Recursion

Table of contents

What is recursion? 1
Recursion ingredients 2
Factorial: the canonical example 2
Finding base cases 6
Worked exercises 7
Dynamic programming 13
Summary 13

See csse1001[1] for course logistics. This is Lecture 12A. 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**.

> Induction is not tested, and does not need to be really understood — it's just motivation. 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 $P : \mathbb{N} \to \{\texttt{True}, \texttt{False}\}$,

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

[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. Proposition: *any* $2^n \times 2^n$ board with $n \in \mathbb{N}$ can be tiled with "corner-tiles" (L-shaped trominoes) so that only one square is left uncovered, and that square can be anywhere. A standard (8×8) chess board is one such board. The proof of the general claim:

- **Base case** $(n = 0)$: a $2^0 \times 2^0 = 1 \times 1$ board is covered with *zero* tiles — the single square is the uncovered one.
- **Induction hypothesis**: assume a $2^n \times 2^n$ board can be tiled this way.
- **Induction step**: cut a $2^{n+1} \times 2^{n+1}$ board into four $2^n \times 2^n$ quadrants. Every quadrant is tiled by the hypothesis. Put the quadrant carrying the board's intended uncovered square anywhere you like; orient the other three so that *their* uncovered squares are the three squares that meet at the centre of the board. Those three squares form a corner-tile shape, so one extra tile covers all three — and the board is left with exactly one uncovered square.

The shape of that argument — *solve four smaller copies of the same problem, then do one unit of work to glue them together* — is exactly the shape of a recursive function. See csse1001-diagram-notation[3] for the drawing conventions the figure follows.

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

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

For instance $3! = 3 \times 2 \times 1 = 6$.

A set of n distinct elements can be arranged in $n!$ different ways.

