COMP101 — Recursion & Recurrence

Towers of Hanoi, GCD, Fibonacci

UM6P — SASE

December 19, 2025

Today's goals

- Understand what recursion is (in code and in problem solving)

Today's goals

- Understand what recursion is (in code and in problem solving)
- Visualize the function call stack and what really happens at runtime

Today's goals

- Understand what recursion is (in code and in problem solving)
- Visualize the function call stack and what really happens at runtime
- Turn recursive thinking into a recurrence for the cost of an algorithm

Today's goals

- Understand what recursion is (in code and in problem solving)
- Visualize the function call stack and what really happens at runtime
- Turn recursive thinking into a recurrence for the cost of an algorithm
- Recognize common recursive patterns (Hanoi, factorial, sum, GCD, Fibonacci)

Today's goals

- Understand what recursion is (in code and in problem solving)
- Visualize the function call stack and what really happens at runtime
- Turn recursive thinking into a recurrence for the cost of an algorithm
- Recognize common recursive patterns (Hanoi, factorial, sum, GCD, Fibonacci)
- See when recursion is a bad idea and how memoization can rescue us

Let's play a game

- Game: Towers of Hanoi

Let's play a game

- Game: Towers of Hanoi
- Rules:

Let's play a game

- Game: Towers of Hanoi
- Rules:
  - Move one disk at a time

Let’s play a game

- Game: Towers of Hanoi
- Rules:
    - Move one disk at a time
    - Never place a larger disk on top of a smaller one

Let's play a game

- Game: Towers of Hanoi
- Rules:
  - Move one disk at a time
  - Never place a larger disk on top of a smaller one
  - Goal: move the full tower from source peg to target peg

Let's play a game

- Game: Towers of Hanoi
- Rules:
    - Move one disk at a time
    - Never place a larger disk on top of a smaller one
    - Goal: move the full tower from source peg to target peg
- Link to online game :

  <https://www.mathsisfun.com/games/towerofhanoi.html>

Let's play a game

- Game: Towers of Hanoi
- Rules:
  - Move one disk at a time
  - Never place a larger disk on top of a smaller one
  - Goal: move the full tower from source peg to target peg
- Link to online game :

  <https://www.mathsisfun.com/games/towerofhanoi.html>
- We will:

Let's play a game

- Game: Towers of Hanoi
- Rules:
  - Move one disk at a time
  - Never place a larger disk on top of a smaller one
  - Goal: move the full tower from source peg to target peg
- Link to online game :

  <https://www.mathsisfun.com/games/towerofhanoi.html>
- We will:
  - Play for small numbers of disks (1, 2, 3, 4)

Let's play a game

- Game: Towers of Hanoi
- Rules:
  - Move one disk at a time
  - Never place a larger disk on top of a smaller one
  - Goal: move the full tower from source peg to target peg
- Link to online game :

  <https://www.mathsisfun.com/games/towerofhanoi.html>
- We will:
  - Play for small numbers of disks (1, 2, 3, 4)
  - Try to guess the minimum number of moves

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other
- To move a tower of n disks from source to target:

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other
- To move a tower of n disks from source to target:
  - Step 1: Magically move the top $n-1$ disks from source to auxiliary

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other
- To move a tower of n disks from source to target:
  - Step 1: Magically move the top $n-1$ disks from source to auxiliary
  - Step 2: Move the largest disk (disk n) from source to target

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other
- To move a tower of n disks from source to target:
  - Step 1: Magically move the top $n-1$ disks from source to auxiliary
  - Step 2: Move the largest disk (disk n) from source to target
  - Step 3: Magically move the $n-1$ disks from auxiliary to target

What if we can magically move $n-1$ disks?

- Imagine we already know how to perfectly move a tower of $n-1$ disks from any peg to any other
- To move a tower of n disks from source to target:
  - Step 1: Magically move the top $n-1$ disks from source to auxiliary
  - Step 2: Move the largest disk (disk n) from source to target
  - Step 3: Magically move the $n-1$ disks from auxiliary to target
- This is the core recursive pattern: solve two smaller copies of the same problem, plus one simple step

Independent subproblems

- Subproblem 1: move $n-1$ disks from source to auxiliary

Independent subproblems

- Subproblem 1: move $n - 1$ disks from source to auxiliary
- Subproblem 2: move $n - 1$ disks from auxiliary to target

Independent subproblems

- Subproblem 1: move $n - 1$ disks from source to auxiliary
- Subproblem 2: move $n - 1$ disks from auxiliary to target
- Once we lift disk n, what happens inside the smaller tower does not care about disk n

