

Pick someone in your group to [join Discord](#). It's fine if multiple people join, but one is enough.

Now switch to Pensieve:

- **Everyone:** Go to discuss.pensieve.co and log in with your @berkeley.edu email, then enter your group number. (Your group number is the number of your Discord channel.)

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Pensieve doesn't work, return to this page and continue with the discussion.

Post in the [#help](#) channel on [Discord](#) if you have trouble.

Getting Started

Say your name and your favorite tree (a particular tree or a kind of tree) in honor of today's topic: tree recursion. (Tree recursive functions are functions that call themselves more than once.)

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

[New] If you have fewer than 4 people in your group, you can merge with another 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.

[For Fun] This emoticon of a guy in a cowboy hat is valid Python: `o[:-D]`

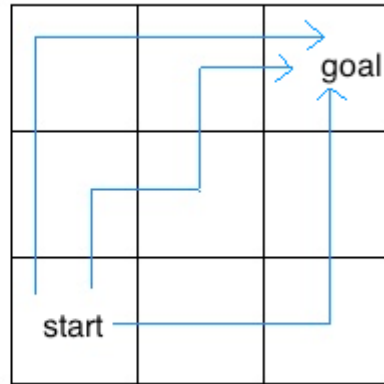
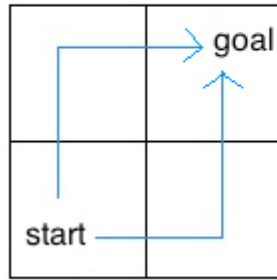
```
>>> o = [2, 0, 2, 4]
>>> [ o[:-D] for D in range(1,4) ]
[[2, 0, 2], [2, 0], [2]]
```

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`.

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).

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.
```

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.

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. Pick a presenter, then send a message to the discuss-queue channel with the @discuss tag, your discussion group number, and the message “Here’s the path!” and a member of the course staff will join your voice channel to hear your description and give feedback.

Tree Recursion with Lists

[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]`), **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.

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
    """
    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:]))
```

First try multiplying the first element by the `max_product` of everything after the first two elements (skipping the second element because it is consecutive with the first), then try skipping the first element and finding the `max_product` of the rest. To find which of these options is better, use `max`.

A great way to get help is to talk to the course staff!

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 ..."

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`.

```
>>> k = 2
>>> s = [4, 3, 1]
>>> [k] + s
[2, 4, 3, 1]
```

```
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
```

`k` is the first number in a list that sums to `n`, and `rest` is the rest of that list, so build a list that sums to `n`.

Call `sums` to build all of the lists that sum to `n-k` so that they can be used to construct lists that sum to `n` by putting

a `k` on the front.

Here is where you ensure that “no two adjacent numbers are the same.” Since `k` will be the first number in the list you’re building, it must not be equal to the first element of `rest` (which will be the second number in the list you’re building).

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 expression 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 and want to talk with the staff, post on Discord!

Document the Occasion

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