[3] courses/csse1001/csse1001-diagram-notation.pdf

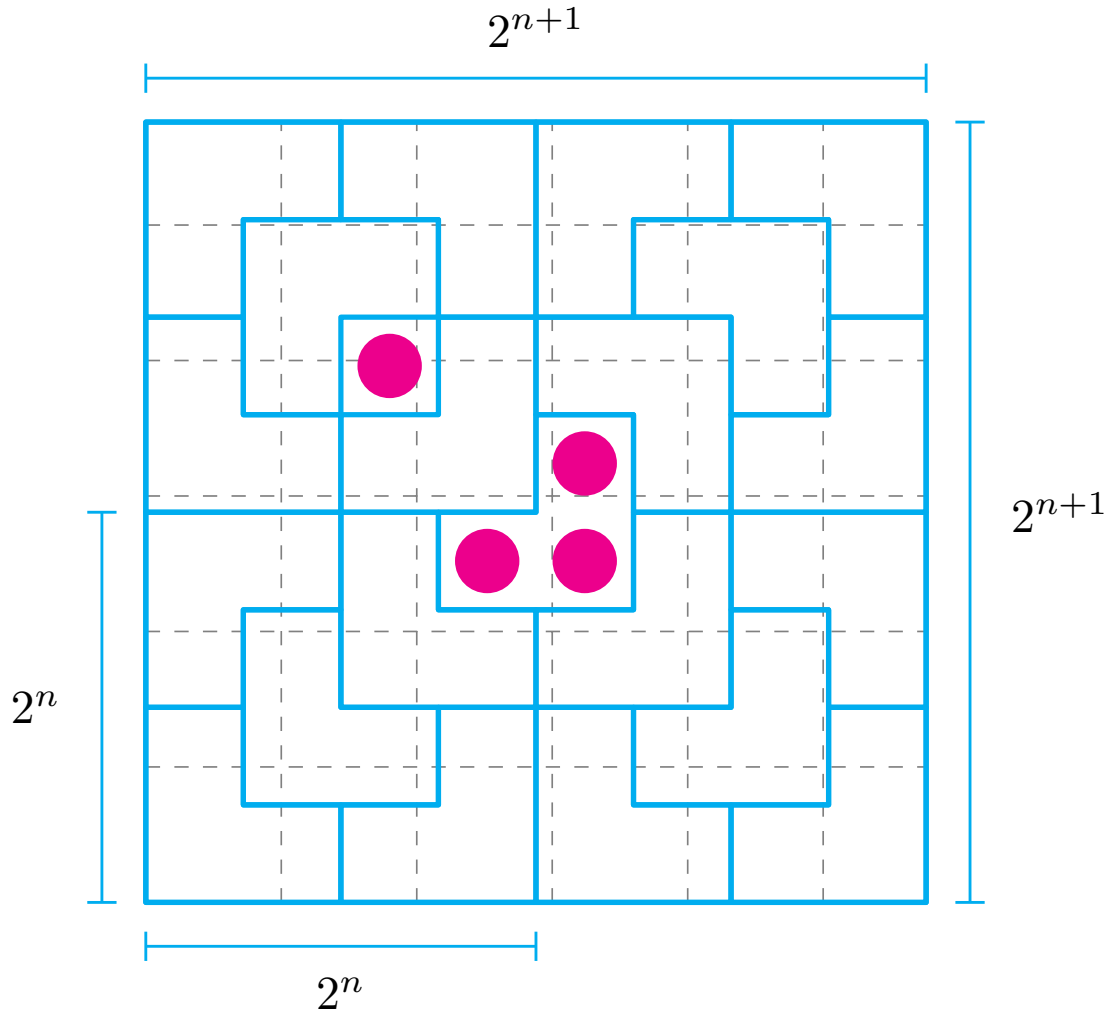

Figure 1: The induction step of the tiling proof. Four $2^n \times 2^n$ quadrants are each tiled by the induction hypothesis; three of them are oriented so their uncovered squares (magenta discs) meet at the centre, where one extra corner-tile covers all three, leaving the fourth quadrant's disc as the whole board's only uncovered square. Tile outlines are cyan, the reference grid is grey and dashed, a magenta disc marks an uncovered square, and the dimension lines are cyan.

```
>>> 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. This is called **bottoming out**.

Winding and unwinding

During the **winding phase** each new recursive call pushes an *activation record* onto a **stack**, and those records complete **last in, first out**. The picture above is not a stack: a stack would be drawn in the reverse order — the deepest call (`factorial(0)`) is the top of the stack, not the bottom of the page.

Once the base case returns, **unwinding** resolves each pending multiplication from the inside out: $1 \times 1 \to 2 \times 1 \to 3 \times 2 \to 4 \times 6 \to 24$.

Stack overflow

Python only allows *finitely* many recursive calls. When you exceed that limit the recursion has **overflowed**.

Exceeding the limit does **not necessarily** mean you are bottoming out — an algorithm may legitimately need more stack than Python provides by default:

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

Both cases side by side:

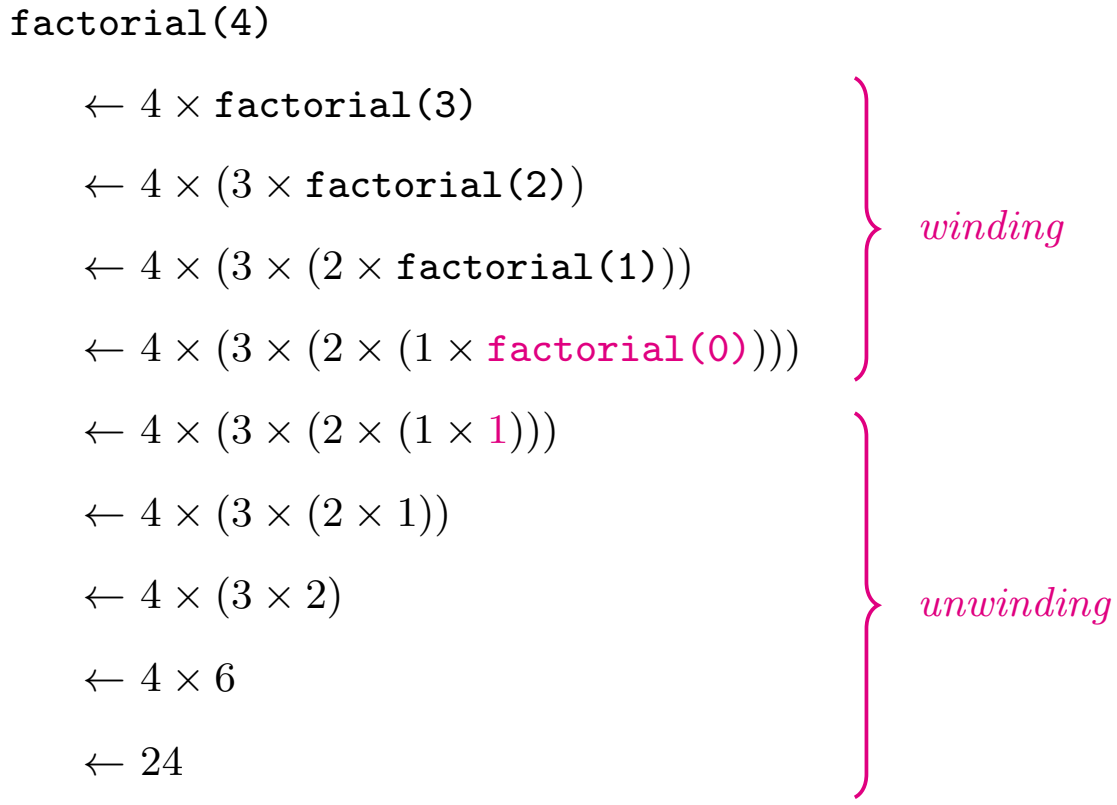

Figure 2: How factorial(4) evaluates. The winding phase builds up the nested product until the base case factorial(0) is reached (magenta), then unwinding collapses it from the inside out to 24. The magenta braces with italic labels mark the winding and unwinding phases.

