Tree Recursion

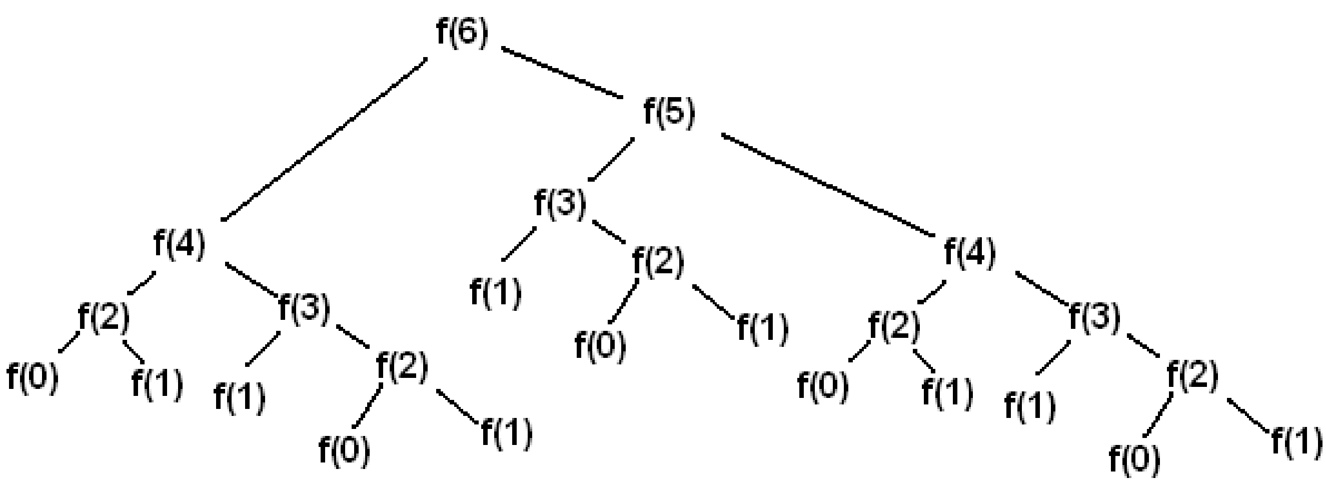
A tree recursive function is a recursive function that makes more than one call to itself, resulting in a tree-like series of calls.

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.

```
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.

Solution: When there is only one step, there is only one way to go up. When there are two steps, we can go up in two ways: take a single 2-step, or take two 1-steps.

- What is the base case for this question? What is the smallest input?

Solution: We actually have two base cases! Our first base case is when there is one step left. `n = 1` is the smallest input because 1 is the smallest positive integer.

Our second base case is when there are two steps left. The primary solution (found below) cannot solve `count_stair_ways(2)` recursively because `count_stair_ways(0)` is undefined.

(`virfib` has two base cases for a similar reason: `virfib(1)` cannot be solved recursively because `virfib(-1)` is undefined.)

Alternate solution: Our first base case is when there are no steps left. This means we reached the top of the stairs with our last action.

Our second base case is when we have overstepped. This means our last action was invalid; in other words, we took two steps when only one step remained.

- What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

Solution: `count_stair_ways(n - 1)` is the number of ways to go up `n - 1` stairs. Equivalently, `count_stair_ways (n - 1)` is the number of ways to go up `n` stairs if our first action is taking one step.

`count_stair_ways(n - 2)` is the number of ways to go up `n - 2` stairs. Equivalently, `count_stair_ways(n - 2)` is the number of ways to go up `n` stairs if our first action is taking two steps.

Now, fill in the code for `count_stair_ways`:

```python
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
    """
    if n == 1:
        return 1
    elif n == 2:
        return 2
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

Here's an alternate solution that corresponds to the alternate base cases:

```python
def count_stair_ways_alt(n):
    """Returns the number of ways to climb up a flight of
    n stairs, moving either 1 step or 2 steps at a time.
    >>> count_stair_ways_alt(4)
    5
    """
    if n == 0:
        return 1
    elif n < 0:
        return 0
    return count_stair_ways_alt(n-1) + count_stair_ways_alt(n-2)
```

You can use Recursion Visualizer to step through the call structure of `count_stair_ways(4)` for the primary solution.

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.

```python
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
    """
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        for step in range(1, k + 1):
            total += count_k(n - step, k)
        return total

# ALTERNATE SOLUTION
def count_k(n, k):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        i = 1
        while i <= k:
            total += count_k(n - i, k)
            i += 1
        return total
```

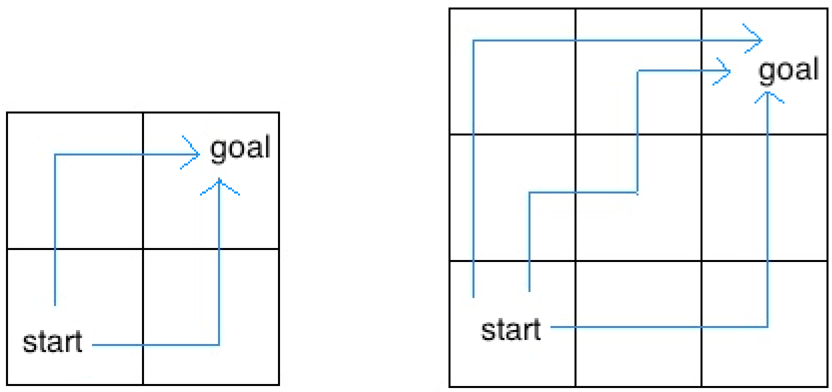
We use a loop in our solution for `count_k`. The first iteration of the loop counts the ways to go up `n` steps if we start by taking one step, the second iteration counts the ways to go up `n` steps if we start by taking two steps, and so on. The very last iteration counts the ways to go up `n` steps if we start by taking `k` steps, which is the most we can take at once.

