# Instructions

> Please download homework materials `hw04.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 3. The starter code for these problems is provided in `hw04.py`, `ADT.py` and `shakespeare.py`, which is distributed as part of the homework materials in the `code` directory.

We have also prepared two optional problems just for fun in section 3. You can find further descriptions there.

**Submission**: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

**Readings**: You might find the following references to the textbook useful:

- Section 2.2
- Section 2.3

# Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website.

Remember, you can use `ok` to test your code:

```
$ python ok  # test all functions
$ python ok -q <func>  # test single function
```

# Problem 1: Couple (100pts)

Implement the function `couple`, which takes in two lists and returns a list that contains lists with i-th elements of two sequences coupled together. You can assume the lengths of two sequences are the same. Try using a list comprehension.

```
def couple(lst1, lst2):
    """Return a list that contains lists with i-th elements of two sequences
    coupled together.
    >>> lst1 = [1, 2, 3]
    >>> lst2 = [4, 5, 6]
    >>> couple(lst1, lst2)
    [[1, 4], [2, 5], [3, 6]]
    >>> lst3 = ['c', 6]
    >>> lst4 = ['s', '1']
    >>> couple(lst3, lst4)
    [['c', 's'], [6, '1']]
    """
    assert len(lst1) == len(lst2)
    "*** YOUR CODE HERE ***"
```

# Problem 2: City Data Abstraction (100 + 100 pts)

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our ADT has one **constructor**:

- `make_city(name, lat, lon)`: Creates a city object with the given name, latitude, and longitude.

We also have the following selectors in order to get the information for each city:

- `get_name(city)`: Returns the city's name
- `get_lat(city)`: Returns the city's latitude
- `get_lon(city)`: Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
>>> nanjing = make_city('Nanjing', 31, 118)
>>> get_name(nanjing)
'Nanjing'
>>> get_lat(nanjing)
31
>>> beijing = make_city('Beijing', 39, 116)
>>> get_lon(beijing)
116
```

All of the selector and constructor functions can be found in the lab file, if you are curious to see how they are implemented. However, the point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

# Problem 2.1 : Distance (100 pts)

We will now implement the function distance, which computes the distance between two city objects. Recall that the distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the sqrt of `(x1 - x2)**2 + (y1 - y2)**2`. We have already imported `sqrt` for your convenience. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

```
from math import sqrt
def distance(city1, city2):
    """
    >>> city1 = make_city('city1', 0, 1)
    >>> city2 = make_city('city2', 0, 2)
    >>> distance(city1, city2)
    1.0
    >>> city3 = make_city('city3', 6.5, 12)
    >>> city4 = make_city('city4', 2.5, 15)
    >>> distance(city3, city4)
    5.0
    """
    "*** YOUR CODE HERE ***"
```

# Problem 2.2 : Closer city (100 pts)

Next, implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the distance function you just defined for this question.

```
def closer_city(lat, lon, city1, city2):
    """
    Returns the name of either city1 or city2, whichever is closest to
    coordinate (lat, lon).

