CSSE1001 — Week 12 Notes

Table of contents

Recursion **2**
- What is recursion? . . . 2
- Recursion ingredients . . . 4
- Factorial: the canonical example . . . 4
- Finding base cases . . . 6
- Worked exercises . . . 7
- Dynamic programming . . . 13
- Summary . . . 14

Towers of Hanoi (Induction & Recursion, Supplementary) **14**
- The puzzle . . . 15
- The proposition . . . 15
- Base case . . . 15
- The inductive step . . . 15
- Validating the algorithm: four disks . . . 17
- Exercise stub . . . 19

Reference material **20**
- Algorithmic Complexity (Big-Oh) . . . 20
- CSSE1001 Diagram Notation . . . 26
- Python Recursion . . . 43
- Sorting Algorithms . . . 46

Recursion

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 $P : \mathbb{N} \to \{\texttt{True}, \texttt{False}\}$,

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

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.

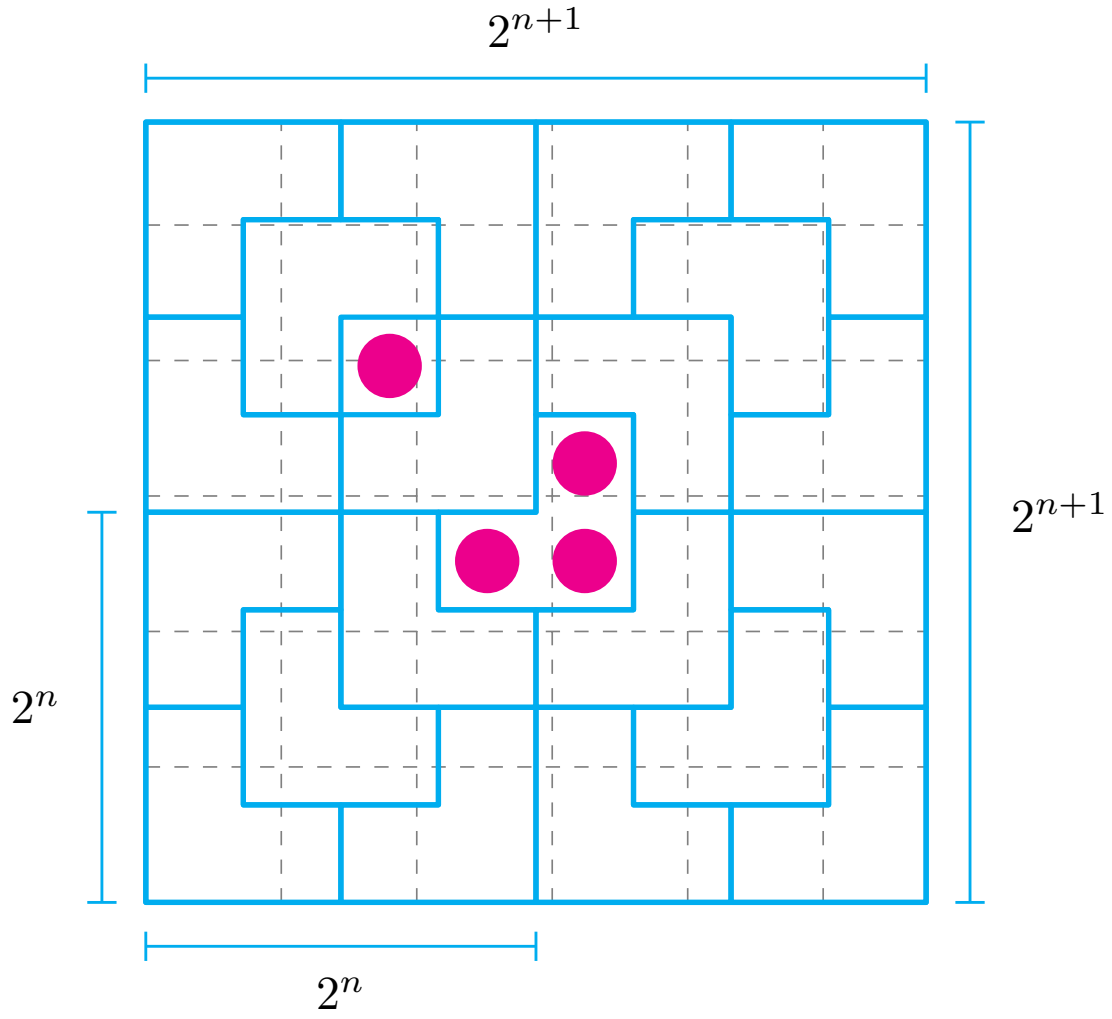


Figure 1: The induction step of the tiling proof. Four $2^n \times 2^n$ quadrants are each tiled by the induction hypothesis; three of them are oriented so their uncovered squares (magenta discs) meet at the centre, where one extra corner-tile covers all three, leaving the fourth quadrant's disc as the 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.

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.

```
>>> 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**.

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

Winding and unwinding

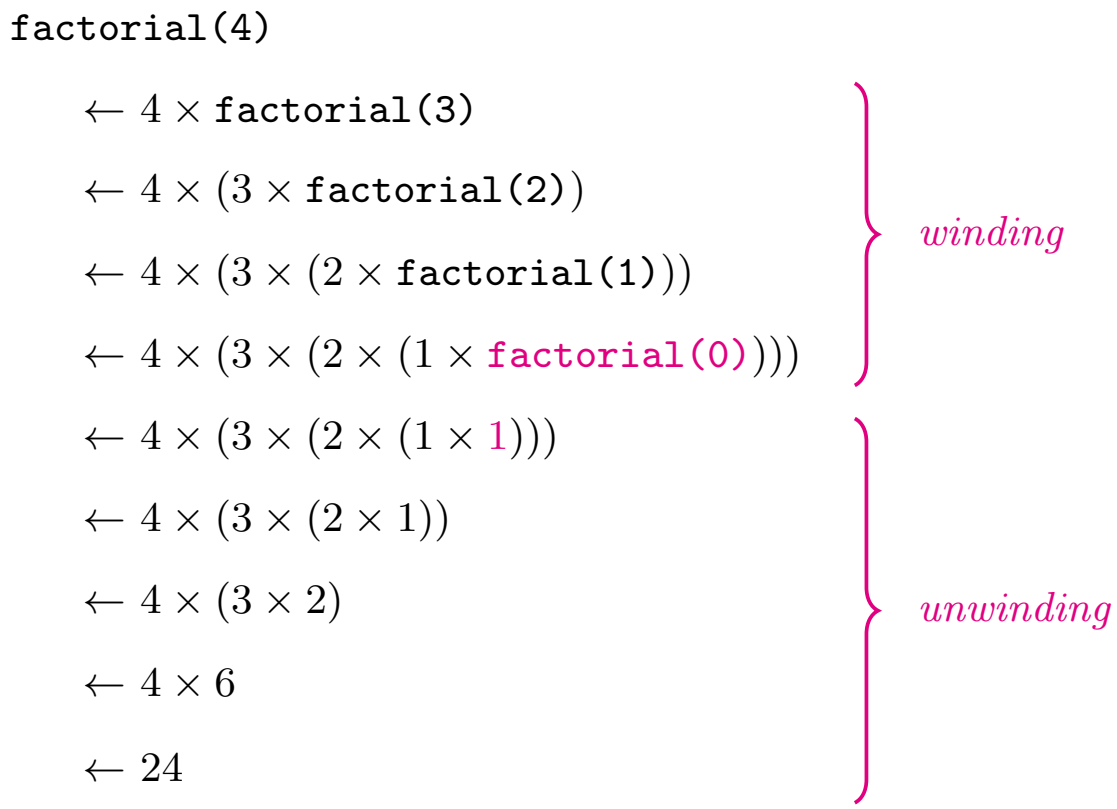


Figure 2: How factorial(4) evaluates. The winding phase builds up the nested product until the base case factorial(0) is reached (magenta), then unwinding collapses it from the inside out to 24. The magenta braces with italic labels mark the winding and unwinding phases.

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:

