CS 61A
Fall 2023

Tree Recursion

Discussion 4: September 20, 2023

Tree Recursion

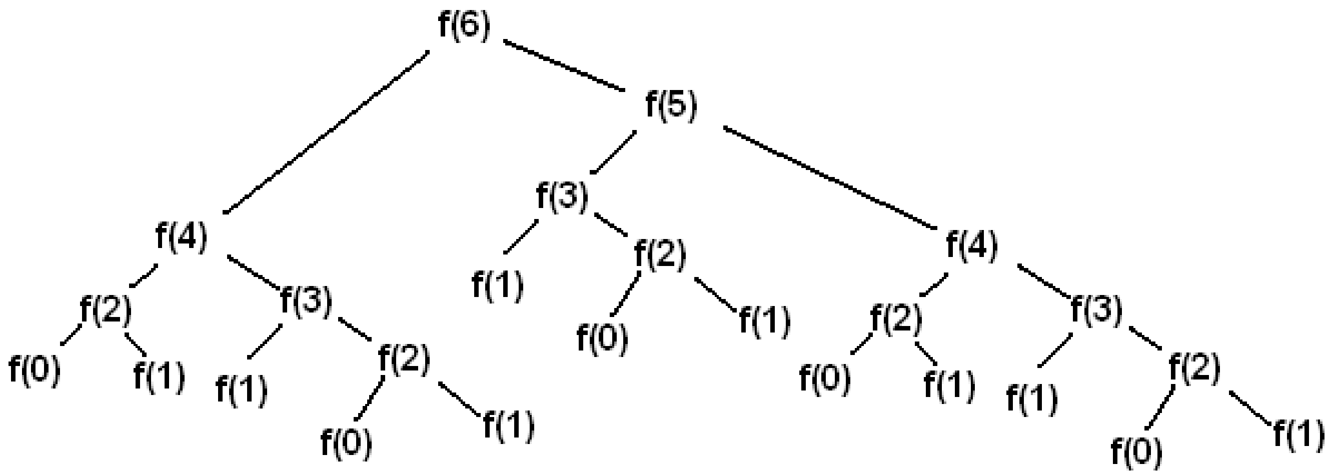
A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.

For example, this is the Virahanka-Fibonacci sequence: `0, 1, 1, 2, 3, 5, 8, 13, ....`

Each term is the sum of the previous two terms. This tree-recursive function calculates the `n`th Virahanka-Fibonacci number.

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

Calling `virfib(6)` results in a call structure that resembles an upside-down tree (where `f` is `virfib`):



Virahanka-Fibonacci tree.

Each recursive call `f(i)` makes a call to `f(i-1)` and a call to `f(i-2)`. Whenever we reach an `f(0)` or `f(1)` call, we can directly return `0` or `1` without making more recursive calls. These calls are our base cases.

A base case returns an answer without depending on the results of other calls. Once we reach a base case, we can go back and answer the recursive calls that led to the base case.

As we will see, tree recursion is often effective for problems with branching choices. In these problems, you make a recursive call for each branching choice.

Q1: Count Stair Ways

Imagine that you want to go up a flight of stairs that has `n` steps, where `n` is a positive integer. You can take either one or two steps each time you move. In how many ways can you go up the entire flight of stairs?

You'll write a function `count_stair_ways` to answer this question. Before you write any code, consider:

- How many ways are there to go up a flight of stairs with `n = 1` step? What about `n = 2` steps? Try writing or drawing out some other examples and see if you notice any patterns.
- What is the base case for this question? What is the smallest input?
- What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Now, fill in the code for `count_stair_ways`:

```
def count_stair_ways(n):
    """Returns the number of ways to climb up a flight of
    n stairs, moving either one step or two steps at a time.
    >>> count_stair_ways(1)
    1
    >>> count_stair_ways(2)
    2
    >>> count_stair_ways(4)
    5
    """
    "*** YOUR CODE HERE ***"









# You can use more space on the back if you want
```

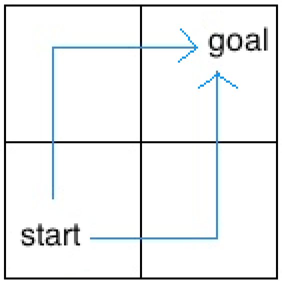
Q2: Count K

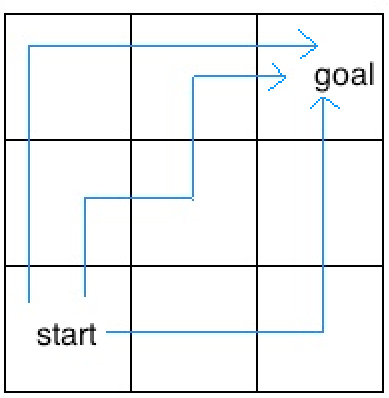
Consider a special version of the `count_stair_ways` problem where we can take up to `k` steps at a time. Write a function `count_k` that calculates the number of ways to go up an `n`-step staircase. Assume `n` and `k` are positive integers.

```
def count_k(n, k):
    """Counts the number of paths up a flight of n stairs
    when taking up to k steps at a time.
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """
    "*** YOUR CODE HERE ***"

























# You can use more space on the back if you want
```

Q3: Insect Combinatorics

An insect is inside 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 can only move up or to the right. Write a function `paths` that takes the length and width of a grid and returns the number of paths the insect can take from the start to the end. (There is a closed-form solution to this problem, but try to answer it with recursion.)





Insect grids.

In the `2` by `2` grid, the insect has two paths from the start to the end. In the `3` by `3` grid, the insect has six paths (only three are shown above).

> **Hint:** What happens if the insect hits the upper or rightmost edge of the grid?

```
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 ***"
















# You can use more space on the back if you want
```

Q4: Max Product

Write a function that takes in a list and returns the maximum product that can be formed using non-consecutive elements of the list. All numbers in the input list are greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be formed using
    non-consecutive elements of s.
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """









# You can use more space on the back if you want
```

Q5: Flatten

Write a function `flatten` that takes a list and returns a "flattened" version of it. The input list may be a "deep list" (a list that contains other lists).

In the following example, `[1, [[2], 3], 4, [5, 6]]` is a deep list because `[[2], 3]` and `[5, 6]` are lists. Note that `[[2], 3]` is itself a deep list.

```
>>> lst = [1, [[2], 3], 4, [5, 6]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6]
```

Hint: you can check if something in Python is a list with the built-in `type` function. For example:

```
>>> type(3) == list
False
>>> type([1, 2, 3]) == list
True
```

```
def flatten(s):
    """Returns a flattened version of list s.

    >>> flatten([1, 2, 3])
    [1, 2, 3]
    >>> deep = [1, [[2], 3], 4, [5, 6]]
    >>> flatten(deep)
    [1, 2, 3, 4, 5, 6]
    >>> deep                                # input list is unchanged
    [1, [[2], 3], 4, [5, 6]]
    >>> very_deep = [['m', ['i', ['n', ['m', 'e', ['w', 't', ['a'], 't', 'i', 'o'], 'n
    ']], 's']]]
    >>> flatten(very_deep)
    ['m', 'i', 'n', 'm', 'e', 'w', 't', 'a', 't', 'i', 'o', 'n', 's']
    """
    "*** YOUR CODE HERE ***"















# You can use more space on the back if you want
```