CS 61A
Fall 2023

Tree Recursion

Discussion 4: September 20, 2023

Getting Started

In this discussion, don't use a Python interpreter to run code until you are sure you are correct. Figure things out and check your work by *thinking* about what your code will do. Not sure? Talk to your group!

[New] 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]`), and slicing. Use those! There will be plenty of time for other list operations when we introduce them next week.

To get help from a TA, send a message to the `discuss-queue` channel with the @discuss tag and your discussion group number.

What everyone will need: * A way to view this worksheet (on your phone is fine) * A way to take notes (paper or a device)

What the group will need: * Someone to join Discord.

[New] If you have only 1 or 2 people in your group, you can join the other group in the room with you.

[New] 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.

Q1: Cowboy Hat

Together, find the shortest expression that can fill the blank below. When your whole group agrees, paste your answer into your group's channel's text chat.

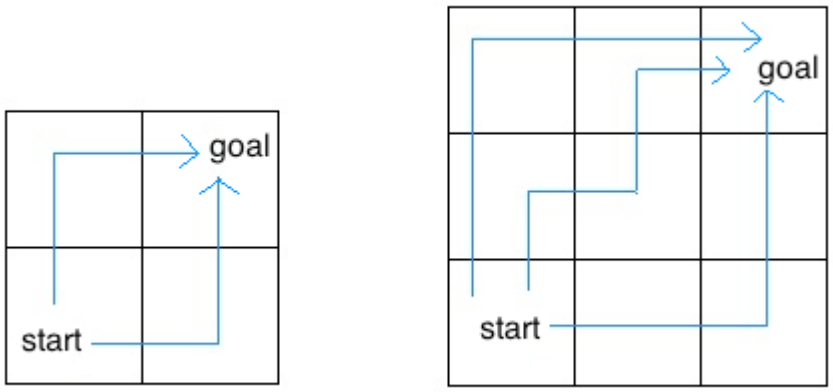
```
>>> o = [2, 0, 2, 3]
>>> [ _______ for D in range(1,4)]
[[2, 0, 2], [2, 0], [2]]
```

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`.

Q2: 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
    """
    "*** 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.

Hint: Use `[k] + s` for a number `k` and list `s` to build a list that starts with `k` and then has all the elements of `s`.

```
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 or aren't sure if you're on the right track, that's normal. Ask for help in the `discuss-queue` channel.

Once your group has converged on a solution, it's time to practice your ability to describe your own code. Pick a presenter, then send a message to the discuss-queue channel with the @discuss tag, your discussion group number, and the message "Ready to sum it up!" and a member of the course staff will join your voice channel to hear your description and give feedback.

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
    """
```

Complete this sentence together and type your answer into your group's channel's text chat. "The recursive case is to choose the larger of … and …"

Document the Occasion

Please all fill out the attendance form (one submission per person per week).

Extra Challenge

If you have time, try this more challenging problem. You're welcome to ask for help on it. This question will be reviewed in lecture because some groups might not finish it.

Q5: Sum More Fun

Implement `nested_sums(n)`, which takes a positve total `n`. It returns a list of all nested lists of `n` 1's that have at least one 1 in between each pair of brackets. For example, `[1, [1, 1], 1]` is an allowed nested list of 4 1's, but `[[1, 1, 1], 1]` is not allowed because it starts with two adjacent brackets without a 1 between them. Likewise, `[1, [1], [1], 1]` is not allowed.

```
def nested_sums(n):
    """Return all nested lists of n 1's with no adjacent brackets.

    >>> for s in nested_sums(5): print(s)
    [1, 1, 1, 1, 1]
    [1, 1, 1, [1], 1]
    [1, 1, [1], 1, 1]
    [1, 1, [1, 1], 1]
    [1, [1], 1, 1, 1]
    [1, [1], 1, [1], 1]
    [1, [1, 1], 1, 1]
    [1, [1, 1, 1], 1]
    [1, [1, [1], 1], 1]
    >>> len(nested_sums(7))
    51
    """
    assert n > 0
    if n == 1:
        return [[1]]
    "*** YOUR CODE HERE ***"
```