```
>>> 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)
```

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`).

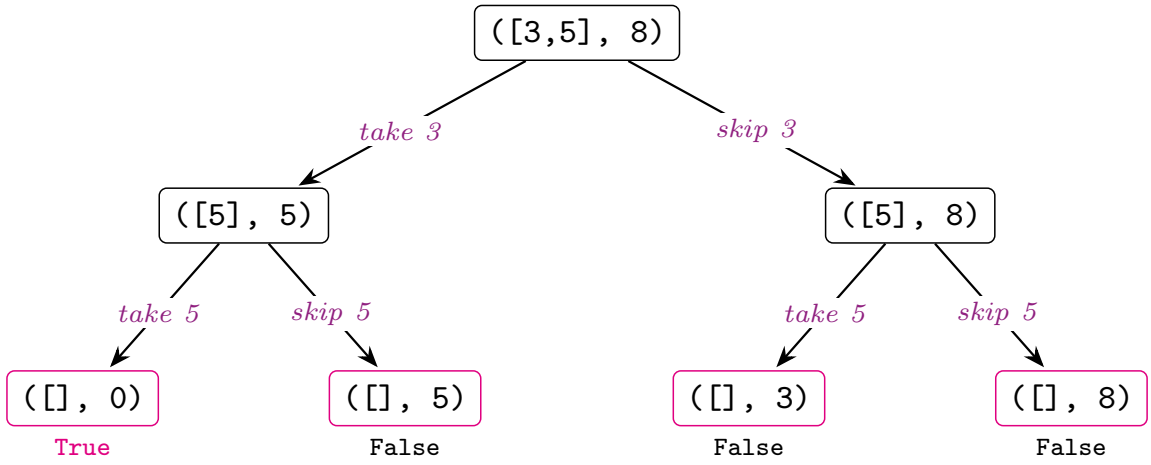


Figure 3: The tree of calls generated by sublist_sum([3, 5], 8). Each node is one invocation, labelled by its arguments; the left edge takes xs[0] into the running total, the right edge skips it. Magenta-outlined nodes are base-case calls, which return not target — only the leftmost, with target driven 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.

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.

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.

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

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).

Towers of Hanoi (Induction & Recursion, Supplementary)

Supplementary material reinforcing 2025-10-20-recursion[6] and python-recursion[7] — a formal induction proof that the Towers of Hanoi puzzle is always solvable, plus a fully validated worked trace.

The boards in the figures use colours that are not in csse1001-diagram-notation[8]: poles and base are tan bars, disks are pink bars (a different pink from the course's usual magenta), and disk width increases with disk number.

[5] courses/csse1001/2025-10-28-complexity-and-sorting.pdf
[6] courses/csse1001/2025-10-20-recursion.pdf
[7] courses/csse1001/python-recursion.pdf
[8] courses/csse1001/csse1001-diagram-notation.pdf

The puzzle

> Move one disk at a time. Bigger disks cannot be stacked on smaller disks. Prove the stack can be moved to the last pole.

The proof says **pole** and the code says **peg**. The two words mean the same thing here.

The proposition

The proposition is a biconditional with the two rules attached as a numbered list:

> $P(n) \iff$ a stack of n-disks can be moved to an *arbitrary* pole while…
>
> 1. Moving one disk at a time.
> 2. Bigger disks never stacked on smaller disks.

The word *arbitrary* is doing real work: the claim is not "the stack can be moved to one particular pole", but "to whichever pole you name". That is what makes the hypothesis strong enough to be used **twice** in the step below — once to reach the spare pole, once to reach the target.

Base case

> $P(0) \iff$ *zero* disks can be moved to an *arbitrary* pole while …

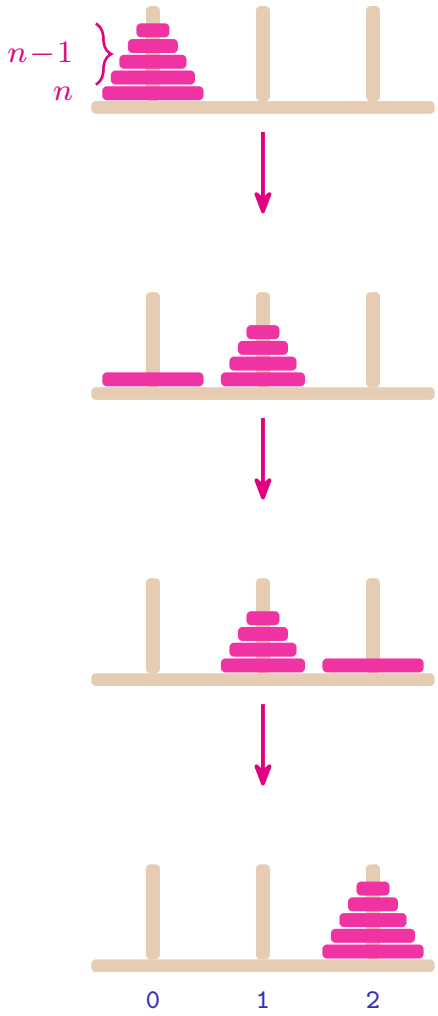
With zero disks the board is three empty poles. This is a **vacuous** case: with nothing to move, there is no move that could break either rule, so the conditions hold trivially.

The inductive step

Assume $P(n-1)$ — that is, suppose we can move $n-1$ disks to any pole we like, obeying both rules. A stack of n disks then crosses the board in four states, and therefore $P(n)$. By the PMI, $\forall n; P(n)$.

Written out:

- **Base case** $P(0)$: vacuously true — there is nothing to move.
- **Inductive step**: assume $P(n-1)$. Then, for a stack of n disks:
  1. Move the top $n-1$ disks onto the spare pole (possible by the inductive hypothesis).
  2. Move the remaining, largest disk onto the target pole directly. This one move is legal precisely because step 1 uncovered it and emptied the target.



Start. All n disks on peg 0 — the state $P(n)$ asks us to move.

`hanoi(n-1, from_peg, off_peg)`

Move the top $n-1$ disks to the spare pole. Legal by the hypothesis $P(n-1)$.

Disk n is uncovered, and the target pole is empty.

`[(n, to_peg)]`

One legal move: nothing sits on disk n, and nothing is on peg 2 for it to land on.

Disk n is home, and never moves again.

`hanoi(n-1, off_peg, to_peg)`

$P(n-1)$ a second time. Every disk in that stack is smaller than disk n, so landing on it breaks no rule.

Therefore $P(n)$. By the PMI, $\forall n\,;\, P(n)$.

Figure 4: The inductive step as four board states (tan poles and base, pink disks, widths increasing with disk number). Five disks stand in for a general n. The pegs are numbered 0/1/2. The magenta brace names the top n-1 disks and the bottom disk n. The three monospace code fragments are the three terms of the return statement of hanoi, with num_disks shortened to n. Nothing about the puzzle forces the spare pole to be the middle one — that is simply the arrangement drawn here, and it matches hanoi(4, 0, 2).

3. Move the $n-1$ disks from the spare pole onto the target pole, on top of the largest disk (again possible by the inductive hypothesis, and legal because every disk in that stack is smaller than the one it lands on).

This gives a valid sequence of moves for n disks, so $P(n)$ holds.

- **Conclusion**: by the Principle of Mathematical Induction, $P(n)$ holds for all n.

This is exactly the recursive structure of the `hanoi` function in 2025-10-20-recursion[9]: **the inductive hypothesis *is* the recursive call**, and the three steps above are the three terms of that function's single `return` statement, in order. The proof establishes only that a solution *exists*; it says nothing about the solution being the shortest one.

Validating the algorithm: four disks

The worked example is the 15-move solution for four disks, replayed one board at a time. The move list in full:

