# **Tree Recursion**

A guide I wrote about problem solving strategies for **tree recursive** problems!

It’s quite long, but you can skip around to read the parts that you think you might find helpful.

The first section is on [what *is* tree recursion and how to recognize it](#_iuw3gtvjvbe0).

The second is just an overview on the difference between [tree recursion vs. recursion on trees](#_usa5nfo33l2k).

The last is essentially a problem walkthrough for max\_product from discussion 7, and you can apply the thinking process to [how to solve tree recursion problems](#_7ugz50t6v8p0) in general!

### 

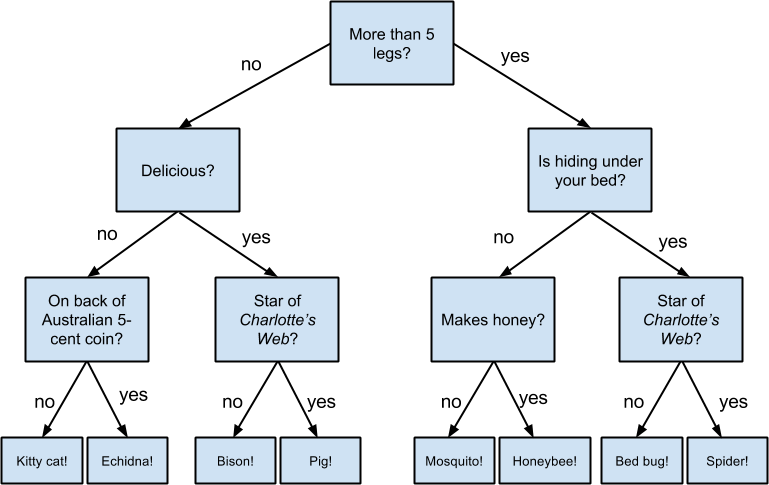
### **What is tree recursion & how to recognize it?**

Tree recursion is **recursion by case**, such that there are multiple **scenarios** you can/should consider to solve your problem.

For example, think of a *decision tree*:

* At your start point, you can go down the “Yes” or “No” path
* For each decision that you make, there’s another “Yes” or “No” that you can take, etc.
* From your start point, there are a bunch of different paths you can take to get to your final decision/outcome, based on what you chose to do previously.
* Your decisions end up “branching” out!

Example of a decision tree that’s being used to figure out what animal we have on hand ([credit](https://www.oreilly.com/library/view/data-science-from/9781491901410/ch17.html)):



With this example, we can see that at our start point, we have **two different pathways** we can diverge from: are there more or less than 5 legs?

Similarly, for tree recursive problems, to get to our **final solution/outcome**, we may need to take into account **multiple** situations.

Consider count\_stair\_ways we did in [discussion 3](https://cs61a.org/disc/disc03.pdf).

With this problem, in order to count the **total number of ways** we could climb the stairway, we had to take into consideration **two** scenarios at our “start” point:

1. How many ways can I climb the stairway if I take just **one** step first?
2. How many ways can I climb the stairway if I take **two** steps first?

The **total** number of ways I can climb the staircase is just the sum of these two scenarios.

Why? Consider for my **very first move**, is it possible to take a number of steps apart from just one or two?

Nope! We only have **two** options at our start point: **one or two steps**. The number of ways to climb the staircase associated with these two options consist of the entire outcome space/all the possibilities/combinations that we can have for this problem.

So, in the end, tree recursion is just **recursion that you need to take into account multiple situations/scenarios for**.

If you’re given a problem that gives you *choices* or that you’re forced to think about *different situations/cases* in order to solve it, you are most likely looking at a tree recursive problem!

### 

### **Tree recursion vs. recursion *on* trees?**

A common question is: what’s the difference between **tree recursion** and **recursion *on* trees**?

It can be confusing, because not only do the two have incredibly similar names, but they are also very similar in terms of problem solving strategy/problem setup.

In fact, whenever we want to solve a problem regarding the **tree abstract data type**, we generally think about using recursion ***on*** these trees. And to apply recursion ***on*** trees, we usually use the principles of **tree recursion**!

Why does this make sense?

Let’s think back to how we defined tree recursion (all of the following are synonymous):

* recursion by **case**
* recursion by **situation**
* recursion on multiple **pathways**/**choices**

The decision tree we saw earlier while defining “tree recursion” was called a decision **tree** because of its branching structure. Each choice gave way to another pathway/branch. Its multiple-situation structure gave way to a shape that looks just like the **trees** we work with in CS (hence the name “decision **tree**” and **tree** recursion). That’s why it makes sense to use **tree recursive** techniques on *actual* trees, as **tree recursion** allows us to address all the different situations we may want to address when it comes to our tree problem.

### **How to solve tree recursion problems?**

Let’s look at the following question max\_product from [discussion 7](https://cs61a.org/disc/disc07_sol.pdf):

|  |
| --- |
| def max\_product(lst): """Return the maximum product that can be formed using lst without using any consecutive numbers >>> max\_product([10,3,1,9,2]) # 10 \* 9 90 >>> max\_product([5,10,5,10,5]) # 5 \* 5 \* 5 125 >>> max\_product([]) 1 """ |

First, let’s break down the **problem statement** in terms of action items we have to hit.