Independent subproblems

- Subproblem 1: move $n-1$ disks from source to auxiliary
- Subproblem 2: move $n-1$ disks from auxiliary to target
- Once we lift disk n, what happens inside the smaller tower does not care about disk n
- This independence is what makes recursion natural:

Independent subproblems

- Subproblem 1: move $n-1$ disks from source to auxiliary
- Subproblem 2: move $n-1$ disks from auxiliary to target
- Once we lift disk n, what happens inside the smaller tower does not care about disk n
- This independence is what makes recursion natural:
  - Each smaller tower is a smaller copy of the original problem

Independent subproblems

- Subproblem 1: move $n - 1$ disks from source to auxiliary
- Subproblem 2: move $n - 1$ disks from auxiliary to target
- Once we lift disk n, what happens inside the smaller tower does not care about disk n
- This independence is what makes recursion natural:
  - Each smaller tower is a smaller copy of the original problem
- Our whole solution will be: base case for tiny towers, plus this splitting for bigger towers

- The full Hanoi state we care about can be encoded by:

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is
  - target: peg where we want this tower to end up

Encoding the problem

- The full Hanoi state we care about can be encoded by:
    - n: number of disks in the tower we are moving
    - source: peg where this tower currently is
    - target: peg where we want this tower to end up
    - aux: the remaining peg we can use as helper

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is
  - target: peg where we want this tower to end up
  - aux: the remaining peg we can use as helper
- A single move is fully determined by:

Encoding the problem

- The full Hanoi state we care about can be encoded by:
    - n: number of disks in the tower we are moving
    - source: peg where this tower currently is
    - target: peg where we want this tower to end up
    - aux: the remaining peg we can use as helper
- A single move is fully determined by:
    - Which peg we take from (source of the move)

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is
  - target: peg where we want this tower to end up
  - aux: the remaining peg we can use as helper
- A single move is fully determined by:
  - Which peg we take from (source of the move)
  - Which peg we place on (target of the move)

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is
  - target: peg where we want this tower to end up
  - aux: the remaining peg we can use as helper
- A single move is fully determined by:
  - Which peg we take from (source of the move)
  - Which peg we place on (target of the move)
- Because we can only touch the top disk of any peg, the actual disk moved is implicit

Encoding the problem

- The full Hanoi state we care about can be encoded by:
  - n: number of disks in the tower we are moving
  - source: peg where this tower currently is
  - target: peg where we want this tower to end up
  - aux: the remaining peg we can use as helper
- A single move is fully determined by:
  - Which peg we take from (source of the move)
  - Which peg we place on (target of the move)
- Because we can only touch the top disk of any peg, the actual disk moved is implicit
- So our recursive function will only need: $\mathrm{hanoi}(n, \mathrm{source}, \mathrm{target}, \mathrm{aux})$

- We want a function: hanoi(n, source, target, aux)

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target
- Recursive case idea for $n \geq 2$:

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target
- Recursive case idea for $n \geq 2$:
  - Move $n - 1$ disks from source to aux

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target
- Recursive case idea for $n \geq 2$:
  - Move $n - 1$ disks from source to aux
  - Move disk n from source to target

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target
- Recursive case idea for $n \geq 2$:
  - Move $n - 1$ disks from source to aux
  - Move disk n from source to target
  - Move $n - 1$ disks from aux to target

Deriving the recursive solution

- We want a function: hanoi(n, source, target, aux)
- Base case idea:
  - If $n = 0$: nothing to do
  - If $n = 1$: single move from source to target
- Recursive case idea for $n \geq 2$:
  - Move $n - 1$ disks from source to aux
  - Move disk n from source to target
  - Move $n - 1$ disks from aux to target
- We will write the Python code live following this structure

Hanoi — recursive Python skeleton

```python
def hanoi(n, source, target, auxiliary):
    """Print moves to solve Towers of Hanoi with n disks."""
    if n == 0:
        return

    # TODO: move top n-1 disks from source to auxiliary

    # TODO: move disk n from source to target

    # TODO: move n-1 disks from auxiliary to target
```

- We will fill in the three TODOs together

Hanoi — recursive Python skeleton

```python
def hanoi(n, source, target, auxiliary):
    """Print moves to solve Towers of Hanoi with n disks."""
    if n == 0:
        return

    # TODO: move top n-1 disks from source to auxiliary

    # TODO: move disk n from source to target

    # TODO: move n-1 disks from auxiliary to target
```