```
(1,1),(2,2),(1,2),(3,1),(1,0),(2,1),(1,1),(4,2),(1,2),(2,0),(1,0),
(3,2),(1,1),(2,2),(1,2)
```

A move is written `(disk, destination_peg)`. Disks are labelled 1 through n and pegs are labelled 0, 1, and 2. The **order inside the pair** is that the first entry is the disk and the second is where it lands, and disk `1` is the *smallest*. Both are fixed by the code, which appends exactly `[(num_disks, to_peg)]` for each move, and confirmed by reading the boards.

Grouping the same 15 moves by the structure of Figure 4 — 7 moves to clear disks 1–3 onto the spare peg, 1 move for disk 4, then 7 moves to bring disks 1–3 back on top:

```
(1,1) (2,2) (1,2) (3,1) (1,0) (2,1) (1,1)
(4,2)
(1,2) (2,0) (1,0) (3,2) (1,1) (2,2) (1,2)
```

That grouping is exactly what `hanoi(4, 0, 2)` from 2025-10-20-recursion[10] returns, as three concatenated pieces:

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

[9] courses/csse1001/2025-10-20-recursion.pdf
[10] courses/csse1001/2025-10-20-recursion.pdf

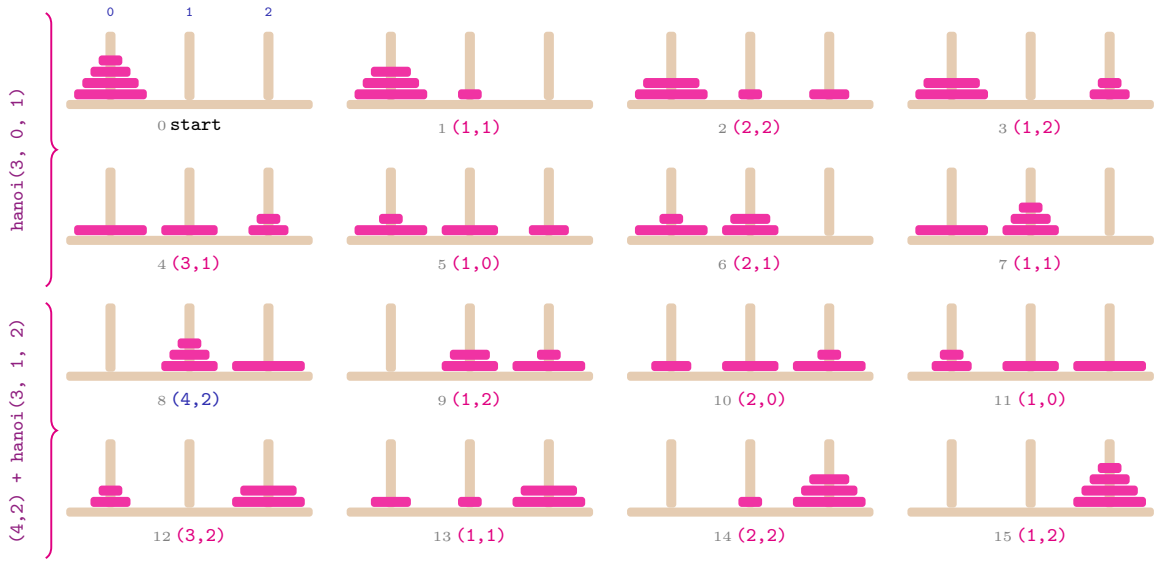


Figure 5: All sixteen board states of the four-disk solution, in reading order. Each panel is captioned with the move that produced it: a small grey move number and the move itself in magenta. The peg numbers are on the first board. Move 8 is in blue — the single move of the largest disk, the pivot of the whole recursion. The two magenta braces name the calls each half of the trace comes from. The split is exact: the eight boards above it are the start state plus the seven moves of hanoi(3, 0, 1), and the eight below are (4,2) followed by the seven moves of hanoi(3, 1, 2).

Two details worth reading off Figure 5. First, **the largest disk moves exactly once** (move 8), and once it is placed nothing above it ever needs to shift it again — the pivot the whole proof turns on. Second, **the spare peg is not fixed**: the outer call uses peg 1 as its spare, but the nested `hanoi(3, 0, 1)` uses peg 2, which is why disks appear on the target peg long before disk 4 arrives there. That is `off_peg = 3 - from_peg - to_peg` doing its work at every level.

How many moves

The count falls straight out of the code's shape: one call to `hanoi(n, ...)` emits whatever `hanoi(n-1, ...)` emits, twice, plus one move of its own, and `hanoi(0, ...)` emits none.

$$T(0) = 0, \qquad T(n) = 2\,T(n-1) + 1 \qquad \Longrightarrow \qquad T(n) = 2^n - 1$$

So four disks take $2^4 - 1 = 15$ moves, which is what the trace above shows, and the cost is exponential in the number of disks — see algorithmic-complexity[11] for that vocabulary.

Exercise stub

The implementation is left as an exercise, starting from only the signature and docstring:

```
def hanoi(n: int, from_peg: int, target_peg: int) -> list[tuple[int, int]]:
    """
    Solve towers of hanoi problem for <n> disks from <from_peg> to
    <target_peg>

    Disks are labelled 1 through n.

