# SQL & Final Review Meta

## Computer Science Mentors 61A

November 28–December 9, 2022

This worksheet is to be used in both week 15 (SQL) and week 16 (Final Review). Please save this worksheet after week 15's section and remember to bring it for week 16.

This is a bit of an unusual worksheet because it covers not one but two weeks. The SQL section should be covered in week 15, while the remaining topics of final review should be covered during week 16. Students should make sure to keep the worksheet in the intervening time so that they can follow along with the section.

On the other hand, if I were a mentor, I would not even use the latter portion of this worksheet for week 16. Instead, I would choose random exam-level problems to help students prep.

If you finish the SQL portion early, of if your students do not feel it is necessary, you can skip to final review during week 15.

**Recommended Timeline (for week 15)**

- SQL minilecture: 10 minute
- Q1 (Fish): 20 minutes
- Q2 (Outfits): 20 minutes

## 1 SQL

SQL (Structured Query Language) is a declarative programming language that allows us to store, access, and manipulate data stored in databases. Each database contains tables, which are rectangular collections of data with rows and columns. This section gives a brief overview of the small subset of SQL used by CS 61A; the full language has many more features.

### 1.1 Creating Tables

#### 1.1.1 SELECT

**SELECT** statements are used to create tables. The following creates a table with a single row and two columns:

```
sqlite> SELECT "Adit" AS first, "Balasubramanian" AS last;
Adit|Balasubramanian
```

Created by Gabe Classon, Aditya Balasubramanian, Alyssa Smith, Esther Shen, Maya Romero, and Manas Khatore

**AS** is an "aliasing" operation that names the columns of the table. Note that built-in keywords such as **AS** and **SELECT** are capitalized by convention in SQL. However, SQL is case insensitive, so we could just as easily write **as** and **select**. Also, each SQL query must end with a semicolon.

### 1.1.2 UNION

**UNION** joins together two tables with the same number of columns by "stacking them on top of each other". The column names of the first table are kept.

```
sqlite> SELECT "Adit" AS first, "Balasubramanian" AS last UNION
...> SELECT "Gabe", "Classon";
Adit|Balasubramanian
Gabe|Classon
```

### 1.1.3 CREATE TABLE

To create a named table (so that we can use it again), the **CREATE TABLE** command is used:

```
CREATE TABLE scms AS
    SELECT "Adit" AS first, "Balasubramanian" AS last UNION
    SELECT "Gabe", "Classon";
```

It is nice to note that SQL syntax is supposed to mirror English grammar closely so that it is natural to understand.

The remaining examples will use the following `team` table:

```
CREATE TABLE team AS
    SELECT "Gabe" AS name, "cat" AS pet, 11 AS birth_month UNION
    SELECT "Adit",         "none",       10 UNION
    SELECT "Alyssa",       "dog",         4 UNION
    SELECT "Esther",       "dog",         6 UNION
    SELECT "Maya",         "dog",         3 UNION
    SELECT "Manas",        "none",       11;
```

## 1.2 Manipulating other tables

We can also write **SELECT** statements to create new tables from other tables. We write the columns we want after the **SELECT** command and use a **FROM** clause to designate the source table. For example, the following will create a new table containing only the `name` and `birth_month` columns of `team`:

```
sqlite> SELECT name, birth_month FROM team;
Adit|10
...
Maya|3
```

Note that the order in which rows are returned is undefined.

An asterisk * selects for all columns of the table:

```
sqlite> SELECT * FROM team;
Adit|none|10
...
Maya|dog|3
```

This is a convenient way to view all of the content of a table.

We may also manipulate the table columns and use **AS** to provide a (new) name to the columns of the resulting table. The following query creates a table with each teammate's name and the number of months between their birth month and June:

```
sqlite> SELECT name, ABS(birth_month - 6) AS june_dist FROM team;
Adit|4
...
Maya|3
```

### 1.2.1 WHERE

**WHERE** allows us to filter rows based on certain criteria. The **WHERE** clause contains a boolean expression; only rows where that expression evaluates to true will be kept.

```
sqlite> SELECT name FROM team WHERE pet = "dog";
Alyssa
Esther
Maya
```

Note that = in SQL is used for equality checking, not assignment.