    >>> berkeley = make_city('Berkeley', 37.87, 112.26)
    >>> stanford = make_city('Stanford', 34.05, 118.25)
    >>> closer_city(38.33, 121.44, berkeley, stanford)
    'Stanford'
    >>> bucharest = make_city('Bucharest', 44.43, 26.10)
    >>> vienna = make_city('Vienna', 48.20, 16.37)
    >>> closer_city(41.29, 174.78, bucharest, vienna)
    'Bucharest'
    """
    "*** YOUR CODE HERE ***"
```

# Problem 3: Trees (100 + 100 + 200 + 100 + 100 pts)

Recall that the `tree` abstract data types mentioned in class, can be recursively defined using a root node and (sub)trees.

Tree ADT has one **constructor**:

- `tree(label, branches=[])`: Creates a tree with the given label and branches.

We also have the following default selectors in order to get the information for each tree:

- `label(tree)`: returns the `tree`'s label.
- `branches(tree)`: returns the `tree`'s branches.
- `is_leaf(object)`: returns the if a `tree` is a leaf.

# Problem 3.1 : Nut Finder (100 pts)

The squirrels on campus need your help! There are a lot of trees on campus and the squirrels would like to know which ones contain nuts. Define the function `nut_finder`, which takes in a tree and returns `True` if the tree contains a node with the value `'nut'` and `False` otherwise.

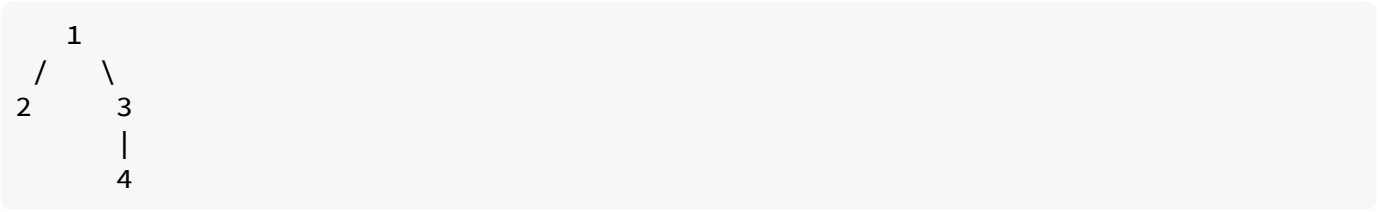
```
def nut_finder(t):
    """Returns True if t contains a node with the value 'nut' and
    False otherwise.

    >>> scrat = tree('nut')
    >>> nut_finder(scrat)
    True
    >>> sproul = tree('roots', [tree('branch1', [tree('leaf'), tree('nut')]),
tree('branch2')])
    >>> nut_finder(sproul)
    True
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6,
[tree(7)])])
    >>> nut_finder(numbers)
    False
    >>> t = tree(1, [tree('nut',[tree('not nut')])])
    >>> nut_finder(t)
    True
    """
    "*** YOUR CODE HERE ***"
```

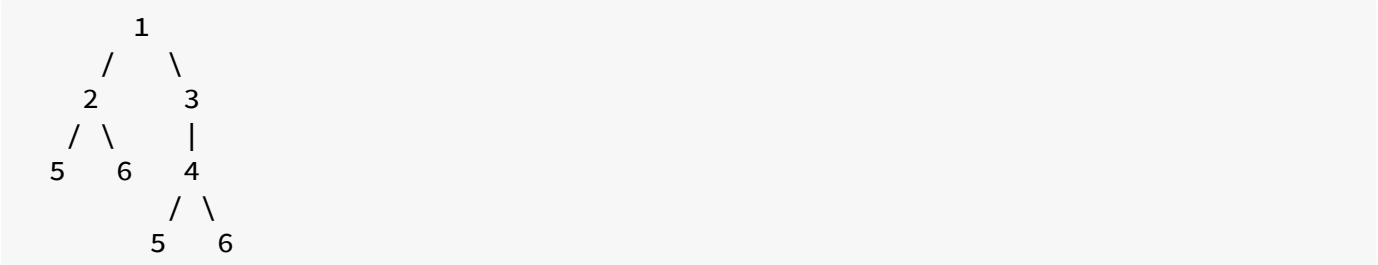
# Problem 3.2: Sprout leaves (100 pts)

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of values, `values`. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each value in `values`.

For example, say we have the tree `t = tree(1, [tree(2), tree(3, [tree(4)])])`:




If we call `sprout_leaves(t, [5, 6])`, the result is the following tree:



```
def sprout_leaves(t, values):
    """Sprout new leaves containing the data in values at each leaf in
    the original tree t and return the resulting tree.

    >>> t1 = tree(1, [tree(2), tree(3)])
    >>> print_tree(t1)
    1
      2
      3
    >>> new1 = sprout_leaves(t1, [4, 5])
    >>> print_tree(new1)
    1
      2
        4
        5
      3
        4
        5

    >>> t2 = tree(1, [tree(2, [tree(3)])])
    >>> print_tree(t2)
    1
      2
        3
    >>> new2 = sprout_leaves(t2, [6, 1, 2])
    >>> print_tree(new2)
    1
      2
        3
          6
          1
          2
    """
    "*** YOUR CODE HERE ***"
```

# Problem 3.3: Add trees (200 pts)

Define the function `add_trees`, which takes in two trees and returns a new tree where each corresponding node from the first tree is added with the node from the second tree. If a node at any particular position is present in one tree but not the other, it should be present in the new tree as well.