    Pegs are labelled 0, 1, and 2.
    """
```

2025-10-20-recursion[12] (Lecture 12A) has a complete implementation, with slightly different naming — `num_disks`/`to_peg` rather than `n`/`target_peg` — and a return annotation of `list[tuple[int]]` where this stub writes the more accurate `list[tuple[int, int]]`, since each move really is a pair. It also has the `off_peg = 3 - from_peg - to_peg` trick that picks the spare peg.

[11] courses/csse1001/algorithmic-complexity.pdf
[12] courses/csse1001/2025-10-20-recursion.pdf

Related: MATH1061's welcome & intro problems[13] previews the same puzzle (and the $2^n - 1$ minimum move count) as motivation for MATH1061's own later induction/recursion content — a nice example of the same idea showing up in two different courses.

Reference material

Algorithmic Complexity (Big-Oh)

An **algorithm** is an ordered sequence of unambiguous instructions that eventually terminates (see history-of-computing[14]). Once we have more than one algorithm for the same problem, we need a way to say which is better — that measure is **complexity**.

This is not a side topic bolted on at the end of semester. The entire subject is defined in one sentence:

> **Computer science** is the practice/art of creating *algorithms* and *measuring their relative effectiveness*.

First met in lecture 1A[15]. sorting-algorithms[16] is the first half of that sentence; complexity is the second half. The course opens on the definition in week 1 and closes on it in week 13.

Why not just time it?

Timing depends on the machine, the language, what else is running, and the particular input. Instead we *count operations*. For a sorting algorithm the operations worth counting are **comparisons** and **swaps**; more generally we count whatever the algorithm does repeatedly.

The machine matters a great deal. The hardware timeline runs from Konrad Zuse's Z3 to a current AMD part, and eight of its machines have a quoted speed. Laid on a single logarithmic axis, those speeds are the argument against the stopwatch.

Nothing about selection sort changed between 1941 and today. The number of comparisons it makes on a nine-element list is the same on the Z3 as on the Threadripper; only the wall-clock time moves, and it moves by a factor of about 10^{13}. **That invariant count is what complexity measures.**

[13] courses/math1061/2026-07-27-welcome-and-intro-problems.pdf
[14] courses/csse1001/history-of-computing.pdf
[15] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf
[16] courses/csse1001/sorting-algorithms.pdf
[17] courses/csse1001/history-of-computing.pdf

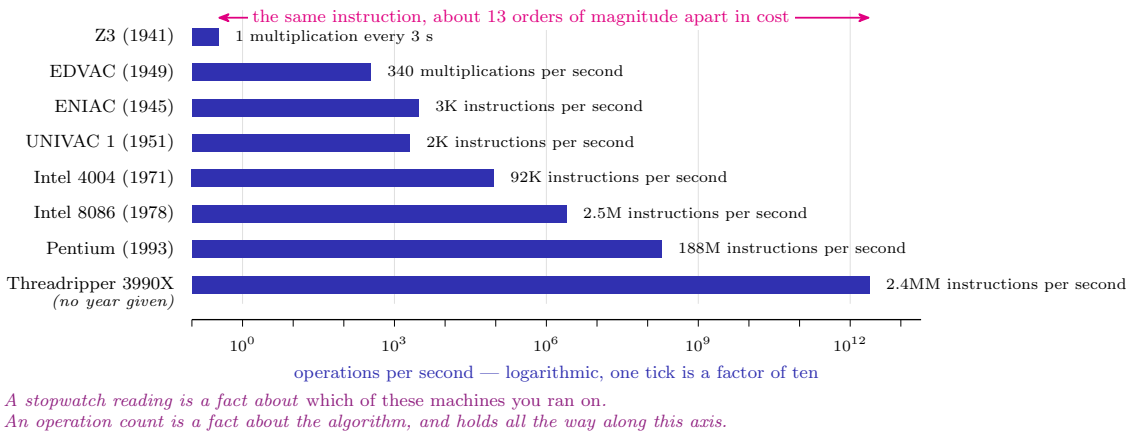


Figure 6: The quoted speeds of the machines in the hardware timeline, on one logarithmic axis. Each bar is one machine. The **units are not comparable** - Z3 and EDVAC are quoted in *multiplications* per second and the rest in *instructions* per second. The rows are not in year order: EDVAC (1949) sits above ENIAC (1945), and the Threadripper has no year. The last figure, `2.4MM`, is placed at 2.4×10^{12}, reading MM as million-million. The magenta span runs from the slowest machine to the fastest. The full table with prices and manufacturers is in history-of-computing[17].

Definition

> We measure the complexity of an algorithm by counting the number of operations performed. But the number of operations is proportional to the length of the input (usually denoted n) — and it is really this *proportion* we are interested in, rather than a precise accounting of operations.

That last sentence is the whole idea: `2n + 2` and `5n + 100` are both "linear", and the difference between them matters far less than the difference between linear and quadratic.

Big-Oh notation

Definition. We say "$f(n)$ is big-oh of $g(n)$", written $f(n) = O(g(n))$, when

$$\exists\, c, k \in \mathbb{R}_{>0} : \forall n \geq k,\ 0 \leq f(n) \leq c\, g(n).$$

That is, when there is some positive constant c such that $c \cdot g$ is *eventually* larger than f. The direction matters: "f is eventually larger than cg" is backwards and contradicts the inequality.

Worked examples:

1. $2x^2 + x + 1 = O(n^2)$, because for $x > 2$ we have $2x^2 + x + 1 < 3x^2$ (take $c = 3$, $k = 2$).
2. $x^3 - \sin(x) = O(n^3)$, because for $x > 1$ we have $x^3 - \sin(x) \leq x^3 + 1 < 2x^3$ (take $c = 2$, $k = 1$).

In practice this means: **drop the constants and keep only the fastest-growing term.**

The four complexity classes in this course

Linear — $O(n)$

```
def linear(xs: list[int]):
    n = len(xs)         # 1  assignment
    index = 0           # 1  assignment
    while index < n:    # n  comparisons
        index += 1      # n  assignments / additions
                        # 2n + 2 = O(n) operations
```

Quadratic — $O(n^2)$

A loop nested inside a loop, each running n times:

```
def foo(xs: list[int]):
    n = len(xs)         # 1
    i = 0               # 1
    while i < n:        # n
        j = 0           # n
        while j < n:    # n * n
            j += 1      # n * n
        i += 1          # n
                        # 2n**2 + 3n + 1 = O(n**2)
```

Constant — $O(1)$

The trap here is that a big number is still a *constant*: the loop below runs ten billion times, but that count does not depend on `n` at all.