- We will fill in the three TODOs together
- Notice how the function calls itself on a smaller value of n

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:
    - Move $n-1$ disks: costs $T(n-1)$ moves

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:
  - Move $n-1$ disks: costs $T(n-1)$ moves
  - Move disk n: costs 1 move

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:
    - Move $n-1$ disks: costs $T(n-1)$ moves
    - Move disk n: costs 1 move
    - Move $n-1$ disks again: costs another $T(n-1)$ moves

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:
  - Move $n-1$ disks: costs $T(n-1)$ moves
  - Move disk n: costs 1 move
  - Move $n-1$ disks again: costs another $T(n-1)$ moves
- So we get the recurrence:

$$T(1) = 1, \qquad T(n) = 2\,T(n-1) + 1 \quad \text{for } n \geq 2$$

How many moves does it take?

- Let $T(n)$ be the minimum number of moves to solve Hanoi with n disks
- From our algorithm:
  - Move $n-1$ disks: costs $T(n-1)$ moves
  - Move disk n: costs 1 move
  - Move $n-1$ disks again: costs another $T(n-1)$ moves
- So we get the recurrence:

$$T(1) = 1, \qquad T(n) = 2\,T(n-1) + 1 \quad \text{for } n \geq 2$$

- This is our first serious recurrence relation

From $T(n) = 2T(n-1) + 1$ to a closed form

- Unroll a few steps:

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2\bigl(2T(n-2) + 1\bigr) + 1 = 4T(n-2) + 3 \\
&= 4\bigl(2T(n-3) + 1\bigr) + 3 = 8T(n-3) + 7 \\
&\vdots
\end{aligned}$$

From $T(n) = 2T(n-1) + 1$ to a closed form

- Unroll a few steps:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2\bigl(2T(n-2) + 1\bigr) + 1 = 4T(n-2) + 3 \\ &= 4\bigl(2T(n-3) + 1\bigr) + 3 = 8T(n-3) + 7 \\ &\ \vdots \end{aligned}$$

- After k steps, pattern:

$$T(n) = 2^k T(n-k) + (2^k - 1)$$

From $T(n) = 2T(n-1) + 1$ to a closed form

- Unroll a few steps:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2\big(2T(n-2) + 1\big) + 1 = 4T(n-2) + 3 \\ &= 4\big(2T(n-3) + 1\big) + 3 = 8T(n-3) + 7 \\ &\vdots \end{aligned}$$

- After k steps, pattern:

$$T(n) = 2^k T(n-k) + (2^k - 1)$$

- Take $k = n - 1$ and use $T(1) = 1$:

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

From $T(n) = 2T(n-1) + 1$ to a closed form

- Unroll a few steps:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2\big(2T(n-2) + 1\big) + 1 = 4T(n-2) + 3 \\ &= 4\big(2T(n-3) + 1\big) + 3 = 8T(n-3) + 7 \\ &\vdots \end{aligned}$$

- After k steps, pattern:

$$T(n) = 2^k T(n-k) + (2^k - 1)$$

- Take $k = n - 1$ and use $T(1) = 1$:

$$T(n) = 2^{n-1}T(1) + (2^{n-1} - 1) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$$

- So:

$$\boxed{T(n) = 2^n - 1} \quad \text{(exponential growth)}$$

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
    - If you split the problem into a subproblems of size n/b

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
    - If you split the problem into a subproblems of size n/b
    - And pay an extra cost $f(n)$ to combine results

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
  - If you split the problem into a subproblems of size n/b
  - And pay an extra cost $f(n)$ to combine results
- Master Theorem tells you how $T(n)$ grows (e.g., $O(n \log n)$, $O(n^2)$, ...)

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
  - If you split the problem into a subproblems of size n/b
  - And pay an extra cost $f(n)$ to combine results
- Master Theorem tells you how $T(n)$ grows (e.g., $O(n \log n)$, $O(n^2)$, $\dots$)
- Our Hanoi recurrence is different ($n - 1$ instead of $n/2$) but it shows:

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
  - If you split the problem into a subproblems of size n/b
  - And pay an extra cost $f(n)$ to combine results
- Master Theorem tells you how $T(n)$ grows (e.g., $O(n \log n)$, $O(n^2)$, …)
- Our Hanoi recurrence is different ($n-1$ instead of $n/2$) but it shows:
  - Recurrence with $T(n-1)$ often leads to exponential growth

Glimpse: Master Theorem intuition

- Many divide-and-conquer algorithms satisfy recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

- Rough intuition:
  - If you split the problem into a subproblems of size n/b
  - And pay an extra cost $f(n)$ to combine results
- Master Theorem tells you how $T(n)$ grows (e.g., $O(n \log n)$, $O(n^2)$, $\ldots$)
- Our Hanoi recurrence is different ($n-1$ instead of $n/2$) but it shows:
  - Recurrence with $T(n-1)$ often leads to exponential growth
  - Recurrence with $T(n/2)$ often leads to logarithmic depth

What does it mean for a function to be recursive?

