Lab 3: Recursion, Tree Recursion

lab03.zip (lab03.zip)

Due by 11:59pm on Thursday, July 1.

Starter Files

Download lab03.zip (lab03.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. Recursive functions have three important components:

1. Base case(s), the simplest possible form of the problem you're trying to solve.
2. Recursive case(s), where the function calls itself with a *simpler argument* as part of the computation.
3. Using the recursive calls to solve the full problem.

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)!`.

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

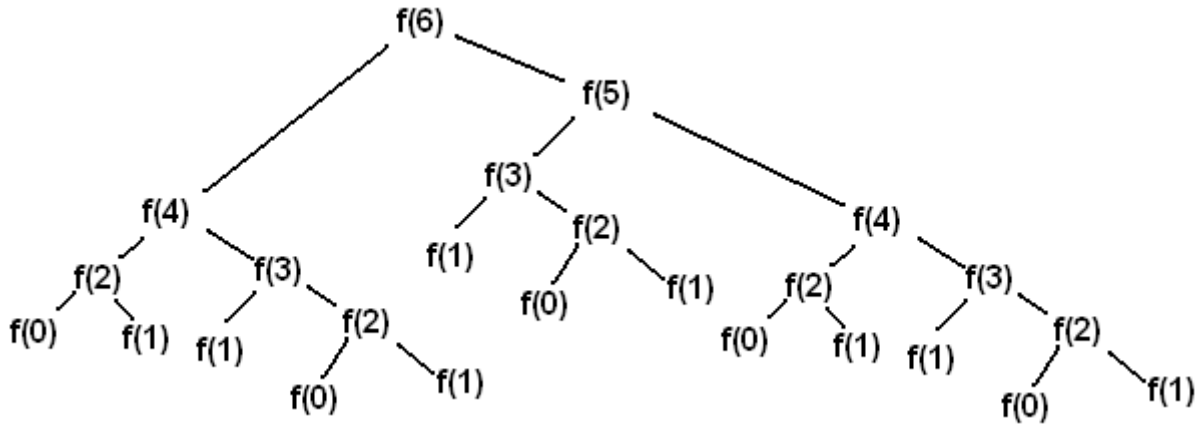
Tree Recursion

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

A classic example of a tree recursion function is finding the nth Fibonacci number (https://www.wikiwand.com/en/Fibonacci_number):

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

Calling `fib(6)` results in the following call structure (where `f` is `fib`):



Each `f(i)` node represents a recursive call to `fib`. Each recursive call makes another two recursive calls. `f(0)` and `f(1)` do not make any recursive calls because they are the base cases of the function. Because of these base cases, we are able to terminate the recursion and beginning accumulating the values.

Generally, tree recursion is effective when you want to explore multiple possibilities or choices at a single step. In these types of problems, you make a recursive call for each choice or for a group of choices. Here are some examples:

- Given a list of paid tasks and a limited amount of time, which tasks should you choose to maximize your pay? This is actually a variation of the Knapsack (https://en.wikipedia.org/wiki/Knapsack_problem) problem, which focuses on finding some optimal combination of different items.
- Suppose you are lost in a maze and see several different paths. How do you find your way out? This is an example of path finding, and is tree recursive because at every step, you could have multiple directions to choose from that could lead out of the maze.
- Your dryer costs $2 per cycle and accepts all types of coins. How many different combinations of coins can you create to run the dryer? This is similar to the partitions (http://composingprograms.com/pages/17-recursive-functions.html#example-partitions) problem from the textbook.

Required Questions

What Would Python Display?

Q1: WWPD: Recursion

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

```
python3 ok -q recursion-wwpd -u
```

For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, `Infinite` if there is an infinite loop or infinite recursion, and `Nothing` if nothing is displayed.

```
>>> def f(a, b):
...     if a > b:
...         return f(a - 3, 2 * b)
...     elif a < b:
...         return f(b // 2, a)
...     else:
...         return b
>>> f(2, 2)
______

>>> f(7, 4)
______

>>> f(2, 28)
______

>>> f(-1, -3)
______
```

Q2: WWPD: Journey to the Center of the Earth

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

```
python3 ok -q drill-wwpd -u
```

For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> def crust():
...     print("70km")
...     def mantle():
...         print("2900km")
...         def core():
...             print("5300km")
...             return mantle()
...         return core
...     return mantle
>>> drill = crust
>>> drill = drill()
______

>>> drill = drill()
______

>>> drill = drill()
_______

>>> drill()
_____
```

Coding Practice

Getting Started Video

Q3: 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.

```
def summation(n, term):
    """Return the sum of numbers 1 through n (including n) with term applied to each
    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
```

Q4: Pascal's Triangle

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. Use `0` if the item does not exist.

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. 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.

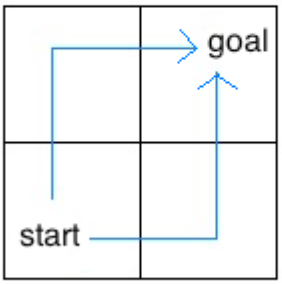
```
def pascal(row, column):
    """Returns the value of the item in Pascal's Triangle
    whose position is specified by row and column.
    >>> pascal(0, 0)
    1
    >>> pascal(0, 5)    # Empty entry; outside of Pascal's Triangle
    0
    >>> pascal(3, 2)    # Row 3 (1 3 3 1), Column 2
    3
   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

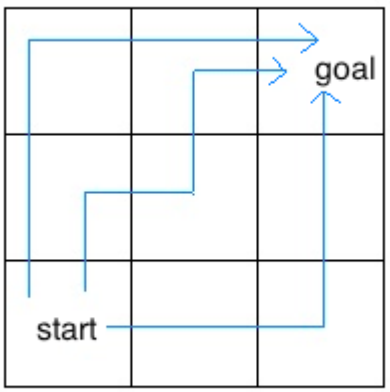
Use Ok to test your code:

```
python3 ok -q pascal
```

Q5: Insect Combinatorics

Consider an insect in an *M* by *N* grid. The insect starts at the bottom left corner, *(0, 0)*, and wants to end up at the top right corner, *(M-1, N-1)*. 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
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```