```
def foo(xs: list[int]):
    n = len(xs)           # 1
    index = 0             # 1
    while i < 10**10:     # 10**10
        index += 1        # 10**10
                          # 2*10**10 + 2 = O(1)
```

Logarithmic — $O(\log n)$

The loop variable is in an *exponent*, so the loop runs about $\log_2 n$ times:

```
def log(xs: list[int]):
    n = len(xs)                 # 1
    index = 0                   # 1
    while 2 ** i < len(xs):     # log[2](n)
        i += 1                  # log[2](n)
                                # 2log[2](n) + 2 = O(log(n))
```

Both this listing and the constant one above mix their counter names: each initialises `index` and then increments `i`. Neither would run; the operation counts in the comments are still the point.

The base of the logarithm is a constant factor ($\log_2 n = \log n / \log 2$), so it disappears inside the big-oh — $O(\log_2 n)$ and $O(\log n)$ mean the same thing.

How to read a loop's complexity

- A loop that advances by a *fixed step* toward `n` → $O(n)$.
- A loop that *multiplies or divides* its counter each pass → $O(\log n)$.
- Nested loops → multiply their counts ($O(n) \times O(n) = O(n^2)$).
- A bound that doesn't mention `n` → $O(1)$, however large it is.
- A bound that is an input **value** rather than an input **length** → see the next section, and be careful.

When the input is a value, not a length

Every example above takes `n = len(xs)`: the input is a collection and n counts its elements. One of the course's earliest algorithms, integer division from Lecture 1B[18], does not fit that mould.

Dividing two integers and still getting an integer is grade-school division: for $x = 17$ and $y = 3$, *17 ÷ 3 is 5 remainder 2*, i.e.

$$17 = 5 \cdot 3 + 2 \qquad \text{or} \qquad \frac{17}{3} = 5 + \frac{2}{3}.$$

Python hands you both halves directly, and the two together reconstruct the input:

[18] courses/csse1001/2025-07-29-python-as-a-calculator.pdf

```
>>> 17 // 3
5
>>> 17 % 3
2
>>> 3 * (17 // 3) + (17 % 3)
17
```

`17 // 3` is the *quotient* and `17 % 3` is the *remainder*. The theorem the two operators rest on:

> **Division algorithm.** Let x and y be positive integers. There is a **unique** quotient q and **unique** remainder r with $0 \leq r < y$ satisfying $x = q \cdot y + r$.

The remainder can also be computed by hand, subtracting `y` away until nothing more can be taken (`while` is not taught until week 4):

```
>>> def remainder(x: int, y: int) -> int:
...     ans = x
...     while ans >= y:
...         ans = ans - y
...     return ans
>>> remainder(17, 7)
3
```

The recursive version (recursion is week 12, python-recursion[19]):