```
>>> fact(998)
402790050127220994538240674597601587306681545756471103647447357787726238...
>>> fact(999)
RecursionError: maximum recursion depth exceeded

>>> import sys
>>> sys.setrecursionlimit(1001)

>>> fact(999)
402387260077093773543702433923003985719374984210714632543799910429938512...

>>> fact(-1)
RecursionError: maximum recursion depth exceeded      # bottoming out
```

So `fact(999)` was only limited by the default ceiling and works once it is raised, while `fact(-1)` fails at *any* limit — that one is a genuine missing-base-case bug.

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* in order for that example to work. For instance,

$$2^k = 2 \cdot 2^{k-1} = \underbrace{2 \times \cdots \times 2}_{k \text{ many times}}$$

What does "0 many times" mean? We have $2^1 = 2$ and $2^1 = 2 \cdot 2^0$, and therefore $2^0 = 1$.

Recursive functions defined over *lists* and *strings* usually have as their base case the empty list and the empty string:

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

Advice:

- Ask yourself how you would solve the problem if you could *solve the same problem on smaller examples.*
- When in doubt, set the base cases to what they *need to be* in order for the **singleton case** to work. The singleton case is the second-smallest one.

Worked exercises

Palindrome check

Determine if a string is a palindrome — a word that is equal to its reverse.

The obvious answer:

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

```
>>> is_palindrome("racecar")
True
>>> is_palindrome("deified")
True
>>> is_palindrome("squid")
False
```

Avoiding the reversal, iteratively:

```
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)

The exercise is to fill in the body so that these hold:

```
>>> def sentence_palindrome(cs: str) -> bool:
...     pass

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

One implementation follows the "strip from whichever end is not a letter, then compare" shape of the plain palindrome above:

```
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])
```

Flatten a list

Flatten a list — that is, remove all internal brackets.

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

def flatten(xs: list) -> list[int]:
    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
```

There are two different recursive cases: a leading `int` is peeled off as a one-element slice (so that `+` still concatenates lists), while a leading *list* is flattened in its own right before being joined to the flattened rest.

Monotonically increasing

Check if a list of numbers is 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
```

This is a case where the base case is chosen so the *singleton* works: a one-element list has nothing to compare, so it must be `True`, and `len(xs) <= 1` covers the empty list too.

Sum of digits

Sum the digits of a number — the sum of the digits of 123 is $1 + 2 + 3 = 6$.

```
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 ***`xs: list[int]`*** *has a sublist that sums to some target.*

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

```
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)
```

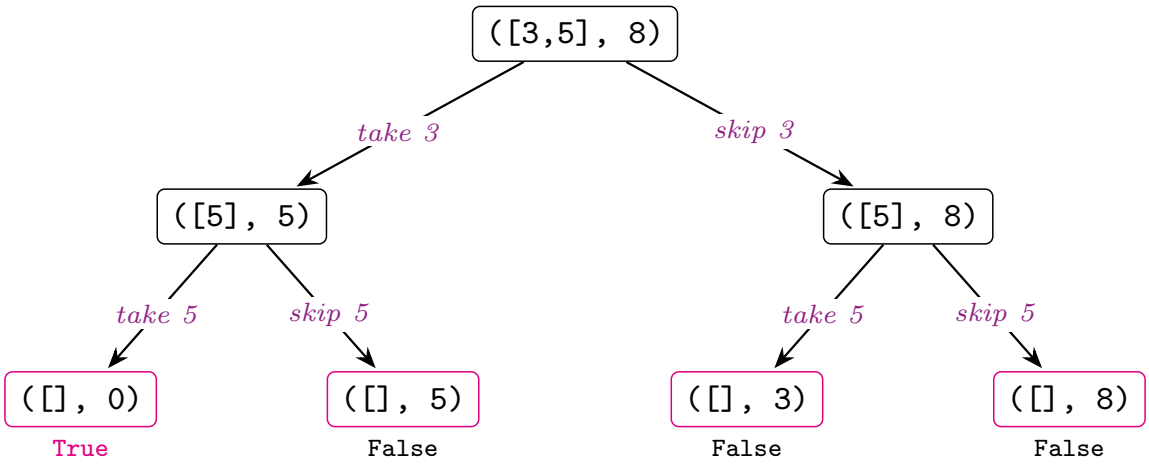

Figure 3: The tree of calls generated by sublist_sum([3, 5], 8). Each node is one invocation, labelled by its arguments; the left edge takes xs[0] into the running total, the right edge skips it. Magenta-outlined nodes are base-case calls, which return not target — only the leftmost, with target driven to 0, returns True, and the chain of ors carries that back to the root. The list has two elements to keep the tree small; the six-element list above would need 64 leaves.