### 1.2.2 ORDER BY

**ORDER BY** specifies a value by which to order the rows of the new table. **ORDER BY ...** may be followed by **ASC** or **DESC** to specify whether they should be ordered in ascending or descending order. **ASC** is default. For strings, ascending order is alphabetical order.

```
sqlite> SELECT name FROM team WHERE pet = "dog" ORDER BY name DESC;
Maya
Esther
Alyssa
```

## 1.3 Joins

Sometimes, you need to compare values across two tables—or across two rows of the same table. Our current tools do not allow for this because they can only consider rows one-by-one. A way of solving this problem is to create a table where the rows consist of every possible combination of rows from the two tables; this is called an **inner join**. Then, we can filter through the combined rows to reveal relationships between rows. It sounds bizarre, but it works.

An inner join is created by specifying multiple source tables in a **WHERE** clause. For example, `SELECT * FROM team AS a, team AS b;` will create a table with 36 rows and 6 columns. The table has 36 rows because each row represents one of 36 possible ways to select two rows from `team` (where order matters). The table has 6 columns because the joined tables have 3 columns each. We use **AS** to give the two source tables different names, since we are joining `team` to itself. The columns of the resulting table are named `a.name, a.pet, a.birth_month, b.name, b.pet, b.birth_month`.

For example, to determine all pairs of people with the same birth month, we can use an inner join:

```
sqlite> SELECT a.name, b.name FROM team AS a, team AS b WHERE a.name < b.name
   AND a.birth_month = b.birth_month;
Gabe|Manas
```

We include `a.name < b.name` to ensure that each pair of people is only listed once. Otherwise, we would get both `Gabe|Manas` and `Manas|Gabe`.

## 1.4 Aggregation

Aggregation uses information from multiple rows in our table to create a single row. Using an aggregation function such as **MAX**, **MIN**, **COUNT**, and **SUM** will automatically aggregate the table data into a single row. For example, the following will collapse the entire table into one row containing the name of the person with the latest birth month:

```
sqlite> SELECT name, MAX(birth_month) FROM team;
Manas|11
```

Note that there are multiple rows with the largest birth month. When this happens, SQL arbitrarily chooses one of the rows to use.

The **COUNT** aggregation function collapses the table into one row containing the number of rows in the table:

```
sqlite> SELECT COUNT(*) FROM team;
6
```

### 1.4.1 GROUP BY

**GROUP BY** groups together all rows with the same value for a particular column. Aggregation is performed on each group instead of on the entire table. There is then *exactly one row* in the resulting table for each group. As before, type of aggregation performed is determined by the choice of aggregation function. The following gives, for each type of pet, the information of the person with the earliest birth month who has that pet:

```
sqlite> SELECT name, pet, MIN(birth_month) FROM team GROUP BY pet;
Gabe|cat|11
Maya|dog|3
Adit|none|10
```

I like to emphasize to my students that there is always exactly one row in the resulting table for each group.

### 1.4.2 HAVING

Just as **WHERE** filters out rows, **HAVING** filters out groups. For example, the following selects for all types of pets owned by more than one teammate:

```
sqlite> SELECT pet FROM team GROUP BY pet HAVING COUNT(*) > 1;
dog
none
```

## 1.5 Syntax

The clauses of a **SELECT** statement always come in this order:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ...;
```

The order roughly reflects the order in which the processing steps are applied. Note that all filtering of rows comes *before* aggregation. That is, aggregation is always performed after the row-by-row filtering is complete.

## Teaching Tips

- Make sure to emphasize the difference between **`HAVING`** and **`WHERE`**. Students are often confused by the similarity of these two clauses.
- `code.cs61a.org` has an interactive SQL terminal that lets you see visually the rows you extract from a given query and also the logical order in which queries are processed. This is a super useful tool.
- When I was learning about SQL, I had many confusions about edge cases. When you use `code.cs61a.org`, try asking your students if there is anything they would like you to type in the interpreter to see how it handles things.

1. CS 61A wants to start a fish hatchery, and we need your help to analyze the data we've collected for the fish populations! Running a hatchery is expensive—we'd like to make some money on the side by selling some seafood (only older fish of course) to make delicious sushi.

   The table `fish` contains a subset of the data that has been collected. The SQL column names are listed in brackets.

   `fish`

| Species [species] | Population [pop] | Breeding Rate [rate] | $/piece [price] | # of pieces per fish [pieces] |
|---|---|---|---|---|
| Salmon | 500 | 3.3 | 4 | 30 |
| Eel | 100 | 1.3 | 4 | 15 |
| Yellowtail | 700 | 2.0 | 3 | 30 |
| Tuna | 600 | 1.1 | 3 | 20 |

   (a) Write a query to find the three most populated fish species.