```
>>> def remainder(x: int, y: int) -> int:
...     return x if x < y else remainder(x-y, y)
>>> remainder(17, 7)
3
```

Now count operations. The loop body runs once per subtraction, and the number of subtractions is exactly $\lfloor x/y \rfloor$ — the quotient. So `remainder(x, y)` costs on the order of x/y operations, and `remainder(10**9, 1)` would run a billion times while `remainder(10, 1)` runs ten. **The cost is set by the magnitude of the input, not by its length.**

That is why complexity is always stated against the *size* of the input rather than its value. It is also why `//` and `%` are worth having: they reach the same two answers without the loop, so the number of subtractions never appears in the running time at all. Whenever you meet a loop whose bound is a number rather than a `len`, check which of the two kinds you are looking at before naming a complexity class.

[19] courses/csse1001/python-recursion.pdf

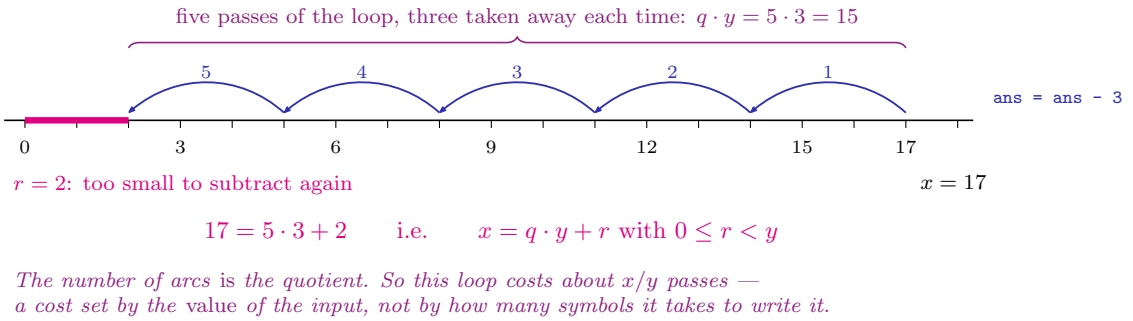


Figure 7: Repeated subtraction for $17 \div 3$. The number line, the arcs and the count of passes show what the `remainder` loop does; the equation beneath it is $17 = 5 \cdot 3 + 2$, with the general form $x = q \cdot y + r$ and $0 \leq r < y$. The code above runs on $(17, 7)$ and the arithmetic on $(17, 3)$; the figure follows the arithmetic.

Complexity of the sorting algorithms

Algorithm	Complexity
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Bubble sort	$O(n^2)$
Merge sort	$O(n \ln n)$

See sorting-algorithms[20] for how each of these works. Merge sort is written here as $O(n \ln n)$ — the same class as $O(n \log n)$, since the base is a constant factor.

Why this matters elsewhere

Complexity is what makes dynamic programming[21] worth doing: a branching recursion like the naive Fibonacci recomputes the same subproblems exponentially many times, and caching collapses that back down to linear. It is also the justification for the dictionary's "magic" hash function in python-dictionaries[22] — a hash table finds a key in roughly constant time where scanning a list would take linear time.

[20] courses/csse1001/sorting-algorithms.pdf
[21] courses/csse1001/python-recursion.pdf
[22] courses/csse1001/python-dictionaries.pdf

CSSE1001 Diagram Notation

This note records the drawing conventions **CSSE1001 actually uses**, inferred from its own lecture slides. It is the reference that later diagrams in this course should be checked against.

> ! This course ships no notation guide
>
> There is **no written style or notation standard** anywhere in the CSSE1001 materials. The course profile lists no prescribed textbook and no drawing guide; the "Learning resources" section points only to the UQ Library, Blackboard and Ed Lessons. Every convention below is **inferred from the drawings themselves**, not quoted from a rule. The three exceptions — places where the slides *state* a rule in words rather than merely demonstrate one — are the naming conventions (see Naming and capitalisation) and the one figure-source note in 12b. Everything else is observation.

How to read the citations

Decks are cited by their lecture code and **PDF page number**, e.g. `10a p7` = the UML deck, seventh page of the PDF.

Do not cite the printed footer numbers. They are unreliable: 09a runs `1/14 … 4/39, 5/39, 6/14 … 15/14, 16/15` and then ends on a page footed `14/14` that is actually PDF page 17. Several decks have footer totals that disagree with their own length (11a's last page reads `14/13`). PDF page index is the only stable address.

Colours were sampled from the rendered pages, quantised to 16 levels per channel, so hex values are accurate to roughly ±8 per channel.

What I sampled

248 pages read as images, out of 726 across 25 decks — 17 of the 25 decks. Full text was extracted from all 25 decks plus the course profile, and every deck was scanned for embedded raster images, so the unread decks were checked for figures even where I did not read them.

Deck	Pages read
01a history-of-computing	22–33
02a python-memory-model	4–28
03a functions	23–28
03b sequence-selection-and-iteration	2–19
04b non-primitive-data	2–17
07a exceptions	14–21

Deck	Pages read
07b scope	5–19
07c object-oriented-programming	2–19
08a dunder-methods	2–5
08b representation-invariants	5–14
09a composition	1–17 (all)
09b inheritance	1–19 (all)
10a uml	1–14 (all)
10b advanced-inheritance	1–20
11a model-view-controller	1–14 (all)
12a recursion	2–15
12b complexity-and-sorting	2–19

Not read as images: 00-welcome, 01b python-as-a-calculator, 02b primitive-data, 04a while-loops, 05a for-loops, 05b comprehensions, 06a string-methods, 06b file-io. An image-extraction scan across all decks found **zero embedded raster images** in seven of those eight (00-welcome has six photo pages), and they are typeset in the same LaTeX Beamer template as the decks I did sample. So the risk that they hide an unseen diagram idiom is low but **not zero** — the eight decks above are the blind spot in this note.

The two slide families

Almost every visual inconsistency in this course traces back to one fact: the decks come from **two different production pipelines**, and many decks mix both on adjacent pages.

LaTeX Beamer family — serif body text (Computer Modern), ruled code listings with line numbers, coloured named block headings, no raster images. Covers 01b, 02a, 02b, 03a, 03b, 04a, 04b, 05a, 05b, 06a, 06b, and most of 07a/07b/12a/12b.

PowerPoint family — sans-serif labels, filled colour boxes, drop shadows, stock photography, dark-theme code screenshots. Every deck carrying `Based on slides from Dr. Azadeh Ghari-Neiat` on its title page (07a, 07b, 07c, 08a, 08b, 09a, 09b, 10a, 10b, 11a) contains PowerPoint-family pages.

Consistency: this split holds across the whole semester and is the single most useful thing to know before drawing anything for this course.

Colour palette

These five colours carry meaning and recur across many decks. Everything else is decorative.

Colour	Sampled	Means	Evidence
Dark blue	`#3030B0`	Slide title	10a p4, p7, p10; 09a p12; 10b p10, p19
Magenta	`#E00080`	Defined term, emphasis, the answer	10a p4; 02a p6; 12a p2
Purple	`#92278F`	Named block heading; margin annotation	03b p4; 12b p19; 09a p12; 10b p13
Green	`#109050` (LaTeX) / `#00B050` (PPT)	A prompt to the student	02a p5; 03b p11; 10a p12; 11a p8
Red	`#F00000`	Warning / failure path	12a p10; 07b p13; 04b p14

Purple `#7030A0` is a different purple. It is used only for the UML legend boxes (10a p7, p8, p10, p11) and for the `Warning` heading in 07b p6. Do not conflate it with the block-heading purple.

Consistency: the blue/magenta/purple/green assignments hold from week 2 (02a) to week 12 (12b). Red is the least consistent — see Inconsistencies.

Named block headings

LaTeX-family slides open a block with a coloured heading on its own line, no box and no rule.

- **Green** — a prompt the student is meant to act on: `Question` (02a p5, 12b p3, p9, p12), `Exercise` (03b p11, p14; 12b p5, p8, p11, p15; 09b p17; 10a p12; 11a p8), `Example` (02a p22, p24, p25), `Task` (07b p17).
- **Magenta italic** — the resolution: `Answer` (02a p6, p7, p8, p9, p23; 12b p18).
- **Purple** — a formal or named block: `Definition (…)` (03b p4, p6, p7, p12; 04b p3; 07c p5; 12b p19), `Proposition` (12a p13), `Base Case` / `Induction Hypothesis` (12a p14), `Observe` (03b p10), `Summary` / `Next` (02a p28, 07a p21, 07b p19), `Constants` (07b p5), and named algorithms `Insertion Sort` / `Bubble Sort` / `Merge Sort` / `Quick Sort` (12b p3, p6, p9, p12, p16).

`Definition` blocks are parenthesised with the term being defined: `Definition (Boolean domain)`, `Definition (Predicate)`, `Definition (List)`, `Definition (Complexity)`.

Consistency: green/magenta/purple hold across at least five decks spanning weeks 2–12. The `Warning` heading is the exception.

Section divider slides

A section break is a slide bearing **one large magenta italic serif word**, left-of-centre and high on the page, with a short parenthetical gloss beneath it in small black roman type.

- `Variables` — *(Named memory locations.)* — 02a p14
- `Sequencing` — *(Order of instructions matters.)* — 02a p21
- `Selection` — *(Controlling program flow.)* — 03b p2
- `Lists` — *(Mutable Ordered Collections.)* — 04b p2
