Lab 4: Recursion, Tree Recursion, Python Lists **lab04.zip (lab04.zip)**

Due by 11:59pm on Wednesday, September 21.

Starter Files

Download lab04.zip (lab04.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Recursion

Recursion

A recursive function is a function that calls itself in its body, either directly or indirectly.

Let's look at the canonical example, `factorial`.

> Factorial, denoted with the `!` operator, is defined as:
>
> ```
> n! = n * (n-1) * ... * 1
> ```
>
> For example, `5! = 5 * 4 * 3 * 2 * 1 = 120`

The recursive implementation for factorial is as follows:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

We know from its definition that 0! is 1. Since `n == 0` is the smallest number we can compute the factorial of, we use it as our base case. The recursive step also follows from the definition of factorial, i.e., `n! = n * (n-1)!`.

Recursive functions have three important components:

1. **Base case.** You can think of the base case as the case of the simplest function input, or as the stopping condition for the recursion.
   In our example, `factorial(0)` is our base case for the `factorial` function.
2. **Recursive call on a smaller problem.** You can think of this step as calling the function on a smaller problem that our current problem depends on. We assume that a recursive call on this smaller problem will give us the expected result; we call this idea the "recursive leap of faith".
   In our example, `factorial(n)` depends on the smaller problem of `factorial(n-1)`.
3. **Solve the larger problem.** In step 2, we found the result of a smaller problem. We want to now use that result to figure out what the result of our current problem should be, which is what we want to return from our current function call.
   In our example, we can compute `factorial(n)` by multiplying the result of our smaller problem `factorial(n-1)` (which represents `(n-1)!`) by `n` (the reasoning being that `n! = n * (n-1)!`).

The next few questions in lab will have you writing recursive functions. Here are some general tips:

- Paradoxically, to write a recursive function, you must assume that the function is fully functional before you finish writing it; this is called the *recursive leap of faith*.
- Consider how you can solve the current problem using the solution to a simpler version of the problem. The amount of work done in a recursive function can be deceptively little: remember to take the leap of faith and *trust the recursion* to solve the slightly smaller problem without worrying about how.
- Think about what the answer would be in the simplest possible case(s). These will be your base cases - the stopping points for your recursive calls. Make sure to consider the possibility that you're missing base cases (this is a common way recursive solutions fail).
- It may help to write an iterative version first.

Tree Recursion

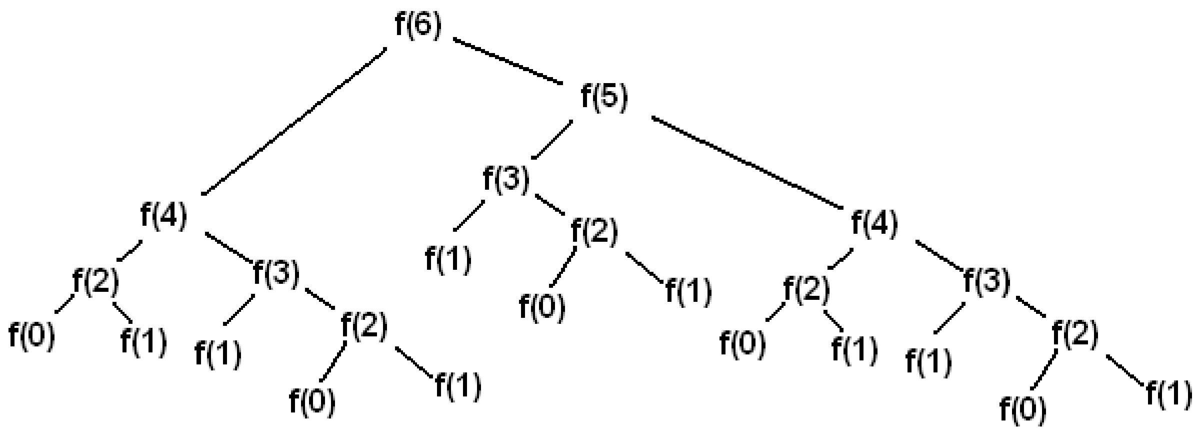
Tree Recursion

A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.