```
SELECT species FROM fish ORDER BY pop DESC LIMIT 3;
```

   (b) Write a query to find the total number of fish in the ocean. Additionally, include the number of species we summed. Your output should have the number of species and the total population.

```
SELECT COUNT(species), SUM(pop) FROM fish;
```

   (c) Profit is good, but more profit is better. Write a query to select the species that yields the most number of pieces for each price. Your output should include the species, price, and pieces.

```
SELECT species, price, MAX(pieces) FROM fish GROUP BY price;
```

   (d) Write a query that returns, for each species, the difference between our hatchery's revenue versus the competitor's revenue for one whole fish.

```
SELECT fish.species, (fish.price - competitor.price) * pieces
    FROM fish, competitor
    WHERE fish.species = competitor.species;
```

   **Teaching Tips**

   - Make sure students know the basics of understanding/looking through a table
     - It may help to write a basic example using SELECT, FROM, and WHERE
   - Have students clearly define which columns they will need from the table (species, price, pieces) before coding
   - Remind students they are able to do arithmetic on cells they select
   - While not in the solution, you can use this problem to explain the benefits of aliasing
     - Show students they can write `FROM fish as __, competitor as __`

2. In this question, you have access to two tables.

   **Grades**, which contains three columns: **day**, class, and score. Each row represents the score you got on a midterm for some class that you took on some **day**.

   **Outfits**, which contains two columns: **day** and color. Each row represents the color of the shirt you wore on some **day**. Assume you have a row for each possible day.

outfits

| Day | Color |
|---|---|
| 11/5 | Blue |
| 9/13 | Red |
| 10/31 | Orange |

grades

| Day | Class | Score |
|---|---|---|
| 10/31 | Music 70 | 88 |
| 9/20 | Math 1A | 72 |

   (a) Instead of actually studying for your finals, you decide it would be the best use of your time to determine what your "lucky shirt" is. Suppose you're pretty happy with your exam scores this semester, so you define your lucky shirt as the shirt you wore to the most exams.

   Write a query that will output the color of your lucky shirt and how many times you wore it.

```
SELECT color, count(g.day) AS cnt
    FROM outfits AS o, grades AS g
    WHERE o.day = g.day
    GROUP BY color
    ORDER BY cnt DESC
    LIMIT 1;
```

### Teaching Tips

- Ensure students know all potential parts of a SQL query and how they work:
  **SELECT** [ ] **FROM** [ ] **WHERE** [ ] **GROUP BY** [ ] **HAVING** [ ] **ORDER BY** [ ] **LIMIT** [ ];
- Remind students about the **SUM**, **MIN**, and **COUNT** functions
- Students may struggle with the variable to use in **ORDER BY**
  - Teach students they can alias created columns in **SELECT**
  - In SQL **SELECT** occurs before **ORDER BY**, so cnt can be used

(b) You want to find out which classes you need to prepare for the most by determining how many points you have so far. However, you only want to do so for classes where you did relatively poorly.

Write a query that will output the sum of your midterm scores for each class along with the corresponding class, but only for classes in which you scored less than 80 points on at least one midterm. List the output from highest to lowest total score.

