Recursion

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 12A's technical content. See python-recursion[2] for the full reference on recursion.

What is recursion?

A function that calls itself is a **recursive function**. This can feel circular — we're using the function to define the function — but it's logically sound, and is essentially an implementation/realization of mathematical **induction**.

> The lecture note explicitly says: you will not be tested on (nor need to really understand) induction — it's just motivation, and there are only one or two questions (of forty) on recursion on the exam. Allocate study time accordingly.

Recursion as induction

The Principle of Mathematical Induction (PMI): for any predicate $P : \mathbb{N} \to \{\text{True}, \text{False}\}$,

$$(P(0) \text{ and } (P(n) \implies P(\text{succ}(n)))) \implies \forall m \in \mathbb{N}, P(m)$$

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-recursion.pdf

In prose: if the proposition is true for zero, and if it being true for n implies it's true for the successor of n, then it's true in general.

Worked example: any $2^n \times 2^n$ board can be tiled with "corner-tiles" (L-shaped trominoes) so that exactly one square is left uncovered.

- **Base case** ($n = 0$): a 1×1 board can be "covered" with zero tiles (the one square is the uncovered one).
- **Induction hypothesis**: assume a $2^n \times 2^n$ board can be tiled this way.
- **Induction step**: a $2^{n+1} \times 2^{n+1}$ board splits into four $2^n \times 2^n$ quadrants. Tile three of the quadrants fully (using one corner-tile in the centre to cover the three inner corners), and tile the fourth quadrant using the induction hypothesis — its single uncovered square becomes the uncovered square for the whole board.

Recursion ingredients

Every recursion must have at least one of each:

- **Base case**: a case defined outright (`if base_case: return constant`).
- **Recursive step**: a line where the function calls itself on "smaller" input.

Factorial: the canonical example

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
def fact(n: int) -> int:
    if not n:           # Pythonic zero check
        return 1        # base case
    return n * fact(n-1)   # inductive case / recursion
```

```
>>> fact(0)
1
>>> fact(3)
6
>>> fact(-1)
RecursionError: maximum recursion depth exceeded
```

`fact(-1)` never reaches the base case — `n` just keeps getting more negative — so it "bottoms out" once Python's recursion limit is hit.

Winding and unwinding

Evaluating `fact(4)`:

```
factorial(4)
← 4 × factorial(3)
← 4 × (3 × factorial(2))
← 4 × (3 × (2 × factorial(1)))
← 4 × (3 × (2 × (1 × factorial(0))))
```

This build-up is the **winding phase** — each recursive call pushes a new activation record onto the call stack (last in, first out). Once the base case is hit, **unwinding** begins, resolving each pending multiplication from the inside out: `1×1 → 2×1 → 3×2 → 4×6 → 24`.

Stack overflow

Python only allows finitely many recursive calls (default limit: 1000). Exceeding it doesn't necessarily mean you're bottoming out (as with `fact(-1)`) — sometimes an algorithm legitimately needs a deeper stack than Python's default:

```
>>> import sys
>>> sys.setrecursionlimit(1500)   # default is 1000
```

Finding base cases

To handle non-obvious base cases, consider an example that makes a **single** recursive call, and ask what the base case must be for that example to work. For instance, $2^k = 2 \cdot 2^{k-1}$ — what does "2 multiplied 0 times" mean? Since $2^1 = 2 \cdot 2^0$ and $2^1 = 2$, it must be that $2^0 = 1$.

Recursive functions over lists/strings usually use the empty list/string as their base case:

Type	Base case
`list`	`[]`
`str`	`''` (empty string)

Advice: ask yourself how you'd solve the problem if you could already solve it on smaller examples. When in doubt, pick base cases so that the *singleton* case (the second-smallest case) works correctly.

Worked exercises

Palindrome check

The obvious, non-recursive answer:

```
def is_palindrome(cs: str) -> bool:
    return cs == cs[::-1]
```

Avoiding a full reversal:

```
def is_palindrome(cs: str) -> bool:
    for k in range(len(cs)//2):
        if cs[k] != cs[-k-1]:
            return False
    return True
```

With recursion:

```
def is_palindrome(cs: str) -> bool:
    if not cs:                                      # base case
        return True
    return cs[0] == cs[-1] and is_palindrome(cs[1:-1])   # recursion
```

Sentence palindrome (ignoring spacing, casing, punctuation)

```
def sentence_palindrome(cs: str) -> bool:
    if not cs:
        return True
    if not cs[0].isalpha():
        return sentence_palindrome(cs[1:])
    if not cs[-1].isalpha():
        return sentence_palindrome(cs[:-1])
    return cs[0].lower() == cs[-1].lower() and sentence_palindrome(cs[1:-1])
```

```
>>> sentence_palindrome("Al lets Della call Ed 'Stella'.")
True
>>> sentence_palindrome("Yo, banana boy!")
True
```

Flatten a list