- `Memes` — *(Recursions through meme examples.)* — 12a p3

When a deck runs a numbered series of dividers, they take **Roman numerals with a colon**, and the series continues **across decks**:

- `I: Structure` — *(Holding named attributes.)* — 07c p8
- `II: Methods` — *(Object scoped functions.)* — 07c p15
- `III: Magic` — *(Overloading built-in functions.)* — 08a p5

Consistency: strong — six plain dividers across four decks, plus a three-part numbered series spanning two decks. One divider (08b p11, `Exercise`) carries no gloss.

Code listings

Code is by far the most frequently drawn thing in this course, and its presentation is a notation in its own right.

Ruled block. A listing is bounded by a **horizontal rule above and below**, full text width, with no vertical rules and no background fill. Line numbers run in tiny grey type in the left margin. Code is monospace with visible letter-spacing.

REPL vs. file. The two are distinguished by prompts, not by styling:

- A **REPL transcript** uses a magenta `>>>` prompt for the first line of an input and a magenta `...` for continuation lines; output follows on its own line with **no prompt** (02a p13, p15–p20; 03b p5, p8, p13, p15–p18; 04b p4–p13; 07b p5–p16; 10b p9, p11, p15, p18).
- A **file or definition** is shown with line numbers but **no prompts at all** (04b p15, p16, p17; 08b p6, p9, p13, p14; 09a p10–p14).

Colour inside code. Black is the default. Magenta marks the token or line under discussion — class names being introduced (`Battery`, `Camera`, `Motor`, 09a p10, p11), the lines inherited or overridden (10b p10, p13, p17), the `assert` keyword (08b p6, p9), the constant being referenced (08b p13, p14). Boolean operators are coloured by identity: `and` in cyan, `or` in green, held consistently through 03b p9, p10, p11, p14.

Vertical ellipsis. Elided code is a centred on its own line (03a p23) or three stacked full stops (07a p15).

Placeholder metasyntax. Things the student must substitute are written in angle brackets in monospace: `<code>` and `ExceptionName_k` (07a p15, p16), `<object_variable>.<method>(<parameters>)` (07c p16), `<statement that should be true>` (08b p5). *Consistency: three decks.*

Annotating code

This is the course's real "diagram" vocabulary — it annotates listings far more often than it draws figures.

Right-margin note. A short **purple italic serif** phrase, right-aligned, on the same line as the code it comments on. Sampled `#92278F` in both 09a p12 and 10b p13 — identical colour, three decks apart. (Measured by histogram at 300 dpi. The neighbouring aside on 09a p13 is `#92268F`, one unit off in green and blue — the same ink to the eye, but the two pages are not bit-identical.)

Examples: *inherit from A*, *override*, *extend from B from A*, *Super is B* (10b p10, p13, p14, p17, p18); *Drone has-a Battery* (09a p12); *this is the memory location where 42 is stored* (02a p13); *Note addition happened first* (03b p5); *ys is an alias of xs*, *Aliasing relationship broken!* (04b p13); *a 'private' variable*, *this is called a 'getter'* (07c p19); *unreachable if line 5 errors* (07a p14).

Consistency: the strongest single convention in the course — present in 02a, 03b, 04b, 07a, 07b, 07c, 09a, 10b. Weeks 2 through 10.

Boxed token. A thin rectangle drawn around an identifier *inside* the listing, marking the thing being named. The box colour is **not fixed**: purple around `Point()` and `self` (07c p9, p16), magenta around `raise ZeroDivisionError` (07a p18), blue around `Child` and purple around `Parent` (09b p9). See Inconsistencies.

Brace grouping. A magenta right-brace `}` spanning consecutive lines, with a magenta italic label beside it — `Attributes` beside the two `self.x = 0` / `self.y = 0` lines (07c p9). *Single instance.*

Pointer arrow. A straight or gently curved arrow from an annotation to the code element it refers to. Magenta in 07c p12 and p16; orange in 09b p12 and p13; blue and purple in 09b p9.

Colour keying. Where an annotation has a coloured box and a coloured arrow and a coloured word in the caption, all three share one colour: on 09b p9 the blue box round `Child`, the blue arrow, and the word "Child" in the caption are all blue, and `Parent` is purple throughout. *Two instances (09b p9, p12); treat as a good practice rather than a settled rule.*

Focus markers. Small **green right-pointing triangles** in the far-left margin flag the lines the reader should look at (07b p7, p12). *Two instances, one deck.*

Scope boxes

The clearest original notation in the course, in 07b.

A **rounded rectangle drawn over a span of code lines** delimits a scope, and its colour names the scope kind:

- **Purple outline, no fill** — global scope; encloses the whole snippet (07b p7, p12).
- **Blue outline, pale blue fill** — local/function scope; encloses the `def` body only (07b p7, p9, p12).

Nesting is literal: on 07b p7 and p12 the blue local box sits **inside** the purple global box.

Each box is paired with a short horizontal arrow leading out to a **colour-matched label** — purple arrow to purple `global variable`, blue arrow to blue `local variable` (07b p7, p8, p9, p12).

Consistency: four pages of one deck, entirely self-consistent. Never reused elsewhere in the semester.

UML class diagrams

The formal UML vocabulary is taught in **10a** and reused once in **10b p19**.

Class box. A black-outlined rectangle in **three compartments**, separated by full-width horizontal rules: name (centred, serif) / attributes / methods (both left-aligned). 10a p6 states the rule in words and shows it twice.

Visibility prefixes. Stated explicitly on 10a p6: `-` private, `+` public, `#` protected. Applied consistently on 10a p7–p11: `#name: str`, `-breed: str`, `+speak()`.

Attribute typing. `name: type` with a Python type — `#age: int`, `-nationality: str`, `-pets: []` (10a p6, p9, p11). Methods show a return type only sometimes: `+info(): str` (10a p6) but bare `+info()` (10a p7–p11).

Empty compartments are kept. `Cat` on 10a p8–p11 keeps a visibly empty attributes compartment rather than collapsing to two compartments.

Single-compartment boxes are acceptable when only structure is being shown — 10a p4 and 10b p19 draw name-only boxes.

Relationship symbols

Relationship	Symbol	Evidence
Inheritance	Solid line, **solid filled** triangular arrowhead, pointing **up into the parent**	10a p4, p7–p11; 10b p19
Association	Plain solid line, no head, optional verb label at one end	10a p8 (`drinks`), p9 (`Associated`)
Composition	Solid line, **filled diamond at the "whole" end**	10a p10
Aggregation	Solid line, **hollow (white-filled) diamond at the "whole" end**	10a p11

> ⚠ The inheritance arrowhead is filled, not hollow
>
> Standard UML draws generalisation as a **hollow** triangle. This course draws it **solid filled**, verified at 600 dpi on 10a p7 and again on 10b p19. Every inheritance arrow in the course is filled. If you conform to standard UML you will not match the slides; if you match the slides you will not match standard UML. **Follow the course.**

Line routing is orthogonal. All relationship lines use right-angle elbows, never diagonals (10a p8–p11). A parent with several children gets a single vertical stem down from the parent to a horizontal bus, with vertical drops to each child (10a p4, p7–p11; 10b p19).

Layout and flow. Parent above, children in a row beneath. A grandchild hangs directly below its own parent with its own vertical arrow (10b p19: `C` below `B`). Associated non-hierarchy classes sit to the **side** at the same height as the class they attach to (10a p9 `Vet`, p10 `Heart`, p11 `Owner`).

Multiplicity. Placed **above the line, adjacent to the end it qualifies**, in magenta — `1..*` at both ends (10a p9), `1` and `1` (10a p10), `0..*` (10a p11). 10a p9 gives the vocabulary in a legend box: `0..1` zero to one, `n` specific number, `0..*` zero to many, `1..*` one to many, `m..n` specific range.

Magenta marks the relationship being taught. On each of 10a p7–p11 the relationship introduced on that slide is drawn in magenta while previously-established structure reverts to black. On p7 the inheritance tree is magenta; by p8 it is black and the new association line is magenta. *Consistency: holds on all five relationship slides — a deliberate, reusable device.*