```
SELECT SUM(score), class
    FROM grades GROUP BY class
    HAVING MIN(score) < 80 ORDER BY SUM(score) DESC;
```

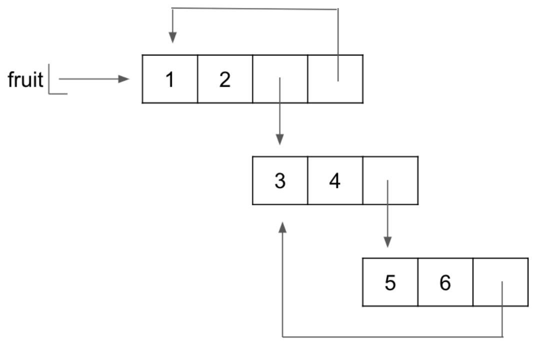
**Teaching Tips**

- Remind students that the '**HAVING**' clause is used to filter out groups (you cannot use **WHERE** on the groups).
- Remind students that aggregation works on each group individually; this means we want to **GROUP BY** the things we want to **SUM** over.

## 2 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
```

**Teaching Tips**

- So many nested lists!! Be very clear when you are explaining these concepts to your students and when you are drawing as it's very easy to get confused.
- Review with students how to use arrows in list diagrams, such as shallow and deep copies, values vs. references.
- The best way to approach this problem is to just go line by line, counting the indices as you go. Remember to make the distinction between reassigning what an arrow in a box points to and updating the value of the box itself (to a value or to another arrow). (This is also good practice for 61B)
- If your students get stuck, a good hint would be to tell them that they do indeed need all the blanks. Do not try to squeeze too many operations into one line.

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).

   (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.

**Teaching Tips**

- For a reminder on Tree paths, it can help to start with the all paths function:

```
def all_paths(t):
    if t.is_leaf():
        return [[t.label]]
    paths = []
    for b in t.branches:
        for path in all_paths(b):
            paths.append([t.label] + path)
    return paths
```

- From there, it becomes a much simpler matter of modifying two things:
  - Making the function a generator so it yields paths one at a time instead of returning a list of paths
  - Only returning non-decreasing paths

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))
```

This is a hard question! Only do it if your students are absolutely assured in their definition of data abstraction and the abstraction barrier.

Feel free to spend even more time on this. Lists are more important for students generally, but understanding data abstraction deeply is a great setup for OOP!

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 $\Theta(n^2)$.

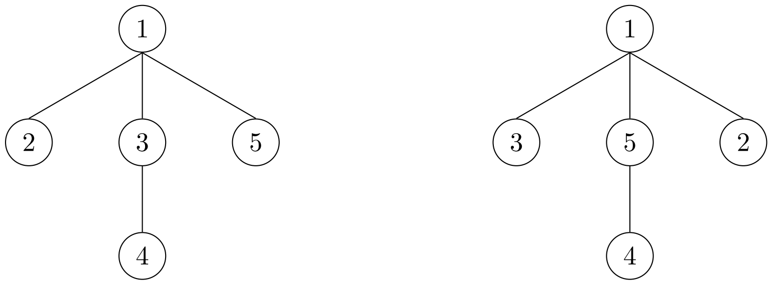
5. Implement the classes so the following code runs.

```
"""
>>> p = Plant()
>>> p.height
1
>>> p.materials
[]
>>> p.absorb()
>>> p.materials
[|Sugar|]
>>> Sugar.sugars_created
1
>>> p.leaf.sugars_used
0
>>> p.grow()
>>> p.materials
[]
>>> p.height
2
>>> p.leaf.sugars_used
1
"""
```

```
class Plant:
    def __init__(self):
        """A Plant has a Leaf, a list of sugars created so far,
        and an initial height of 1.
        """
        self.leaf = Leaf(self)
        self.materials = []
        self.height = 1

    def absorb(self):
        """Calls the Leaf to create sugar."""
        self.leaf.absorb()

    def grow(self):
        """A Plant consumes all of its sugars to grow, each of which
        increases its height by 1.
        """
        for sugar in self.materials:
            sugar.activate()
            self.height += 1

class Leaf:
    def __init__(self, plant): # plant is a Plant instance
        """A Leaf is initially alive, and keeps track of how many
        sugars it has created.
        """
        self.alive = True
        self.sugars_used = 0
        self.plant = plant

    def absorb(self):
        """If this Leaf is alive, a Sugar is added to the plant's
        list of sugars.
        """
        if self.alive:
            self.plant.materials.append(Sugar(self, self.plant))


    def __repr__(self):
        return '|Leaf|'

class Sugar:
    sugars_created = 0

    def __init__(self, leaf, plant):
        self.leaf = leaf
        self.plant = plant
        Sugar.sugars_created += 1

    def activate(self):
        """A sugar is used."""
        self.leaf.sugars_used += 1
        self.plant.materials.remove(self)

    def __repr__(self):
        return '|Sugar|'
```

6. 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 ______________________________________:

        ______________________________________________

        ______________________________________________

        ______________________________________________
```

```
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)
```