For example, let's say we want to recursively calculate the `n` th Virahanka-Fibonacci number (https://en.wikipedia.org/wiki/Fibonacci_number), defined as:

```
def virfib(n):
    if n == 0 or n == 1:
        return n
    return virfib(n - 1) + virfib(n - 2)
```

Calling `virfib(6)` results in the following call structure that looks like an upside-down tree (where `f` is `virfib`):



Each `f(i)` node represents a recursive call to `virfib`. Each recursive call `f(i)` makes another two recursive calls, which are to `f(i-1)` and `f(i-2)`. Whenever we reach a `f(0)` or `f(1)` node, we can directly return `0` or `1` rather than making more recursive calls, since these are our base cases.

In other words, base cases have the information needed to return an answer directly, without depending upon results from other recursive calls. Once we've reached a base case, we can then begin returning back from the recursive calls that led us to the base case in the first place.

Generally, tree recursion can be effective for problems where there are multiple possibilities or choices at a current state. In these types of problems, you make a recursive call for each choice or for a group of choices.

List Comprehensions

List Comprehensions

List comprehensions are a compact and powerful way of creating new lists out of sequences. The general syntax for a list comprehension is the following:

```
[<expression> for <element> in <sequence> if <conditional>]
```

where the `if <conditional>` section is optional.

The syntax is designed to read like English: "Compute the expression for each element in the sequence (if the conditional is true for that element)."

```
>>> [i**2 for i in [1, 2, 3, 4] if i % 2 == 0]
[4, 16]
```

This list comprehension will:

- Compute the expression `i**2`
- For each element `i` in the sequence `[1, 2, 3, 4]`
- Where `i % 2 == 0` (`i` is an even number),

and then put the resulting values of the expressions into a new list.

In other words, this list comprehension will create a new list that contains the square of every even element of the original list `[1, 2, 3, 4]`.

We can also rewrite a list comprehension as an equivalent `for` statement, such as for the example above:

```
>>> lst = []
>>> for i in [1, 2, 3, 4]:
...     if i % 2 == 0:
...         lst = lst + [i**2]
>>> lst
[4, 16]
```

Required Questions

Getting Started Videos

Recursion/Tree Recursion

Q1: WWPD: Squared Virahanka Fibonacci

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q squared-virfib-wwpd -u
```

Hint: If you are stuck, make sure to try drawing out the recursive call tree! This is a challenging problem -- doing so will really help with understanding how tree recursion works. We strongly encourage trying to draw things out before asking for help from your TA/AIs.

Background: In the Squared Virahanka Fibonacci sequence, each number in the sequence is the square of the sum of the previous two numbers in the sequence. The first 0th and 1st number in the sequence are 0 and 1, respectively. The recursive `virfib_sq` function takes in an argument `n` and returns the `nth` number in the Square Virahanka Fibonacci sequence.

```
>>> def virfib_sq(n):
>>>     print(n)
>>>     if n <= 1:
>>>         return n
>>>     return (virfib_sq(n - 1) + virfib_sq(n - 2)) ** 2
>>> r0 = virfib_sq(0)
______

>>> r1 = virfib_sq(1)
______

>>> r2 = virfib_sq(2)
______

>>> r3 = virfib_sq(3)
______

>>> r3
______

>>> (r1 + r2) ** 2
______

>>> r4 = virfib_sq(4)
______

>>> r4
______
```

Q2: Summation

Write a recursive implementation of `summation`, which takes a positive integer `n` and a function `term`. It applies `term` to every number from `1` to `n` including `n` and returns the sum.

Important: Use recursion; the tests will fail if you use any loops (for, while).

