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]]
```

The shortest solution is `o[:-D]`, which looks a bit like this fellow: .

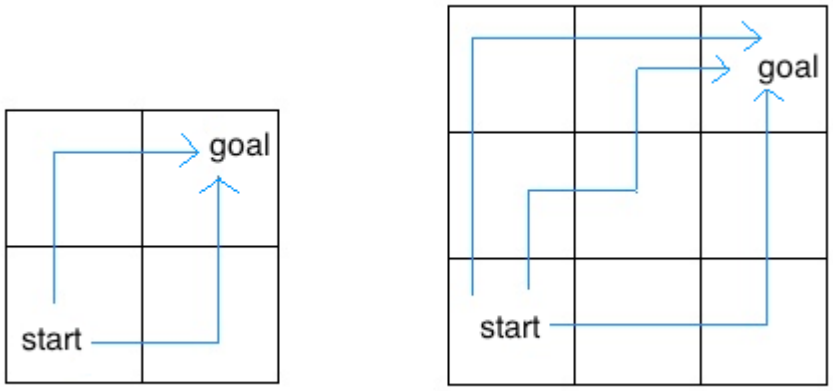
Tree Recursion

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
    """
    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.

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 + [[k] + rest for rest in sums(n-k, m) if rest == [] or rest[0]
    != k]
    return result
```

The recursive case is that each list that sums to `n` is an integer `k` (up to `m`) followed by the elements of a list that sums to `n-k` and does not start with `k`.

Here are some key ideas in translating this into code: - If `rest` is `[2, 3]`, then `[1] + rest` is `[1, 2, 3]`. - In the expression `[... for rest in sums(...) if ...]`, `rest` will be bound to each of the lists within the list returned by the recursive call. For example, if `sums(3, 2)` was called, then `rest` would be bound to `[1, 2]` (and then later `[2, 1]`).

In the solution, the expresson below creates a list of lists that start with `k` and are followed by the elements of `rest`, first checking that `rest` does not also start with `k` (which would construct a list starting with two k's).

```
[[k] + rest for rest in sums(n-k, m) if rest == [] or rest[0] != k]
```

For example, in `sums(5, 2)` with `k` equal to 2, the recursive call `sums(3, 2)` would first assign `rest` to `[1, 2]`, and so `[k] + rest` would be `[2, 1, 2]`. Then, it would assign rest to `[2, 1]` which would be skipped by the `if`, avoiding `[2, 2, 1]`, which has two adjacent 2's.

You can use [recursion visualizer][RecursionVisualizerSumFun] to step through the call structure of `sums(5, 3)`.

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
    """
    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:]))
```

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]]
    result = [[1] + rest for rest in nested_sums(n-1)]
    for k in range(1, n-1):
        for nest in nested_sums(k):
            result = result + [[1, nest] + rest for rest in nested_sums(n-k-1)]
    return result
```

The recursive case is that a nested sum of n 1's is either: - a 1 followed by the elements of a nested sum of n-1 1's, or - a 1 followed by a nested sum of k 1's followed by the elements of a nested sum of n-1-k 1's.

Here are some key ideas in translating this into code: - A `result` list can be built up by starting with an initial list and then adding more lists. - If `rest` is `[1, 1]`, then `[1] + rest` is `[1, 1, 1]`. - If `nest` and `rest` are both `[1, 1]`, then `[1, nest] + rest` is `[1, [1, 1], 1, 1]`. - In the expression `[... for rest in nested_sums(...)]`, `rest`

will be bound to each of the nested sums in the recursive call. For example, if `nested_sums(4)` was called, then `rest` would be bound to `[1, [1, 1], 1]` (among other options).

In the solution, the expression below creates lists of the first type: a 1 followed by the elements of a nested sum.

```
[[1] + rest for rest in nested_sums(n-1)]
```

For example, if `n` is 5, then `rest` iterates over all `nested_sums(4)` which include `[1, 1, 1, 1]`, `[1, 1, [1], 1]`, `[1, [1], 1, 1]`, and `[1, [1, 1], 1]`. If `rest` is the last of these, then `[1] + rest` is `[1, 1, [1, 1], 1]`. This line of code handles all cases that start with two 1's in a row.

The lines below create lists of the second type: a 1 followed by a nested list followed by the elements of another nested list. Schematically: `[1, [...], ...]`

```
for k in range(1, n-1):
    for nest in nested_sums(k):
        result = result + [[1, nest] + rest for rest in nested_sums(n-k-1)]
```

Here, `k` iterates over the number of 1's in the second element of the result. Therefore, `[1, nest]` has `k+1` 1's, and so `rest` needs the remaining `n-k-1` 1's. For example, if `n` is 5 and `k` is 2, then `nest` and `rest` would both be `[1, 1]`, forming `[1, [1, 1], 1, 1]`.