Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = ""
COLLABORATORS = ""
```

Homework 3: Recursion

CSE 30 Spring 2022

Instructions

The Format of a Python Notebook

This is a Python Notebook homework. It consists of various types of cells:

- Text: you can read them :-)
- Code: you should run them, as they may set up the problems that you are asked to solve.
- **Solution:** These are cells where you should enter a solution. You will see a marker in these cells that indicates where your work should be inserted.

```
    # YOUR CODE HERE
```

- Test: These cells contains some tests, and are worth some points. You should run the cells as a way to debug your code, and to see if you understood the question, and whether the output of your code is produced in the correct format. The notebook contains both the tests you see, and some secret ones that you cannot see. This prevents you from using the simple trick of hard-coding the desired output.

Running your notebook

Running a cell. To run a cell of the notebook, either click on the icon to its top left, or press shift-ENTER (or shift-Return).

Disconnections. When you open a notebook, Google automatically connects a server to the web page, so that you can type code in your browser, and the code is run on that server. If you are idle for more than a few minutes, Google keeps all you typed (none of your work is lost), but the server may be disconnected due to inactivity. When the server is disconnected, it loses all memory of anything you have defined (functions, classes, variables, etc).

If you do get disconnected, select Runtime > Run All (or Runtime > Run before) to ensure everything is defined as it should.

DO NOT

- **Do not add, delete, reorder, remove cells.** This breaks the relationship between your work, and the grading system, making it impossible to grade your work.

Debugging

To debug, you can add print statements to your code. They should have no effect on the tests. Just be careful that if you add too many of them inside loops and similar, you may cause for some of the tests we will do such an enormous amount of output that grading might timeout (and you may not get credit for an answer).

Asking for help

The tutors and TAs should have access to the notebook; otherwise, you can always share a link with them. In this way, they can take a look at your work and help you with debugging and with any questions you might have.

Submitting Your Notebook

To submit:

- **Check your work.** Before submitting, select Runtime > Restart and Run All, and check that you don't get any unexpected error.
- **Download the notebook.** Click on File > Download .ipynb . **Do not download the .py file.**
- **Upload.** Upload the .ipynb file to **this Google form**.
- This homework is due by 11:59pm on **Tuesday, 12 April 2022**.

The Test

There are three questions in this assignment, for a total of 40 points.

In [ ]:
```
# Let me define the function I use for testing.  Don't change this cell.

def check_equal(x, y, msg=None):
    if x == y:
        if msg is None:
            print("Success")
        else:
            print(msg, ": Success")
    else:
        if msg is None:
            print("Error:")
        else:
            print("Error in", msg, ":")
        print("    Your answer was:", x)
        print("    Correct answer: ", y)
    assert x == y, "%r and %r are different" % (x, y)
```

Question 1: Sum of Leaves

Consider a binary tree defined as in the book chapter, where a node can be either a number (an integer or floating-point number), or a tuple consisting of a left subtree, and a right subtree. Examples of trees are:

```
4.5
(3, 5)
(3, (5, 7))
((3, 4), ((4, 2), 9))
```

Write a function `total(t)` that takes the tree `t`, and returns the total of all the leaves in the tree.

In [ ]:
```
def total(t):
    if type(t) == tuple:
      (x, y) = t
      sumx = total(x)
      sumy = total(y)
      return sumx + sumy
    else:
      sumt = t
      return sumt
```

In [ ]:
```
# This is a place where you can write additional tests to help you test
# your code, or debugging code, if you need.  You can also leave it blank.

total(((3, 5), 8))
```

Out[ ]: 16

Here are some tests for your code.

In [ ]:
```
### 10 points.  Simple tests.

check_equal(total(5.4), 5.4)
check_equal(total((4, 6)), 10)
```

```
Success
Success
```

In [ ]:
```
### 10 points.  Tests on trees.

check_equal(total((3, (4, (5, 6)))), 18)
check_equal(total((((3, 4), 5), ((4, 3), (2, 1)))), 22)
```

```
Success
Success
```

Question 2: Largest Subtree Total

For a tree $t = (t_1, t_2)$, the subtrees of t consist of t, and of the subtrees of t_1 and t_2. That is, in formulas,

$$\text{subtrees}(t) = \{t\} \cup \text{subtrees}(t_1) \cup \text{subtrees}(t_2).$$

To give a concrete example, the subtrees of $t = (3, (4, 5))$ are:

- $(3, (4, 5))$
- 3
- $(4, 5)$
- 4
- 5

Given a tree `t`, write a function `max_subtree_total(t)` that computes the largest total of any subtree in `t`.

Of course, if the numbers in the tree are all non-negative, then the largest total corresponds to the complete tree, but this is not the case if there are leaves with value smaller than 0. For instance, for the tree