**Teaching Tips**

- As with most other tree problems, annotating the given examples and drawing out examples of your own will be a big help for students. Make sure they really understand the method of rotation and the rules that the problem establishes.
- Remind your students to pay close attention to the data types of whatever they are working on. For example, do they need to create a new Tree? How can they traverse across the branches? How can they access the value of a node?
- The second line in the for loop may be hard to get because of the modulo. Try to think of an example where this modulo would apply, draw it out, and see if your students catch it.
- Be sure to highlight the distinction between nondestructive and destructive recursive methods and point out the key differences in implementing each type of function.
- Since there isn't an if/else format for base cases vs. recursive case, it may be harder for students to understand what is going on in the problem. Try to break it down into several steps for them to guide them through each line.

7. 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).

```
;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)))
      )
    )
  )
)
```

## 9 Tail Recursion

8. Write a tail recursive function, `skip-list`, that takes in a potentially nested list `lst` and a filter function `filter-fn`, goes through each element in order, and returns a new list that contains all elements that pass the `filter-fn`. The returned list is *not nested*.

   Hint: `pair?` is a predicate procedure that returns true if its argument is a Scheme list and false otherwise.

```
;Doctests
scm> (skip-list '(1 (3)) even?)
()
scm> (skip-list '(1 (2 (3 4) 5) 6 (7) 8 9) odd?)
(1 3 5 7 9)

(define (skip-list lst filter-fn)
  (define (skip-list-tail _________ _________ next)

    (cond
        ((null? lst) (if (null? _________)
                         ________________________________________

                         ________________________________________))

        ((pair? ________) (________________________________________))

        ((________________) ________________________________________)

        (else ____________________________________)
    )
  )
  (skip-list-tail ____________ ____________ ____________)
)
```

```
(define (skip-list lst filter-fn)
  (define (skip-list-tail lst lst-so-far next)
    (cond
      ((null? lst) (if (null? next)
                       lst-so-far
                       (skip-list-tail (car next) lst-so-far (cdr next))))
      ((pair? (car lst))
        (skip-list-tail (car lst)
                        lst-so-far
                        (cons (cdr lst) next)))
      ((filter-fn (car lst))
        (skip-list-tail (cdr lst)
                       (append lst-so-far (list (car lst)))
                       next))
      (else (skip-list-tail (cdr lst) lst-so-far next)))
    )
  (skip-list-tail lst nil nil)
)
```

## 10 Macros

9. (Spring 2018 Final)
Implement `lambda-macro`, a macro that creates anonymous macros. A lambda-macro expression has a list of formal parameters and one body expression. It creates a *macro* with those formal parameters and that body. Assume that the symbol anon is not use anywhere else in a program that contains lambda-macro.

```
;Doctests
scm> (define mac (lambda-macro (x) `(begin ,x ,x)))
mac
scm> (mac (print 1))
1
1
(define-macro (lambda-macro bindings body)

      `(begin (________________________________

               ________________________________

               ________________________________)

         anon))
```

```
`(begin
   (define-macro ,(cons 'anon bindings) ,body)
   anon))
```

10. Define a macro, `eval-and-check` that takes in three expressions and evaluates each expression in order. If the last expression evaluates to a truth-y value, return the symbol ok. Otherwise, return fail.

```
;Doctests
scm> (eval-and-check #f #f #t)
ok
scm> (eval-and-check (+ 2 3) (print 2) (> 2 3))
2
fail
scm> (eval-and-check (define x 1) (print x) (> x 0))
1
ok
```

```
(define-macro (eval-and-check expr1 expr2 expr3)

     ________________________________________

     ________________________________________

     ________________________________________)
```

```
(define-macro (eval-and-check expr1 expr2 expr3)
      `(if (begin ,expr1 ,expr2 ,expr3)
            'ok
            'fail))
```

**Teaching Tips**

- If they offer a solution using `cdr`, remind them of the **`begin`** function, which returns the last expression.
- Remind them of the need to quote `ok` and `fail`.