CS 61A
Spring 2026

Tree Recursion

Discussion 4: February 18, 2026

Definition: Tree recursive functions are functions that call themselves more than once.

Recursion takes practice. Please don't get discouraged if you're struggling to write recursive functions. Instead, every time you do solve one (even with help or in a group), make note of what you had to realize to make progress. Students improve through practice and reflection.

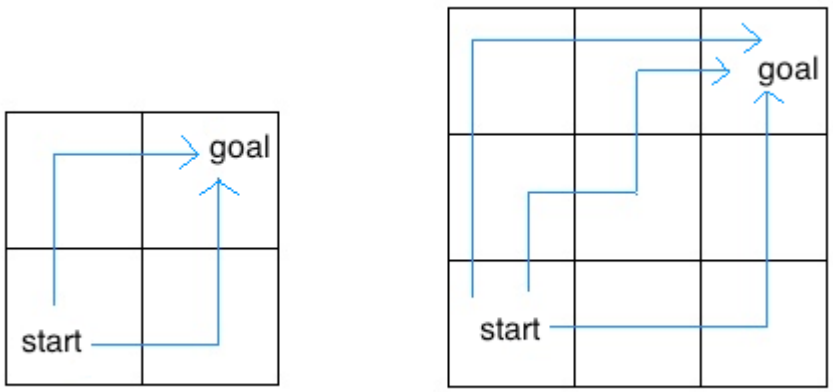
Tree Recursion

For the following questions, don't start trying to write code right away. Instead, start by describing the recursive case in words. Some examples:

- In `fib` from lecture, the recursive case is to add together the previous two Fibonacci numbers.
- In `double_eights` from lab, the recursive case is to check for double eights in the rest of the number.
- In `count_partitions` from lecture, the recursive case is to partition `n-m` using parts up to size `m` **and** to partition `n` using parts up to size `m-1`.

Q1: 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).

```
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 ***"
```

Presentation Time: Once your group has converged on a solution, it's time to practice your ability to describe why your recursive case is correct. Nominate someone and have them present to the group for practice. If you want feedback, ask.

Tree Recursion with Lists

Some of you already know list operations that we haven't covered yet, such as `append`. Don't use those today. All you need are list literals (e.g., `[1, 2, 3]`), item selection (e.g., `s[0]`), list addition (e.g., `[1] + [2, 3]`), `len` (e.g., `len(s)`), and slicing (e.g., `s[1:]`). Use those! There will be plenty of time for other list operations when we introduce them next week.

Important: The most important thing to remember about lists is that a non-empty list `s` can be split into its first element `s[0]` and the rest of the list `s[1:]`.

```
>>> s = [2, 3, 6, 4]
>>> s[0]
2
>>> s[1:]
[3, 6, 4]
```

Q2: Max Product

Implement `max_product`, which takes a list of numbers and returns the maximum product that can be formed by multiplying together non-consecutive elements of the list. Assume that all numbers in the input list are greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product of 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([])                 # The product of no numbers is 1
    1
    """
    "*** YOUR CODE HERE ***"
```

Q3: Sum Fun

Implement `sums(n, m)`, which takes a total `n` and maximum `m`. It returns a list of all lists: 1. that sum to `n`, 2. that contain only positive numbers up to `m`, and 3. in which no two adjacent numbers are the same.

Two lists with the same numbers in a different order should both be returned.

Here's a recursive approach that matches the template below: build up the `result` list by building all lists that sum to `n` and start with `k`, for each `k` from 1 to `m`. For example, the result of `sums(5, 3)` is made up of three lists:

- `[[1, 3, 1]]` starts with 1,
- `[[2, 1, 2], [2, 3]]` start with 2, and
- `[[3, 2]]` starts with 3.

```
def sums(n, m):
    """Return lists that sum to n containing positive numbers up to m that
    have no adjacent repeats.

    >>> sums(5, 1)
    []
    >>> sums(5, 2)
    [[2, 1, 2]]
    >>> sums(5, 3)
    [[1, 3, 1], [2, 1, 2], [2, 3], [3, 2]]
    >>> sums(5, 5)
    [[1, 3, 1], [1, 4], [2, 1, 2], [2, 3], [3, 2], [4, 1], [5]]
    >>> sums(6, 3)
    [[1, 2, 1, 2], [1, 2, 3], [1, 3, 2], [2, 1, 2, 1], [2, 1, 3], [2, 3, 1], [3, 1, 2],
    [3, 2, 1]]
    """
    if n < 0:
        return []
    if n == 0:
        sums_to_zero = []     # The only way to sum to zero using positives
        return [sums_to_zero] # Return a list of all the ways to sum to zero
    result = []
    for k in range(1, m + 1):
        result = result + [ ___ for rest in ___ if rest == [] or ___ ]
    return result
```

If you get stuck (which many groups do), ask for help!

Optional Question

Tree recursion problems often appear on exams. Here's one:

Q4: A Perfect Question

This question was Fall 2023 Midterm 2 Question 4(a). The original exam version had an extra blank (where `total < k * k` appears below), but also included some guidance via multiple choice options and hints.

Definition. A *perfect square* is `k*k` for some integer `k`.

Implement `fit`, which takes positive integers `total` and `n`. It returns `True` or `False` indicating whether there are `n` **positive** perfect squares that sum to `total`. The perfect squares need not be unique.

```
def fit(total, n):
    """Return whether there are n positive perfect squares that sums to total.

    >>> [fit(4, 1), fit(4, 2), fit(4, 3), fit(4, 4)]  # 1*(2*2) for n=1; 4*(1*1) for n=4
    [True, False, False, True]
    >>> [fit(12, n) for n in range(3, 8)]  # 3*(2*2), 3*(1*1)+3*3, 4*(1*1)+2*(2*2)
    [True, True, False, True, False]
    >>> [fit(32, 2), fit(32, 3), fit(32, 4), fit(32, 5)] # 2*(4*4), 3*(1*1)+2*2+5*5
    [True, False, False, True]
    """
    def f(total, n, k):
        if ______:
            return True
        elif ______:
            return False
        else:
            return ______
    return f(total, n, 1)
```