```
def flatten(xs: list) -> list:
    if not xs:
        return xs                                    # base case
    if type(xs[0]) is int:
        return xs[0:1] + flatten(xs[1:])             # recursion
    return flatten(xs[0]) + flatten(xs[1:])          # recursion

>>> flatten([1, [2], [[3,4]], [[5], [6,7,8], 9]])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Monotonically increasing

```
def increasing(xs: list[int]) -> bool:
    if len(xs) <= 1:                                 # base case
        return True
    return xs[0] < xs[1] and increasing(xs[1:])      # recursion
```

Sum of digits

```
def sum_of_digits(n: int) -> int:
    if not n:
        return 0
    return (n % 10) + sum_of_digits(n // 10)
```

Sublist sum (tree recursion)

Determine if any sublist of `xs` sums to a target value:

```
def sublist_sum(xs: list[int], target: int) -> bool:
    if not xs:
        return not target        # true only for target == 0
    return sublist_sum(xs[1:], target-xs[0]) \
        or sublist_sum(xs[1:], target)

>>> xs = [3, 5, 6, -3, 1, 2]
>>> sublist_sum(xs, 9)
True     # [3, 5, 1]
>>> sublist_sum(xs, -1)
True     # [-3, 2]
```

Every call to `sublist_sum` here triggers *two* further recursive calls (include `xs[0]` in the target sum, or don't) — this branches into a **tree** of recursive calls, rather than the single linear chain seen in `fact` or `is_palindrome`.

Counting grid paths

How many distinct paths are there from the origin to `(x, y)`, moving only in the positive x or y direction?

```
def num_paths(x_cord: int, y_cord: int) -> int:
    if (x_cord, y_cord) == (0, 0):
        return 1
    if x_cord < 0 or y_cord < 0:
        return 0
    return num_paths(x_cord-1, y_cord) + num_paths(x_cord, y_cord-1)
```

Towers of Hanoi

Move `num_disks` disks (labelled $1 \ldots n$, biggest to smallest) from `from_peg` to `to_peg` (pegs labelled 0, 1, 2), moving one disk at a time and never placing a bigger disk on a smaller one:

```
def hanoi(num_disks: int, from_peg: int, to_peg: int) -> list[tuple[int, int]]:
    if not num_disks:
        return []

    off_peg = 3 - from_peg - to_peg

    return (hanoi(num_disks-1, from_peg, off_peg)
            + [(num_disks, to_peg)]
            + hanoi(num_disks-1, off_peg, to_peg))
```

```
>>> hanoi(2, 0, 2)
[(1, 1), (2, 2), (1, 2)]
```

The recursive idea: to move n disks from `from_peg` to `to_peg`, first move the top $n-1$ disks out of the way (to the spare `off_peg`), move the single bottom disk directly, then move those $n-1$ disks from the spare peg onto their final destination.

See towers-of-hanoi[3] for a formal induction proof that this always works, plus a fully validated 4-disc move trace.

[3] courses/csse1001/towers-of-hanoi.pdf

Tiling exercise (unsolved)

The lecture also poses tiling a $2^n \times 2^n$ board with distinct numbered tiles (rather than just proving a tiling exists, as in the induction example above), leaving `0` for the uncovered square:

```
>>> tile(2)
[[0, 2, 3, 3],
 [2, 2, 1, 3],
 [4, 1, 1, 5],
 [4, 4, 5, 5]]
```

No solution was given in the lecture materials — left here as an open exercise (a natural recursive approach mirrors the induction proof: solve the smaller board, then place three tiles to cover the other three quadrants).

Longest common subsequence

```
def lcs(xs: str, ys: str) -> str:
    if not xs or not ys:
        return ""
    if xs[0] == ys[0]:
        return xs[0] + lcs(xs[1:], ys[1:])
    return max(lcs(xs[1:], ys), lcs(xs, ys[1:]), key=len)
```

```
>>> lcs("cbfbcdeb", "cbebcee")
'cbbce'
```

Dynamic programming

Branching recursion (like `sublist_sum`) can be slow because the same subproblems get recomputed many times. **Dynamic programming** caches previous results to avoid this — most effective when there are many overlapping subproblems, like the naive Fibonacci function:

```
def fib(n: int) -> int:
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

With caching:

```
fib_cache = {0: 1, 1: 1}          # base cases go here
def fib(n: int) -> int:
    global fib_cache
    if n in fib_cache:
        return fib_cache[n]
    fib_cache[n] = fib(n-1) + fib(n-2)
    return fib_cache[n]
```

Summary

- Recursion is a function calling itself, with a base case and a recursive step.
- It's the programming realization of mathematical induction.
- Evaluation has a winding phase (building up calls) and an unwinding phase (resolving them).
- Common base cases: `0` for numbers, `[]`/`''` for lists/strings.
- Some recursions branch into a tree of calls rather than a single chain — caching (dynamic programming) can make these tractable.

Next: functional programming and complexity (Week 13).