```
def summation(n, term):
    """Return the sum of numbers 1 through n (including n) wíth term applied to each nu
    Implement using recursion!

    >>> summation(5, lambda x: x * x * x) # 1^3 + 2^3 + 3^3 + 4^3 + 5^3
    225
    >>> summation(9, lambda x: x + 1) # 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
    54
    >>> summation(5, lambda x: 2**x) # 2^1 + 2^2 + 2^3 + 2^4 + 2^5
    62
    >>> # Do not use while/for loops!
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'summation',
    ...       ['While', 'For'])
    True
    """
    assert n >= 1
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q summation
```

Q3: Pascal's Triangle

Pascal's triangle gives the coefficients of a binomial expansion; if you expand the expression `(a + b) ** n`, all coefficients will be found on the `n` th row of the triangle, and the coefficient of the `i` th term will be at the `i` th column.

Here's a part of the Pascal's trangle:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Every number in Pascal's triangle is defined as the sum of the item above it and the item above and to the left of it. Rows and columns are zero-indexed; that is, the first row is row 0 instead of 1 and the first column is column 0 instead of column 1. For example, the item at row 2, column 1 in Pascal's triangle is 2.

Now, define the procedure `pascal(row, column)` which takes a row and a column, and finds the value of the item at that position in Pascal's triangle. Note that Pascal's triangle is only defined at certain areas; use `0` if the item does not exist. For the purposes of this question, you may also assume that `row >= 0` and `column >= 0`.

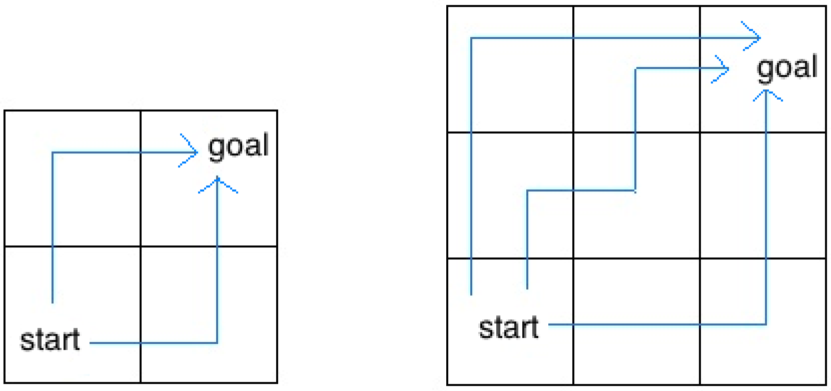
```
def pascal(row, column):
    """Returns the value of the item in Pascal's Triangle
    whose position is specified by row and column.
    >>> pascal(0, 0)    # The top left (the point of the triangle)
    1
    >>> pascal(0, 5)    # Empty entry; outside of Pascal's Triangle
    0
    >>> pascal(3, 2)    # Row 3 (1 3 3 1), Column 2
    3
    >>> pascal(4, 2)     # Row 4 (1 4 6 4 1), Column 2
    6
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q pascal
```

Q4: Insect Combinatorics

Consider an insect in an *M* by *N* grid. The insect starts at the bottom left corner, *(1, 1)*, and wants to end up at the top right corner, *(M, N)*. The insect is only capable of moving right or up. Write a function `paths` that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. (There is a closed-form solution (https://en.wikipedia.org/wiki/Closed-form_expression) to this problem, but try to answer it procedurally using recursion.)



For example, the 2 by 2 grid has a total of two ways for the insect to move from the start to the goal. For the 3 by 3 grid, the insect has 6 diferent paths (only 3 are shown above).

> **Hint:** What happens if we hit the top or rightmost edge?

```
def paths(m, n):
    """Return the number of paths from one corner of an
    M by N grid to the opposite corner.