We use the `total` variable to track the sum of the results of our recursive `count_k` calls. When the loop ends, `total` will store the number of ways to go up `n` stairs and will include every possible starting move.

You can use recursion visualizer to step through the call structure of `count_k(3, 3)`.

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 height 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
    """
    if m == 1 or n == 1:
        return 1
    return paths(m - 1, n) + paths(m, n - 1)
    # Base case: Look at the two visual examples given. Since the insect
    # can only move to the right or up, once it hits either the rightmost edge
    # or the upper edge, it has a single remaining path -- the insect has
    # no choice but to go straight up or straight right (respectively) at that point.
    # There is no way for it to backtrack by going left or down.

    # Alternative solution:
    if m == 1 and n == 1:
        return 1
    if m < 1 or n < 1:
        return 0
    return paths(m - 1, n) + paths(m, n - 1)
    # This solution is similar to the alternate solution for Count Stair Ways.
    # If we reach the exact destination, we have found a unique path (first base case),
    but if
    # we overshoot, we have not found a valid path (second base case).

    # Notice, however, that this solution is not as short and simple as the first
    solution
    # since it doesn't make use of the insect's restricted movements (only right or up)
    # to cut the program short. We have to reach the exact destination for the second
    solution,
    # while in the first we just have to reach the right or top boundary.
```

The recursive case is that there are paths from the square to the right through an (m, n-1) grid and paths from the square above through an (m-1, n) grid.

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
    """
    if s == []:
        return 1
    if len(s) == 1:
        return s[0]
    else:
        return max(s[0] * max_product(s[2:]), max_product(s[1:]))
        # OR
        return max(s[0] * max_product(s[2:]), s[1] * max_product(s[3:]))
```

This solution begins with the idea that we either include `s[0]` in the product or not:

- If we include `s[0]`, we cannot include `s[1]`.
- If we don't include `s[0]`, we can include `s[1]`.

The recursive case is that we choose the larger of: - multiplying `s[0]` by the `max_product` of `s[2:]` (skipping `s[1]`) OR - just the `max_product` of `s[1:]` (skipping `s[0]`)

Here are some key ideas in translating this into code: - The built-in `max` function can find the larger of two numbers, which in this case come from two recursive calls. - In every case, `max_product` is called on a list of numbers and its return value is treated as a number.

An expression for this recursive case is:

`max(s[0] * max_product(s[2:]), max_product(s[1:]))`

Since this expression never refers to `s[1]`, and `s[2:]` evaluates to the empty list even for a one-element list `s`, the second base case (`len(s) == 1`) can be omitted if this recursive case is used.

The recursive solution above explores some options that we know in advance will not be the maximum, such as skipping both `s[0]` and `s[1]`. Alternatively, the recursive case could be that we choose the larger of: - multiplying `s[0]` by the `max_product` of `s[2:]` (skipping `s[1]`) OR - multiplying `s[1]` by the `max_product` of `s[3:]` (skipping `s[0]` and `s[2]`)

An expression for this recursive case is:

`max(s[0] * max_product(s[2:]), s[1] * max_product(s[3:]))`

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']
    """
    lst = []
    for elem in s:
        if type(elem) == list:
            lst += flatten(elem)
        else:
            lst += [elem]
    return lst

# Alternate solution
    if not s:
        return []
    elif type(s[0]) == list:
        return flatten(s[0]) + flatten(s[1:])
    else:
        return [s[0]] + flatten(s[1:])
```

The provided solution creates a new list `lst` in order to avoid mutating `s`. Our goal is to add each element in `s` to `lst` while preserving the flatness of `lst`.

If `lst` is a flat list and `elem` is not a list, then `lst + [elem]` will be a flat list:

```
>>> lst = [1, 2, 3]
>>> elem = 4
>>> lst + [elem]
[1, 2, 3, 4]                # flat!
```

But this won't work when `elem` is a list. For example:

```
>>> lst = [1, 2, 3, 4]
>>> elem = [5, 6]
>>> lst + [elem]
[1, 2, 3, 4, [5, 6]]        # deep! :(
```

If `lst` and `elem` are flat lists, then `lst + elem` will be flat:

```
>>> lst = [1, 2, 3, 4]
>>> elem = [5, 6]
>>> lst + elem
[1, 2, 3, 4, 5, 6]          # flat! :)
```

But what if `elem` is a deep list? Then `lst + elem` will be deep:

```
>>> lst = [1]
>>> elem = [[2], 3]
>>> lst + elem
>>> [1, [2], 3]             # deep! >:0
```

So we need to flatten `elem` before adding it to `lst`! This is where recursion comes into play.

```
>>> lst = [1]
>>> elem = [[2], 3]
>>> lst + flatten(elem)
>>> [1, 2, 3]               # flat!   ()
```

In conclusion, we perform `lst += [elem]` when `elem` is not a list and `lst += flatten(elem)` when `elem` is a list.

Note that `flatten([5, 6]) == [5, 6]`, so calling `flatten(elem)` is unnecessary but harmless when `elem` is a shallow list.