- A function is recursive if it calls itself

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:
  - It solves a problem by calling itself on a smaller (simpler) input

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:
  - It solves a problem by calling itself on a smaller (simpler) input
  - It must have at least one base case that does not call itself

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:
  - It solves a problem by calling itself on a smaller (simpler) input
  - It must have at least one base case that does not call itself
- Every recursive function implies a call stack:

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:
  - It solves a problem by calling itself on a smaller (simpler) input
  - It must have at least one base case that does not call itself
- Every recursive function implies a call stack:
  - Each call waits for the result of the next call

What does it mean for a function to be recursive?

- A function is recursive if it calls itself
- More precisely:
  - It solves a problem by calling itself on a smaller (simpler) input
  - It must have at least one base case that does not call itself
- Every recursive function implies a call stack:
  - Each call waits for the result of the next call
  - When the base case hits, we unwind the stack back up

Common pitfalls with recursion

- Missing or incorrect base case:

Common pitfalls with recursion

- Missing or incorrect base case:
    - Leads to infinite recursion until the program crashes

Common pitfalls with recursion

- Missing or incorrect base case:
  - Leads to infinite recursion until the program crashes
- No progress toward the base case:

Common pitfalls with recursion

- Missing or incorrect base case:
  - Leads to infinite recursion until the program crashes
- No progress toward the base case:
  - Example: calling the function again with the same argument

Common pitfalls with recursion

- Missing or incorrect base case:
  - Leads to infinite recursion until the program crashes
- No progress toward the base case:
  - Example: calling the function again with the same argument
- In Python: RecursionError: maximum recursion depth exceeded

Common pitfalls with recursion

- Missing or incorrect base case:
  - Leads to infinite recursion until the program crashes
- No progress toward the base case:
  - Example: calling the function again with the same argument
- In Python: RecursionError: maximum recursion depth exceeded
  - Each call uses stack space

Common pitfalls with recursion

- Missing or incorrect base case:
  - Leads to infinite recursion until the program crashes
- No progress toward the base case:
  - Example: calling the function again with the same argument
- In Python: RecursionError: maximum recursion depth exceeded
  - Each call uses stack space
  - Deep recursion hits the interpreter limit

Python recursion limit (handle with care)

- Python protects you with a default maximum recursion depth

```
import sys

print(sys.getrecursionlimit())   # default is often ~1000

# Increase the limit (dangerous if you have a bug!):
sys.setrecursionlimit(10**6)
```

Python recursion limit (handle with care)

- Python protects you with a default maximum recursion depth
- You can inspect and change it:

```
import sys

print(sys.getrecursionlimit())  # default is often ~1000

# Increase the limit (dangerous if you have a bug!):
sys.setrecursionlimit(10**6)
```

Python recursion limit (handle with care)

- Python protects you with a default maximum recursion depth
- You can inspect and change it:

```
import sys

print(sys.getrecursionlimit())  # default is often ~1000

# Increase the limit (dangerous if you have a bug!):
sys.setrecursionlimit(10**6)
```

- This is a last resort, not a fix for bad recursion design

Python recursion limit (handle with care)

- Python protects you with a default maximum recursion depth
- You can inspect and change it:

```
import sys

print(sys.getrecursionlimit())   # default is often ~1000

# Increase the limit (dangerous if you have a bug!):
sys.setrecursionlimit(10**6)
```

- This is a last resort, not a fix for bad recursion design
- First: fix your base cases, reduce depth, or use an iterative approach

To the drawing board: call stack

- On the board, we will trace:

To the drawing board: call stack

- On the board, we will trace:
  - A small example like hanoi(3, 'A', 'C', 'B') or fact(4)

To the drawing board: call stack

- On the board, we will trace:
  - A small example like hanoi(3, 'A', 'C', 'B') or fact(4)
  - How each call creates a new frame with its own local variables