    >>> paths(2, 2)
    2
    >>> paths(5, 7)
    210
    >>> paths(117, 1)
    1
    >>> paths(1, 157)
    1
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q paths
```

List Comprehensions

Q5: Couple

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.

> *Hint*: You may find the built in range (https://www.w3schools.com/python/ref_func_range.asp) function helpful.

```
def couple(s, t):
    """Return a list of two-element lists in which the i-th element is [s[i], t[i]].

    >>> a = [1, 2, 3]
    >>> b = [4, 5, 6]
    >>> couple(a, b)
    [[1, 4], [2, 5], [3, 6]]
    >>> c = ['c', 6]
    >>> d = ['s', '1']
    >>> couple(c, d)
    [['c', 's'], [6, '1']]
    """
    assert len(s) == len(t)
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q couple
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Recursion

Q6: Double Eights

Write a recursive function that takes in a number `n` and determines if the digits contain two adjacent `8`s. You can assume that `n` is at least a two-digit number. You may have already done this problem iteratively as an Extra Practice problem in Lab 1.

> **Hint:** Remember what tools you can use in order to isolate digits of a number. If you have trouble figuring out how to implement the recursion, try first finding an iterative solution, and think about how you might be able to turn any loops in recursion.

```
def double_eights(n):
    """ Returns whether or not n has two digits in row that
    are the number 8. Assume n has at least two digits in it.

    >>> double_eights(1288)
    True
    >>> double_eights(880)
    True
    >>> double_eights(538835)
    True
    >>> double_eights(284682)
    False
    >>> double_eights(588138)
    True
    >>> double_eights(78)
    False
    >>> from construct_check import check
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'double_eights', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q double_eights
```

List Comprehensions

Q7: Coordinates

Implement a function `coords` that takes a function `fn`, a sequence `seq`, and a `lower` and `upper` bound on the output of the function. `coords` then returns a list of coordinate pairs (lists) such that:

- Each (x, y) pair is represented as `[x, fn(x)]`
- The x-coordinates are elements in the sequence
- The result contains only pairs whose y-coordinate is within the upper and lower bounds (inclusive)

See the doctest for examples.

> *Note*: your answer can only be *one line long*. You should make use of list comprehensions!

```
def coords(fn, seq, lower, upper):
    """
    >>> seq = [-4, -2, 0, 1, 3]
    >>> fn = lambda x: x**2
    >>> coords(fn, seq, 1, 9)
    [[-2, 4], [1, 1], [3, 9]]
    """
    "*** YOUR CODE HERE ***"
    return ______
```

Use Ok to test your code:

```
python3 ok -q coords
```

> Reflect: What are the drawbacks to the one-line answer, in terms of using computer resources?

Q8: Riffle Shuffle

A common way of shuffling cards is known as the riffle shuffle (https://www.youtube.com/watch?v=JmbVNyIiD54). The shuffle produces a new configuration of cards in which the top card is followed by the middle card, then by the second card, then the card after the middle, and so forth.

Write a list comprehension that riffle shuffles a sequence of items. You can assume the sequence contains an even number of items.

Hint: There are two ways you can write this as a single list comprehension: 1) You may find the expression `k%2`, which evaluates to 0 on even numbers and 1 on odd numbers, to be alternatively access the beginning and middle of the deck. 2) You can utilize an if expression in your comprehension for the odd and even numbers respectively.

```
def riffle(deck):
    """Produces a single, perfect riffle shuffle of DECK, consisting of
    DECK[0], DECK[M], DECK[1], DECK[M+1], ... where M is position of the
    second half of the deck.  Assume that len(DECK) is even.
    >>> riffle([3, 4, 5, 6])
    [3, 5, 4, 6]
    >>> riffle(range(20))
    [0, 10, 1, 11, 2, 12, 3, 13, 4, 14, 5, 15, 6, 16, 7, 17, 8, 18, 9, 19]
    """
    "*** YOUR CODE HERE ***"
    return _______
```

Use Ok to test your code:

```
python3 ok -q riffle
```