L12

AI Game Development & Computer Science Summer Program

Lecture 12: Recursion II: Chains and Trees

The pattern stays the same when the data shape changes

Name the stop case, make the input smaller, then combine the smaller answer.

Function	Stop case	Smaller input	Combine
countNodes	no node	node.next	1 + rest
contains	no node or match	node.next	return rest
countMatches	no node	node.next	here + rest
factorial	0 or invalid	num - 1	num * rest
tree helpers	no root	left and right	current + branches

Recursive code is easier to write when this table is clear before typing.

countNodes follows next until the chain ends

The base case gives an answer for the missing link.


```
function countNodes(node: ListNode | undefined): number {
  if (node === undefined) return 0;

  return 1 + countNodes(node.next);
}
```

Trace habit

- Ask what this call can see now.
- Move only to `node.next`.
- Add one as the calls return.

contains has two ways to stop

An empty chain is false. A matching current node is true.

```
function contains(
  node: ListNode | undefined,
  target: string,
): boolean {
  if (node === undefined) return false;
  if (node.value === target) return true;

  return contains(node.next, target);
}
```

Search order

- Check the missing-node case first.
- Check the current value second.
- Only recurse if the answer is not known yet.

Predict

- `contains(a, "miss")` stops at B.
- `contains(a, "coin")` reaches the missing link.

countMatches keeps one local answer and asks for the rest

```
function countMatches(
  node: ListNode | undefined,
  target: string,
): number {
  if (node === undefined) return 0;

  const here = node.value === target ? 1 : 0;
  return here + countMatches(node.next, target);
}
```

Call	here	rest	return
hit	1	1	2
miss	0	1	1
hit	1	0	1
end	-	-	0

factorial is the same one-step shrink without nodes

```
function factorial(num: number): number | undefined {
  if (num < 0) return undefined;
  if (num === 0) return 1;

  const next = factorial(num - 1);
  if (next === undefined) return undefined;

  return num * next;
}
```

Trace

- 4! = 4 * 3 * 2 * 1
- `factorial(0)` returns 1.
- Each call saves `num` until the smaller answer returns.

Invalid input

- A negative number is not part of this function's useful domain.
- The return type says that clearly.

When a smaller call might fail, handle that result first

TypeScript will not let you multiply by `undefined` unless you ignore the type system.

Why `number | undefined`?

- The function returns a number for valid inputs.
- It returns `undefined` for negative inputs.
- The caller must handle both possibilities.

Why save `next`?

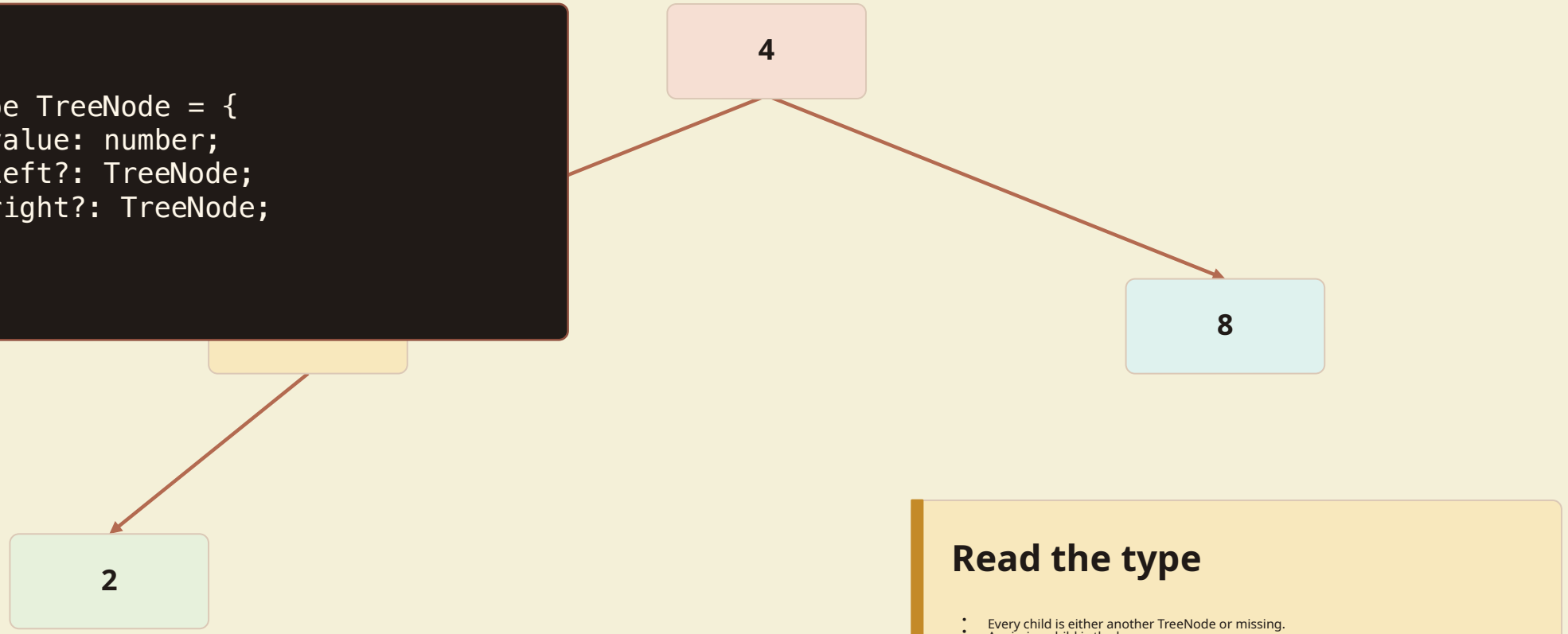
- `factorial(num - 1)` might be undefined.
- Check it before multiplying.
- Then combine with `num`.

```
const next = factorial(num - 1);

if (next === undefined) {
  return undefined;
}

return num * next;
```

