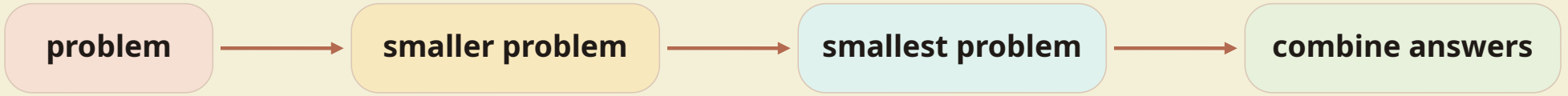
L11

AI Game Development & Computer Science Summer Program

Lecture 11: Recursion I: Self-Similar Problems

Recursion solves a problem by solving a smaller version

The function calls itself on a simpler input.



Key question

- Can the big problem be described as one step plus a smaller problem of the same kind?

Every recursive function needs two parts

One part stops. One part makes progress.

Base case

- The input is simple enough to answer directly.
- No self-call happens here.
- This prevents infinite recursion.

Recursive case

- Break the problem into a smaller input.
- Call the same function on that smaller input.
- Combine the smaller answer with the current step.

sumTo is the simplest recursion shape

Stop at zero. Otherwise add the current number to the smaller sum.

```
const sumTo = (n: number): number => {
  if (n <= 0) return 0;
  return n + sumTo(n - 1);
};
```

Trace

- sumTo(3)
- 3 plus sumTo(2)
- 2 plus sumTo(1)
- 1 plus sumTo(0)
- then add on the way back

Trace calls and returns separately

Calls go down to the base case. Returns build the answer back up.

Call	Action	Return
sumTo(3)	wait for sumTo(2)	6
sumTo(2)	wait for sumTo(1)	3
sumTo(1)	wait for sumTo(0)	1
sumTo(0)	base case	0

Trace habit

- Indent the smaller calls.
- Do not combine until the smaller call returns.

Divide, invoke, combine

Use this checklist before writing recursive code.

Divide

- What is one piece you can handle now?
- What smaller input remains?

Invoke

- Call the same function on the smaller input.
- Trust that it returns the smaller answer.

Combine

- Merge the current piece with the smaller answer.
- Return the full answer.

Example 2: count matches in a node chain

```
type ListNode = { value: string; next?: ListNode };

const countMatches = (node: ListNode | undefined, target:
string): number => {
  if (!node) return 0;
  const here = node.value === target ? 1 : 0;
  return here + countMatches(node.next, target);
};
```

Self-similar shape

- Current work: check this node.
- Smaller problem: count from `next`.
- Base case: missing node returns 0.

Trust the smaller call, but verify progress

The recursive call must move toward the base case.

Trust

- Assume `countMatches(node.next, target)` correctly counts the smaller chain.
- Then focus on how to combine the current node with that answer.

Verify

- Is the input smaller?
- Will it eventually reach the base case?
- Does every branch return the declared type?

Trust is a writing strategy. Tracing and tests are the proof.

Recursion and loops can solve the same kinds of problems

Pick the form that makes the structure clearer.

Loop is often clearer when

- You walk through a chain without branching.
- You keep a simple accumulator.
- The problem does not contain smaller nested versions.

Recursion is often clearer when

- The data is nested.
- The problem is naturally self-similar.
- The answer is built from smaller answers.

Common recursion bugs

Most recursion bugs are missing base cases or no progress.

No base case

- The function never stops calling itself.
- Add a direct answer for the simplest input.

No progress

- The recursive call uses the same input.
- Make the input smaller or simpler.

Wrong combine

- The smaller answer is correct, but the current step joins it incorrectly.
- Trace returns upward.

Lab block: trace, fix, then build

Use the rest of class to practice the same idea in several ways.

Guided practice

- Trace sumTo(4) by hand.
- Fix one missing base case.
- Explain why the input gets smaller.

Project practice

- Find one repeated structure in your project idea.
- Decide whether a loop or recursion would be clearer.
- Use AI for a trace only after your own table.

The goal is fluency with base case, progress, and combine steps, not memorizing recursion tricks.

Lab build: recursive count

Write a recursive helper and prove that it stops.

```
const countMatches = (node: ListNode | undefined, target:
string): number => {
  if (!node) return 0;
  const current = node.value === target ? 1 : 0;
  return current + countMatches(node.next, target);
};
```

Build order

- Write the empty-chain base case.
- Check the current node.
- Trust the smaller count from `next`.
- Combine current plus smaller answer.
- Trace three calls.

Lab plan: practice, build, apply

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

Guided practice

- Trace sumTo calls and returns.
- Identify base case, smaller input, and combine step.
- Explain why recursion can be useful even when a loop would also work.

Build target

- Complete a recursive ListNode count-matches call table.
- Complete at least three L11 recursion puzzles as short stations.
- Optionally build the recursive helper after the table is correct.

Project time

- Look for a repeated structure in P1.
- Decide whether recursion or a loop is clearer for the actual project.

P1: Game Rules Milestone

L03-L15

Bonus activities

Choose one if the main lab is solid and tested.

Stretch options

- Add a recursive contains helper.
- Compare recursive and loop versions.
- Ask AI for a call table and correct one row.

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 base case and smaller input are obvious.

Exit ticket

- One base case and its return value.
- One recursive case that makes progress.
- One divide, invoke, combine explanation.
- One recursion bug and how to spot it.
- One AI-generated trace row you verified.

Next class preview

- More ListNode chains.
- Tree-shaped nodes.
- Solving problems shaped like smaller copies.