```
(-2, (3, 4))
```

the largest total corresponds to the subtree `(3, 4)`, which has total 7.

If you wish, you can use the `total` function you developed in the previous question, but you don't have to.

In [ ]:
```
def max_subtree_total(t):
    if type(t) == tuple:
      (t1, t2) = t
      sumt = total(t)
      largestsumt1 = max_subtree_total(t1)
```

```
      largestsumt2 = max_subtree_total(t2)
      return max(sumt, largestsumt1, largestsumt2)
    else:
      sumt = total(t)
      return sumt
```

In [ ]:
```
# This is a place where you can write additional tests to help you test
# your code, or debugging code, if you need.  You can also leave it blank.

max_subtree_total((-2, (3, 4)))
```

Out[ ]: 7

In [ ]:
```
### Some simple cases first.

check_equal(max_subtree_total(3), 3)
check_equal(max_subtree_total((4, 5)), 9)
```

```
Success
Success
```

In [ ]:
```
### 10 points.  More complex cases.

check_equal(max_subtree_total((-3, 4)), 4)
check_equal(max_subtree_total((-3, -4)), -3)
check_equal(max_subtree_total(((-3, 5), (2, 1))), 5)
check_equal(max_subtree_total((((4, 5), -3), (12, (3, 4)))), 25)
```

```
Success
Success
Success
Success
```

Question 3

Consider a list, for instance, `[3, 4, 5, 3, 2, 5, 6, 4]`. A *subsequence* or *sublist* of a list is a list you obtain by removing zero or more elements from the original list. For example, subsequences of the above list are:

```
[]
[4, 5, 6]
[3, 4, 5, 5, 6]
[3, 5, 2]
```

A *nondecreasing* subsequence is a sequence in which a value is not smaller than any of the previous ones. In the examples above, the first three subsequences are nondecreasing; the last one is not, as the last `2` is smaller than `3` and `5`.

Your task is to write a function `nondecsub(l)` which, given a list `l`, returns a list consisting of *all* the nondecreasing subsequences of `l`.

Hint: It is easier to write a function `nondecsub(l, threshold=None)`, and we have set up the problem for you in this way. The idea is that `threshold` represents a threshold, below which you should not select elements from `l`.

Remember: your goal is to reduce the problem to one that is smaller, for example, one that involves a list that is shorter by one. Look at how the code for permutations was done. Reduce your problem to one that is smaller, and use the `threshold` parameter to ensure that the generated subsequences are nondecreasing.

In [ ]:
```
def nondecsub(l, threshold=None):
    # YOUR CODE HERE
    """maxlist = []
    for i in range(len(l) + 1):
      for j in range(i):
        threshold = None
        templist = l[j: i]
        if threshold == None:
            threshold = templist[0]"""
   if len(l) == 0:
     yield

def store(l):
  full_list = l

# This code helps in testing, as it transforms a list of lists into a set of tuples

def normalize(ll):
    s = set()
    for l in ll:
        s.add(tuple(l))
    return s

def n(l):
    return normalize(nondecsub(l, threshold=None))
```

In [ ]:
```
# This is a place where you can write additional tests to help you test
# your code, or debugging code, if you need.  You can also leave it blank.

# YOUR CODE HERE
nondecsub([4])
```

Out[ ]:
```
[[4]]
```

In [ ]:
```
### 10 points: simple tests

check_equal(n([4]), normalize([[], [4]]))

check_equal(n([]), normalize([[]]))

check_equal(n([3, 4]), normalize([
    [], [3], [4], [3, 4]
]))
```

```
check_equal(n([4, 3]), normalize([
    [], [3], [4]
]))
```

In [ ]:
```
### 10 points: more complicated tests

check_equal(n([-1, 0, 3, 4, 3, 5]), normalize([
    (3, 4, 5), (0, 4, 5), (-1, 0), (-1, 0, 3, 3, 5), (-1, 0, 3, 4),
    (-1,), (-1, 5), (0, 3), (3, 3), (-1, 3, 3), (3,), (-1, 3, 4),
    (3, 3, 5), (-1, 4, 5), (-1, 0, 3, 4, 5), (-1, 0, 3, 5),
    (-1, 3, 3, 5), (-1, 3, 5), (0, 4), (5,), (-1, 0, 3, 3),
    (-1, 0, 3), (0, 3, 3), (0, 3, 3, 5), (-1, 0, 4, 5), (4, 5), (-1, 0, 5),
    (0, 5), (-1, 0, 4), (3, 5), (0,), (0, 3, 4), (-1, 3), (0, 3, 5), (4,),
    (), (0, 3, 4, 5), (-1, 3, 4, 5), (-1, 4), (3, 4)
]))
```