A tree node has two smaller problems

Instead of one `next`, a binary tree has `left` and `right`.

```
type TreeNode = {
  value: number;
  left?: TreeNode;
  right?: TreeNode;
};
```



Read the type

- Every child is either another TreeNode or missing.
  A missing child is the base case.

countTreeNodes counts the current node plus both branches

This is tree recursion because one call makes two recursive calls.

```
function countTreeNodes(root: TreeNode | undefined): number {
  if (root === undefined) return 0;

  return 1
    + countTreeNodes(root.left)
    + countTreeNodes(root.right);
}
```

Base case

- A missing child contributes 0 nodes.

Combine step

- Count this node.
- Count the left branch.
- Count the right branch.
- Add the three answers.

sumTreeNodes changes the current value, not the traversal

The shape is the same as countTreeNodes.

```
function sumTreeNodes(root: TreeNode | undefined): number {
  if (root === undefined) return 0;

  return root.value
    + sumTreeNodes(root.left)
    + sumTreeNodes(root.right);
}
```

Helper	Current value	Left	Right
count	1	count(left)	count(right)
sum	root.value	sum(left)	sum(right)

A good recursion exercise often changes only one part of the combine step.

sumEvenTreeNodes filters before it combines

```
function sumEvenTreeNodes(root: TreeNode | undefined): number {
  if (root === undefined) return 0;

  const here = root.value % 2 === 0 ? root.value : 0;

  return here
    + sumEvenTreeNodes(root.left)
    + sumEvenTreeNodes(root.right);
}
```

Modulo check

- `value % 2 === 0` means the value is even.
- Odd values do not disappear; their branches still count.

Why this matters

- Filtering the current node is separate from traversing the tree.

Trace trees from leaves back to the root

A parent can combine branch answers only after the branches return.

Call	count	sum	sumEven
empty child	0	0	0
node 2	1	2	2
node 3 with left 2	2	5	2
node 8	1	8	8
root 4	4	17	14

Use the same tree

- Root: 4
- Left branch: 3 with left child 2
- Right branch: 8

Check

- Even values are 4, 2, and 8.
- 4 + 2 + 8 = 14.

Tree recursion means more than one smaller call

The tree shape tells the function to ask both branches.

Linear recursion

- One smaller call.
- `node.next`
- `num - 1`

Tree recursion

- Two or more smaller calls.
- `root.left`
- `root.right`

Combine rule

- Add branch answers for counts and sums.
- Use `||` or `&&` for boolean searches.
- Keep the base case simple.

Practice by changing one part at a time

Reuse a working shape instead of starting from a blank page.

Chain helpers

- Trace `countNodes`.
- Add a match base case for `contains`.
- Add `here + rest` for `countMatches`.

Factorial

- Handle negative input.
- Stop at 0.
- Multiply by the smaller answer only after it exists.

Tree helpers

- Count nodes.
- Sum values.
- Sum only even values while still visiting every branch.

Most bugs come from stopping, shrinking, or combining incorrectly

Debug recursive code by naming which part is wrong.

Stop bug

- No empty-chain or empty-tree case.
- No invalid-input case for factorial.

Shrink bug

- Calling the function again with the same node.
- Using `num` instead of `num - 1`.

Combine bug

- Forgetting one branch.
- Adding odd values in `sumEvenTreeNodes`.
- Multiplying before checking `undefined`.

Lab block: finish the same shapes with checks

Write less code, but explain each base case and smaller input.

Complete

- `countNodes`
- `contains`
- `countMatches`
- `factorial`

Extend

- `countTreeNodes`
- `sumTreeNodes`
- `sumEvenTreeNodes`

Prove

- Trace empty input.
- Trace one multi-node chain.
- Trace one tree from leaves to root.

Lab plan: practice, build, apply

Use the main lab block to turn the concept into working evidence.

Guided practice

- Trace `countNodes`, `contains`, and `countMatches` on `hit -> miss -> hit`.
- Trace `factorial(4)` and `factorial(-1)` before coding.
- Trace count, sum, and sum-even on one small tree.

Build target

- Complete `countNodes`, `contains`, and `countMatches`.
- Complete `factorial` with a negative-input guard.
- Complete `countTreeNodes`, `sumTreeNodes`, and `sumEvenTreeNodes`.

Project time

- Save the recursion trace as CS exercise evidence.
- Use a recursive helper in P1 only if the game rule naturally has a recursive shape.
- Otherwise, keep P1 simple and treat this as core CS practice.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Write `containsTree(root, target)` for numbers.
- Write `maxTreeValue(root)` and decide what it should return for an empty tree.
- Ask AI for a wrong `sumEvenTreeNodes` implementation and identify the bug.

Done when

- The main lab still works.
- You can explain the tradeoff or edge case.
- You saved one proof object: test, trace, screenshot, or note.

P1: Game Rules Milestone

L03-L15

Checkpoint

Move on when the type tells you which smaller calls to make.

Exit ticket

- One chain helper with a base case.
- One factorial trace that reaches 0.
- One tree helper that uses left and right.
- One test or trace for empty input.

Next class preview

- Branching recursion as search.
- Success and failure base cases.
- Counting possible paths.