Remember when you read questions to look closely at the problem definition & the doctests to figure out what it is exactly that you need to do:  
**Problem given Our interpretation**

|  |  |
| --- | --- |
| Write a function that **takes in a list** | **Input**: list of numbers |
| **Returns the maximum product** that can be formed using **non-consecutive elements** of the list. | **Output**: a number  **Restriction**: the product cannot be formed by consecutive (i.e. adjacent) elements |
| The input list will contain only numbers **greater than or equal to 1**. | **Output**: a number that is >= 1  (Also returns 1) |
| Note the doctest: max\_product([]) | **Output**: 1 |

Why are each of these things important?

Whenever we write functions, *especially* **recursive** functions,   
we always want to think about the **domain and range**.

What is being passed in?

What is being returned?

Having a solid understanding of the domain and range help us when we try to apply the “[recursive leap of faith](https://docs.google.com/document/d/1p0-DWUZTEuSPxJvMK43yODzFWSxOKSi9wfL8J0FClio/edit?usp=sharing)” to help us solve the problem! You can read more in the linked guide, if you’d like a refresher on what it means to take that “recursive leap of faith.”

Alright, now back to the problem at hand.

First, how can we recognize that this is a tree recursive question?

* Well, we’re not dealing with an explicit tree structure . . . so it might not be super obvious we need to use tree recursion.
* But, let’s think about the problem a little bit more.
* What exactly is it asking us?
* We need to find the **maximum product** in the list without consecutive numbers.
* **Maximum** product tells us that there is probably *more than one* products that can be calculated with the list passed in . . . looks like a problem with multiple different “pathways”/scenarios to me!

If you wanted to be even more explicit, you could draw out a tree diagram yourself about what a “decision” tree might look like for this problem statement to really help you visualize this problem and the various pathways that exist that lead to the **tree recursive** nature.

But alright, now that we have some idea that perhaps this might be tree recursion, how might we go about solving this? As a reminder, tree recursion is still just **recursion** with multiple recursive calls (one for each ‘scenario’ that we’re considering). And, with all recursion, we generally think of the following three steps:

1. Base case
2. Recursive call
3. Using the recursive call to piece things together

So what’s our **base case**?

Well, we saw earlier in the table we created via the doctests that max\_product([]) returns 1.

Looks like a solid base case to me — you can’t get an input smaller than an empty list!

|  |
| --- |
| if lst == []:  return 1 |

Alright, looks like a good first step. Now let’s actually move into the recursive calls.

Let’s think about **what** this problem is asking.

It wants us to look for the **maximum product** of the list **without** consecutive numbers.

In general, when we look for a *product* of numbers within a list, how might we go about that?

Well, we’d be just multiplying elements of the list together right!

So given some list with some numbers, **how do we know which numbers we want to use** for that maximum product? That’s a really great question to frame for our decision tree! We want to focus on *what* numbers are going to end up being in our final maximum product multiplication. As in, we can basically look at each number in the list and ask: are you going to be in the final product’s multiplication sequence or not?

Okay so let’s consider that problem then:

Is the first element of the list going to be in the maximum product?

This is a **yes** or **no** question.

So let’s define two variables that represent this yes/no pathway:

* with\_first ⇒ the maximum product of the list *if we used the first element*
* without\_first ⇒ the maximum product of the list *if we* ***didn’t*** *use the first element*

Real quick, before we jump into the recursion and actually calculate the values of the variables, how might we use these two values to get the **maximum product** of the list as a whole?

|  |
| --- |
| return max(with\_first, without\_first) |

We know that the maximum product has to come from either *using* the first element, or *not* using the first element — there’s no other possibility! There’s no way we could get a product in our list that doesn’t use one of these two situations (either first element is present, or it’s not present). You can also think of this as a **decision tree with two initial branches!**

Okay so now that we know we’ve successfully covered all the potential pathways with this, let’s think about how we would actually get the values of with\_first and without\_first!

with\_first we said was the maximum product of the list **if we used the first element**.

* using the first element means that lst[0] is a factor in the total product
* okay, now that we have lst[0] confirmed, what’s our next step?
* recall what it means to be a maximum product of the list in the context of with\_first
  + lst[0] \* **valid** maximum product of the rest of the list
* okay now let’s break down “**valid** maximum product of the rest of the list “
* first, what do we mean by “**valid**”? well, remember we have a **restriction** in this problem we need to consider: **non consecutive elements only**.
  + therefore, is lst[1] a valid factor for with\_first? nope!
  + only elements in lst[2:] onwards are **valid** to be considered in the final product, since we decided for sure to use lst[0] for with\_first
* ok! now we know using lst[0], means looking at lst[2:] for the rest of the product.
* how do we obtain the maximum product for lst[2:], to combine with lst[0] to get the total max product in the context of using lst[0] for with\_first?
* we have a function for that! this is where our recursive call comes in: max\_product(lst[2:])

Hence, with\_first = lst[0] \* max\_product(lst[2:])

Following a similar logic for without\_first, we can reach the conclusion that

without\_first = max\_product(lst[1:])

* This is when we **don’t** use lst[0]
* So the valid part of the list we can use to get the maximum product of lst, given that we don’t use lst[0] is going to be lst[1:]
* What’s the max product of lst[1:]?
* We have a function for that, max\_product!

So the final solution will be:

|  |
| --- |
| if lst == []:  return 1 with\_first = lst[0] \* max\_product(lst[2:]) without\_first = max\_product(lst[1:]) return max(with\_first, without\_first) |

This is also very similar to count partitions, if you remember that!