- On the board, we will trace:
  - A small example like hanoi(3, 'A', 'C', 'B') or fact(4)
  - How each call creates a new frame with its own local variables
- Goal: see how the stack grows and then unwinds

To the drawing board: call stack

- On the board, we will trace:
  - A small example like hanoi(3, 'A', 'C', 'B') or fact(4)
  - How each call creates a new frame with its own local variables
- Goal: see how the stack grows and then unwinds
- Keep this mental model for all future recursive functions

Anatomy of a recursive function

- Base case(s):

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly

Anatomy of a recursive function

- Base case(s):
    - Smallest inputs where we can answer directly
    - No recursive calls here

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:
  - Express the solution in terms of smaller inputs

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:
  - Express the solution in terms of smaller inputs
  - Make sure each step moves closer to a base case

Anatomy of a recursive function

- Base case(s):
    - Smallest inputs where we can answer directly
    - No recursive calls here
- Recursive case:
    - Express the solution in terms of smaller inputs
    - Make sure each step moves closer to a base case
- Progress guarantee:

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:
  - Express the solution in terms of smaller inputs
  - Make sure each step moves closer to a base case
- Progress guarantee:
  - On each call, something gets simpler: $n - 1$, smaller slice, smaller range, …

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:
  - Express the solution in terms of smaller inputs
  - Make sure each step moves closer to a base case
- Progress guarantee:
  - On each call, something gets simpler: $n - 1$, smaller slice, smaller range, ...
- If you forget the base case or progress:

Anatomy of a recursive function

- Base case(s):
  - Smallest inputs where we can answer directly
  - No recursive calls here
- Recursive case:
  - Express the solution in terms of smaller inputs
  - Make sure each step moves closer to a base case
- Progress guarantee:
  - On each call, something gets simpler: $n - 1$, smaller slice, smaller range, ...
- If you forget the base case or progress:
  - The recursion will go on forever and eventually crash

Example: factorial

- Mathematical definition:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1 \end{cases}$$

```python
def fact(n):
    if n == 0:            # base case
        return 1
    return n * fact(n - 1)  # recursive case
```

Example: factorial

- Mathematical definition:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1 \end{cases}$$

- Directly turns into a recursive function:

```
def fact(n):
    if n == 0:             # base case
        return 1
    return n * fact(n - 1)  # recursive case
```

Example: factorial

- Mathematical definition:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n \geq 1 \end{cases}$$

- Directly turns into a recursive function:

```
def fact(n):
    if n == 0:              # base case
        return 1
    return n * fact(n - 1)  # recursive case
```

- Each call reduces n by 1 until we hit the base case $n = 0$

Example: sum from 1 to n

- Let $S(n)$ be the sum of integers from 1 to n:

$$S(n) = 1 + 2 + \cdots + n$$

```python
def sum_to_n(n):
    if n == 1:              # base case
        return 1
    return sum_to_n(n - 1) + n
```

Example: sum from 1 to n

- Let $S(n)$ be the sum of integers from 1 to n:

$$S(n) = 1 + 2 + \cdots + n$$

- Recurrence:

$$S(1) = 1, \qquad S(n) = S(n-1) + n$$

```python
def sum_to_n(n):
    if n == 1:              # base case
        return 1
    return sum_to_n(n - 1) + n
```

Example: sum from 1 to n

- Let $S(n)$ be the sum of integers from 1 to n:

$$S(n) = 1 + 2 + \cdots + n$$

- Recurrence:

$$S(1) = 1, \qquad S(n) = S(n-1) + n$$

```python
def sum_to_n(n):
    if n == 1:                # base case
        return 1
    return sum_to_n(n - 1) + n
```

- Again: smaller input $n - 1$ + one simple local operation ($+n$)

Example: GCD via Euclid's algorithm

- Goal: compute $\gcd(a, b)$ (greatest common divisor)

```
def gcd(a, b):
    if b == 0:          # base case
        return a
    return gcd(b, a % b)
```

Example: GCD via Euclid's algorithm

- Goal: compute $\gcd(a, b)$ (greatest common divisor)
- Key idea: $\gcd(a, b) = \gcd(b, a \bmod b)$

```
def gcd(a, b):
    if b == 0:            # base case
        return a
    return gcd(b, a % b)
```

Example: GCD via Euclid's algorithm

- Goal: compute $\gcd(a, b)$ (greatest common divisor)
- Key idea: $\gcd(a, b) = \gcd(b, a \bmod b)$
- Recurrence:

$$\gcd(a, 0) = a, \qquad \gcd(a, b) = \gcd(b, a \bmod b) \text{ for } b \neq 0$$

```python
def gcd(a, b):
    if b == 0:            # base case
        return a
    return gcd(b, a % b)
```

