occur.

Q7 *Messi Iterators* **(6 points)**

In the game of soccer/football, a team is comprised of players who each have a unique jersey number. The starting lineup of a team is the subset of players who play at the very beginning of the game. Implement `starting_lineup`, which takes in:

1. A finite iterator `players` which returns unique positive integers (representing jersey numbers)
2. A non-negative integer `k`, the number of players in the lineup

It returns an iterator over the lineup which must follow these rules:

- Never include the first player in `players`.
- Maximize the number of jersey numbers that are multiples of `10` (excluding the first player).
- If there are not enough multiples of `10` to reach `k` players, fill the remaining spots with the other players, in the order they are returned by the iterator.
- Return an iterator representing the lineup.

You may assume `players` contains enough elements such that calling `starting_lineup` will produce an iterator that contains at least `k` elements.

```
def starting_lineup(players, k):
    """
    >>> players = iter([7, 10, 3, 20, 5, 30])
    >>> lineup = starting_lineup(players, 2)
    >>> next(lineup)
    10
    >>> next(lineup)
    20
    >>> next(lineup)
    StopIteration

    >>> players = iter([99, 1, 2, 10, 4])
    >>> list(starting_lineup(players, 3))
    [10, 1, 2]
    """
    tens = []
    others = []
    __(a)__
    for p in players:
        if __(b)__:
            __(c)__
        else:
            __(d)__
    lineup = __(e)__
    return __(f)__
```

Q7.1 (1 point) Fill in blank (a).

```
next(players)
```

Q7.2 (1 point) Fill in blank (b).

```
p % 10 == 0
```

Solution: A valid alternate solution is `p % 10 == 0 and len(tens) < k` if slicing is not used in blank (e) or (f).

Q7.3 (1 point) Fill in blank (c).

```
tens.append(p)
```

Q7.4 (1 point) Fill in blank (d).

```
others.append(p)
```

Q7.5 (1 point) Fill in blank (e).

```
tens + others
```

Solution: Alternatively, if blank (f) is filled with `iter(lineup)`, this blank can have `(tens + others)[:k]`.

Q7.6 (1 point) Fill in blank (f).

```
iter(lineup[:k])
```

Solution: See alternate solution in blank (e).

Solution: The first player is skipped with `next(players)`. Then we loop through the rest of the iterator and separate the players into two lists: multiples of 10 go in `tens`, and all other players go in `others`.

Since we want to prioritize multiples of 10, we concatenate the lists as `tens + others`. Finally, we only keep the first k players from this combined lineup and return an iterator over them using `iter(lineup[:k])`.

Another alternate solution is if you switch the semantic meaning of `tens` and `others` and keep everything else consistent (e.g. you store all non-multiples of 10 in `tens` and multiples of 10 in `others`).

Yet another alternate solution involves defining a boolean `passed` which tracks if the first player has been excluded, and then using the ternary operator to handle everyone except the first player:

```
def starting_lineup(players, k):
  tens = []
  others = []
  passed = False
  for p in players:
      if not passed:
          passed = True
      else:
          tens.append(p) if not p % 10 else others.append(p)
  lineup = tens + others
  return iter(lineup[:k])
```

Q8 *Just for fun!* **(0 points)**

Q8.1 Draw something fun, or write a message for the staff! This is also the place where you can report any potential academic misconduct you witnessed during the exam (please include as much detail as possible). All reports will be kept anonymous.