This creates a **tree** of recursive calls, as every function invocation triggers *two* recursive calls — either `xs[0]` is part of the sublist (so the rest must make up `target - xs[0]`) or it isn't (so the rest must still make up `target`).

Every earlier example (`fact`, `is_palindrome`, `flatten`'s int branch, `increasing`, `sum_of_digits`) makes at most one call per invocation and so unwinds along a single chain; this one doubles at every level, so a list of n elements costs 2^n leaf calls.

Counting grid paths

Standing on a grid at the origin, you can only move along the x and y axes by positive increments. How many distinct paths are there to an arbitrary position on the grid?

```
def num_paths(x_cord: int, y_cord: int) -> int:
    """
    How many paths from (0, 0) to x_cord, y_cord when you can
    only move in positive direction.
    """
    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)
```

Two base cases here: the origin is reached exactly one way, and stepping off the grid contributes nothing. The recursive step is the same "branch on the last choice" idea as `sublist_sum` — the last move was either horizontal or vertical.

Towers of Hanoi

Move `num_disks` disks (labelled `1`, `2`, …, `num_disks`, with `1` the smallest) from `from_peg` to `to_peg`, pegs numbered 0, 1 and 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]]:
    """Move <num_disks> many disks labelled 1, 2, ..., num_disks
    from peg <from_peg> to <to_peg>. Pegs are numbered 0, 1, and 2.
    """
    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. The arithmetic trick `off_peg = 3 - from_peg - to_peg` picks the remaining peg, because $0+1+2=3$.

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

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

Tiling exercise (unsolved)

The exercise is to tile 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:

```
>>> def tile(n: int) -> list[list[int]]
...     pass
>>> tile(2)
[[0, 2, 3, 3],
 [2, 2, 1, 3],
 [4, 1, 1, 5],
 [4, 4, 5, 5]]
```

This is left as an open exercise. A natural recursive approach mirrors the induction proof in Figure 1: tile the four quadrants recursively, then number one more tile for the three squares that meet at the centre.

Longest common subsequence

The string `"abc"` has eight substrings: `""`, `"a"`, `"b"`, `"c"`, `"ab"`, `"ac"`, `"bc"`, `"abc"`. Write a function that, given two strings, returns the longest common subsequence of those two strings.

```
def lcs(xs: str, ys: str) -> str:
    """
    >>> lcs("cbfbcdeb", "cbebcee")
    'cbbce'
    """
    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)
```

Another branching recursion: when the leading characters differ, both "drop a character from `xs`" and "drop a character from `ys`" have to be explored and the longer result kept.

Dynamic programming

One problem with recursion is its efficiency with **branching** recursions. Caching previous results is a strategy that is quite powerful when the branching recursion has many **overlapping subproblems** — the classic case being naive Fibonacci:

```
>>> def fib(n: int) -> int:
...     """ Return the nth Fibonacci number."""
...     if n < 2:
...         return 1
...     return fib(n-1) + fib(n-2)
```

With caching it is *much* faster:

```
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]
```

The base cases live in the cache dictionary rather than in an `if`, and the cache lookup at the top is what turns the tree of calls back into something linear.

Summary

- Recursion is a function calling itself, with a base case and a recursive step.
- It's the programming realization of mathematical induction — the tiling proof and the recursive `tile`/`hanoi` functions have the same structure.
- Evaluation has a winding phase (building up a stack of activation records) and an unwinding phase (resolving them last in, first out).
- Common base cases: `0` for numbers, `[]`/`''` for lists/strings; pick the base case so that the singleton case works.
- Overflowing the stack is not the same bug as bottoming out — raising `sys.setrecursionlimit` fixes the first and never fixes the second.
- Some recursions branch into a tree of calls rather than a single chain — caching (dynamic programming) can make those tractable.

Next: 2025-10-28-complexity-and-sorting[5] — functional programming and complexity (Week 13).

[5] courses/csse1001/2025-10-28-complexity-and-sorting.pdf