```
def add_trees(t1, t2):
    """
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(add_trees(numbers, numbers))
    2
      4
        6
        8
      10
        12
          14
        16
    >>> print_tree(add_trees(tree(2), tree(3, [tree(4), tree(5)])))
    5
      4
      5
    >>> print_tree(add_trees(tree(2, [tree(3)]), tree(2, [tree(3), tree(4)])))
    4
      6
      4
    >>> print_tree(add_trees(tree(2, [tree(3, [tree(4), tree(5)])]), \
    tree(2, [tree(3, [tree(4)]), tree(5)])))
    4
      6
        8
        5
      5
    """
    "*** YOUR CODE HERE ***"
```

# Problem 3.4: Big Path (100 pts)

A *path* through a tree is a list of adjacent node values that starts with the root value. Paths can end at any node but they must always go from one node to one of its branches and may not go upwards.

For example, the paths of `tree(1, [tree(2), tree(3, [tree(4), tree(5)])])` are

```
[1]
[1, 2]
[1, 3]
[1, 3, 4]
[1, 3, 5]
```

Implement `bigpath`, which takes a tree `t` and an integer `n`. It returns the number of paths in `t` whose sum is at least `n`. Assume that all node values of `t` are integers.

```
def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5)])])
    >>> bigpath(t, 3)
    4
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """
    "*** YOUR CODE HERE ***"
```

# Problem 3.5: Bigger Path (100 pts)

Now, implement `bigger_path`, which removes the restriction that paths must begin at the root -- they can begin at any node. You can use `bigpath` to help your implementation.

```
def bigger_path(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5)])])
    >>> bigger_path(t, 3)
    9
    >>> bigger_path(t, 6)
    4
    >>> bigger_path(t, 9)
    1
    """
    "*** YOUR CODE HERE ***"
```

# Just for fun Problems

This section is out of scope for our course, so the problems below is optional. That is, the problems in this section **don't** count for your final score and **don't** have any deadline. Do it at any time if you want an extra challenge.

At this time, we **don't** provide *Online Judgement*. The local **doctest** is enough for you to check your answer. You can find the starter code in `shakespeare.py`.

# Problem 4: Fold Tree (0pts)

It is often the case that we want to fold a tree into a value. Consider the following implementations of `count_leaves` and `label_sum`.

```
def count_leaves(t):
    """Count the leaves of a tree."""
    if is_leaf(t):
        return 1
    return sum([count_leaves(b) for b in branches(t)])

def label_sum(t):
    """Sum up the labels of all nodes in a tree."""
    if is_leaf(t):
        return label(t)
    return label(t) + sum([label_sum(b) for b in branches(t)])
```

The implementations look quite similar! Now, it is time to define a function `fold_tree` to abstract such a process.

```
def fold_tree(t, base_func, merge_func):
    """Fold tree into a value according to base_func and merge_func"""
    "*** YOUR CODE HERE ***"
```

You can feel free to define the meaning and usage of `base_func` and `merge_func`. Then use `fold_tree` to implement `count_leaves`, `label_sum` and `preorder`.

```
def count_leaves(t):
    """Count the leaves of a tree.

    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5)])])
    >>> count_leaves(t)
    3
    """
    return fold_tree(t, 'YOUR EXPRESSION HERE', 'YOUR EXPRESSION HERE')

def label_sum(t):
    """Sum up the labels of all nodes in a tree.

    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5)])])
    >>> label_sum(t)
    15
    """
    return fold_tree(t, 'YOUR EXPRESSION HERE', 'YOUR EXPRESSION HERE')

def preorder(t):
    """Return a list of the entries in this tree in the order that they
    would be visited by a preorder traversal.