Example: GCD via Euclid's algorithm

- Goal: compute $\gcd(a, b)$ (greatest common divisor)
- Key idea: $\gcd(a, b) = \gcd(b, a \bmod b)$
- Recurrence:

$$\gcd(a, 0) = a, \qquad \gcd(a, b) = \gcd(b, a \bmod b) \text{ for } b \neq 0$$

```
def gcd(a, b):
    if b == 0:            # base case
        return a
    return gcd(b, a % b)
```

- Each step makes the second argument smaller, until it reaches 0

Example: Fibonacci numbers

- Definition:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

```python
def fib(n):
    if n <= 1:              # base cases: 0 and 1
        return n
    return fib(n - 1) + fib(n - 2)
```

Example: Fibonacci numbers

- Definition:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

- Naive recursive implementation:

```python
def fib(n):
    if n <= 1:            # base cases: 0 and 1
        return n
    return fib(n - 1) + fib(n - 2)
```

Example: Fibonacci numbers

- Definition:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

- Naive recursive implementation:

```
def fib(n):
    if n <= 1:          # base cases: 0 and 1
        return n
    return fib(n - 1) + fib(n - 2)
```

- Looks elegant, but hides a big cost

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
    - fib(n-1) and fib(n-2)

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
  - fib(n-1) and fib(n-2)
- Call tree explodes:

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
  - fib(n-1) and fib(n-2)
- Call tree explodes:
  - Many values are recomputed again and again

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
  - fib(n-1) and fib(n-2)
- Call tree explodes:
  - Many values are recomputed again and again
  - Runtime grows roughly like c^n (exponential)

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
  - fib(n-1) and fib(n-2)
- Call tree explodes:
  - Many values are recomputed again and again
  - Runtime grows roughly like c^n (exponential)
- This is a classic example of recursion done wrong from a performance point of view

Why naive Fibonacci is a disaster

- Each call to fib(n) makes two recursive calls:
  - fib(n-1) and fib(n-2)
- Call tree explodes:
  - Many values are recomputed again and again
  - Runtime grows roughly like c^n (exponential)
- This is a classic example of recursion done wrong from a performance point of view
- We need a way to remember results we have already computed

Fixing Fibonacci with memoization

- Memoization idea:

```python
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)

    return memo[n]
```

Fixing Fibonacci with memoization

- Memoization idea:
  - Store results of expensive calls

```python
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)

    return memo[n]
```

Fixing Fibonacci with memoization

- Memoization idea:
  - Store results of expensive calls
  - Reuse them instead of recomputing

```
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)

    return memo[n]
```

Fixing Fibonacci with memoization

- Memoization idea:
  - Store results of expensive calls
  - Reuse them instead of recomputing

```python
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)

    return memo[n]
```

Fixing Fibonacci with memoization

- Memoization idea:
  - Store results of expensive calls
  - Reuse them instead of recomputing

```
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)

    return memo[n]
```

Recap

- Recursion: solve big problems by trusting solutions to smaller copies

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
    - A clean recursive algorithm

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
  - A clean recursive algorithm
  - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
  - A clean recursive algorithm
  - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$
- Many classic algorithms are recursive: factorial, sum, GCD, Fibonacci, divide-and-conquer

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
    - A clean recursive algorithm
    - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$
- Many classic algorithms are recursive: factorial, sum, GCD, Fibonacci, divide-and-conquer
- Always design:

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
  - A clean recursive algorithm
  - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$
- Many classic algorithms are recursive: factorial, sum, GCD, Fibonacci, divide-and-conquer
- Always design:
  - Clear base cases

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
  - A clean recursive algorithm
  - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$
- Many classic algorithms are recursive: factorial, sum, GCD, Fibonacci, divide-and-conquer
- Always design:
  - Clear base cases
  - Recursive steps that shrink the problem

Recap

- Recursion: solve big problems by trusting solutions to smaller copies
- Towers of Hanoi gave us both:
  - A clean recursive algorithm
  - A recurrence $T(n) = 2T(n-1) + 1 \Rightarrow T(n) = 2^n - 1$
- Many classic algorithms are recursive: factorial, sum, GCD, Fibonacci, divide-and-conquer
- Always design:
  - Clear base cases
  - Recursive steps that shrink the problem
- Next time: we will keep using recurrences to reason about algorithm complexity