    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5)])])
    >>> preorder(t)
    [1, 2, 3, 4, 5]
    """
    return fold_tree(t, 'YOUR EXPRESSION HERE', 'YOUR EXPRESSION HERE')
```

# Problem 5: Shakespeare and Dictionaries (0pts)

We will use dictionaries to approximate the entire works of Shakespeare! We're going to use a bigram language model. Here's the idea: We start with some word -- we'll use "The" as an example. Then we look through all of the texts of Shakespeare and for every instance of "The" we record the word that follows "The" and add it to a list, known as the successors of "The". Now suppose we've done this for every word Shakespeare has used, ever.

Let's go back to "The". Now, we randomly choose a word from this list, say "cat". Then we look up the successors of "cat" and randomly choose a word from that list, and we continue this process. This eventually will terminate in a period (".") and we will have generated a Shakespearean sentence!

The object that we'll be looking things up in is called a "successor table", although really it's just a dictionary. The keys in this dictionary are words, and the values are lists of successors to those words.

# Problem 4.1: Successor Tables

Here's an incomplete definition of the `build_successors_table` function. The input is a list of words (corresponding to a Shakespearean text), and the output is a successors table. (By default, the first word is a successor to "."). See the example below.

```
def build_successors_table(tokens):
    """Return a dictionary: keys are words; values are lists of successors.

    >>> text = ['We', 'came', 'to', 'investigate', ',', 'catch', 'bad', 'guys',
'and', 'to', 'eat', 'pie', '.']
    >>> table = build_successors_table(text)
    >>> sorted(table)
    [',', '.', 'We', 'and', 'bad', 'came', 'catch', 'eat', 'guys',
'investigate', 'pie', 'to']
    >>> table['to']
    ['investigate', 'eat']
    >>> table['pie']
    ['.']
    >>> table['.']
    ['We']
    """
    table = {}
    prev = '.'
    for word in tokens:
        if prev not in table:
            "*** YOUR CODE HERE ***"
        "*** YOUR CODE HERE ***"
        prev = word
    return table
```

# Problem 4.2: Construct the Sentence

Let's generate some sentences! Suppose we're given a starting word. We can look up this word in our table to find its list of successors, and then randomly select a word from this list to be the next word in the sentence. Then we just repeat until we reach some ending punctuation.

This might not be a bad time to play around with adding strings together as well. Let's fill in the `construct_sent` function!

```
def construct_sent(word, table):
    """Prints a random sentence starting with word, sampling from
    table.

    >>> table = {'Wow': ['!'], 'Sentences': ['are'], 'are': ['cool'], 'cool':
['.']}
    >>> construct_sent('Wow', table)
    'Wow!'
    >>> construct_sent('Sentences', table)
    'Sentences are cool.'
    """
    import random
    result = ''
    while word not in ['.', '!', '?']:
        "*** YOUR CODE HERE ***"
    return result.strip() + word
```

**Hint**: to randomly select from a list, import the Python random library with `import random` and use the expression `random.choice(my_list)`

# Problem 4.3: Become Shakespeare

Great! Now let's try to run our functions with some actual data. The following snippet included in the skeleton code will return a list containing the words in all of the works of Shakespeare.

```
def shakespeare_tokens(url='http://composingprograms.com/shakespeare.txt'):
    """Return the words of Shakespeare's plays as a list."""
    import os
    from urllib.request import urlopen
    shakespeare = urlopen(url)
    return shakespeare.read().decode(encoding='ascii').split()
```

Uncomment the following two lines to run the above function and build the successors table from those tokens.

```
# Uncomment the following two lines
# tokens = shakespeare_tokens()
# table = build_successors_table(tokens)
```

Next, let's define a utility function that constructs sentences from this successors table:

```
>>> def sent():
...     return construct_sent('The', table)
>>> sent()
" The plebeians have done us must be news-cramm'd."

>>> sent()
" The ravish'd thee , with the mercy of beauty!"

>>> sent()
" The bird of Tunis , or two white and plucker down with better ; that's God's
sake."
```

Notice that all the sentences start with the word "The". With a few modifications, we can make our sentences start with a random word. The following `random_sent` function (defined in your starter file) will do the trick:

```
def random_sent():
    import random
    return construct_sent(random.choice(table['.']), table)
```

Go ahead and load your file into Python (be sure to use the -i flag). You can now call the `random_sent` function to generate random Shakespearean sentences!

```
>>> random_sent()
' Long live by thy name , then , Dost thou more angel , good Master Deep-vow ,
And tak'st more ado but following her , my sight Of speaking false!'

>>> random_sent()
' Yes , why blame him , as is as I shall find a case , That plays at the public
weal or the ghost.'
```