Legend box. A rounded rectangle with a **purple** `#7030A0` outline, containing the relationship name in purple and the bare symbol drawn beneath it, parked in the top-left corner (10a p7 Inheritance, p8 Association, p10 Composition, p11 Aggregation). The multiplicity legend on 10a p9 uses the same box for a text list instead of a symbol.

Non-UML class and object diagrams

Before UML is taught in week 10, the course draws classes at least **four other ways**. These are not variants of one style — they are four unrelated idioms.

(a) Uniform blue hierarchy — flat steel-blue `#4F81BD` filled rectangles, white sans-serif labels, parent above and children in a row, joined by a `#3D6696` orthogonal bracket with **no arrowhead at all** — a darker shade of the box fill, not black, though it reads as black at slide size. Photos of instances sit below each child. (09b p2, p3)

(b) Multi-colour hierarchy — parent box **black** with white text; each child a different Office-palette fill — and these are the stock Office 2007 accents exactly, which is why they recur: `#C0504D` brick red, `#4BACC6` teal, `#F79646` orange, `#215968` dark teal, olive. Connectors are thin `#4A7EBB` **straight lines** — diagonal from the outer children, vertical from the one directly beneath — and each ends in **its own arrowhead**. On 09b p5 three arrowheads overlap on `Animal`'s bottom edge; at slide size they read as one, but there are three. (09b p4, p5, p6)

Black for the root holds in all three of these diagrams (`Employee`, `Animal`, `Person`). The child colours carry no meaning I can find — `Dog` stays teal on 09b p5 even though it is itself a parent.

(c) Attributes-over-methods box — the class name in **purple bold sans-serif above** two stacked **magenta-outlined** boxes: attributes on top, methods beneath, with a photo of an instance alongside. (07c p6, three times: `Dog`, `Student`, `Smartphone`)

(d) Attributes-left / methods-right — the class in a black-outlined box in the centre with its name in black and `(class)` beneath in magenta; a **purple-outlined** attributes box to the **left** and a **magenta-outlined** methods box to the **right**, each with a plain black label above it. Instances are photographs **below**, reached by straight black **downward** arrows with solid heads. (08a p2)

> **Instantiation arrows point down; inheritance arrows point up**
>
> 08a p2 draws class → instance as a **downward** arrow. Every inheritance arrow in the course points **up** to the parent. The two are never shown on the same page, so the contrast is safe — but keep it.

(e) Containment ("has-a") boxes — composition is drawn as **nesting, not as a line**. A large steel-blue `#4080B0` filled rectangle is the whole, labelled in large white text at its right edge (`car`, `drone`); a smaller brick-red `#C05040` rounded rectangle sits **inside** it and is the part, labelled in white at its top with its methods listed beneath (`engine` / `stop()` / `start()`). (09a p6, p16)

Substitution is shown by stacking two such figures and joining them with a **thick curved orange `#F79646` arrow** from the old configuration to the new (09a p16, "Hot-Swapping").

Consistency: (e) holds twice in one deck and is the course's only purpose-built composition picture. (a)–(d) are one or two pages each.

The large worked example

10a p13 — the only whole-system class diagram in the course — is drawn in a completely different style from the rest of 10a:

- Single-compartment boxes, name only, **pastel fills with matching darker strokes**.
- Sampled fills `#D0E0F0` blue / `#D0E0D0` green / `#F0E0C0` orange, with strokes `#6080B0` / `#80B060` / `#D09000`. These are the **draw.io / diagrams.net default shape colours**, matched exactly — this figure was built in draw.io with stock styling, not drawn to match the deck.
- **Dashed lines with small open arrowheads** for association/dependency (`WTController` → `WTView`, `WTModel` → `Tank`), **solid lines with the usual filled triangle** for inheritance.
- Tiny `1` / `N` multiplicities at the line ends.
- Green fill appears on `Tank`, `Tile` and `Enemy` — the three classes that have subclasses. **This is my inference from one figure**, and `Battlefield` (blue, no subclasses) is consistent with it while `WTView` (orange, alone) is not explained. Do not rely on it.

Reference and memory diagrams

The course does not draw its own reference diagrams. It uses Python Tutor.

`pythontutor.com` output is embedded as a screenshot in 07b p7, p12 and 07c p13, p14, and named as a live-demo cue five times in 02a (p22, p24, p25, p26, p27, captioned `Python Tutor 1.` through `Python Tutor 5.`).

The Python Tutor rendering, as it appears in those screenshots:

- Two columns under bold headers **`Frames`** (left) and **`Objects`** (right).
- A frame is a **pale blue box** with a bold title (`Global frame`) and one row per name; each row is `name` right-aligned against a value cell.
- An object is a **pale yellow box** with a small red-brown type caption floating above it — `Point class`, `Point instance`, `function`.
- An attribute inside an instance is a two-cell row: grey name cell, white value cell.
- A reference is a **curved blue-grey arrow** from a frame slot to the object.
- **Aliasing is shown by two arrows converging on one object** (07c p14), versus two arrows to two separate objects for equal-but-distinct values (07c p13).

Consistency: two decks, four screenshots, plus five named references in a third deck spanning week 2 to week 7. Treat Python Tutor as this course's canonical memory picture.

Bit and memory notation

Established in 02a and legended explicitly on p4.

- **A bit is a square cell. Filled black = on/1. White with a black outline = off/0.** 02a p4 draws the two side by side and labels them `On` and `Off`.
- Held on 02a p6, p7, p8, p9, p10, p11, p12 — seven further pages.
- **A bit string is a horizontal row** of contiguous squares, read left to right, with `. . .` after the last cell for continuation (02a p10, p11).
- **Memory is rows of words stacked vertically**, each row prefixed in the left margin by its **address as a small decimal number** (02a p12: 381–388).
- Enumerating all n-bit patterns stacks one row per pattern (02a p7, p8, p9).

Consistency: very strong within 02a. Never used again — no later deck draws a bit.

Data-structure and algorithm-trace diagrams

> **i** The one explicit figure-source statement in the course
>
> 12b p2: *"Note: The diagrams from today's lecture are taken from `runestone.academy`."* This is the closest thing CSSE1001 has to a textbook lineage, and it governs **every sorting and merge-tree figure in 12b** (p4, p7, p10, p13, p14). Those figures are Runestone Academy's, not the lecturer's, which is why their visual language matches nothing else in the course.

List cells. A list is a horizontal row of adjacent square cells with a thin grey border and a soft drop shadow, one integer per cell in small sans-serif, centred. (12b p4, p7, p10, p13, p14)

Shading marks state. White = untouched; grey fills mark the cells the current step is acting on or the region already sorted. The **exact meaning of each grey level is never legended** — see Not established.

A trace runs top to bottom. Each row is one step; time advances downward. (12b p4, p7, p10)

Step captions sit to the right of each row, small black sans-serif, in the present tense of what just happened: `93 is largest`, `inserted 26`, `Exchange` / `No Exchange`, `91 in place after first pass`. (12b p4, p7, p10)

Movement arrows are thin black orthogonal arrows drawn *above* the row, running from source cell to destination cell (12b p4). A whole pass gets a long labelled arrow spanning the figure above everything else — `First pass` (12b p10).

Recursion trees (merge sort, 12b p13–p14) are drawn as binary trees of list-rows:

- `Merge Sort - Winding` (p13): the full list at the top, splitting downward, arrows pointing **down** from a list to its two halves.
- `Merge Sort - Unwinding` (p14): the singletons at the top, arrows pointing **down** into each merged result, ending in the sorted list at the bottom.

Both use downward arrows. The recursion direction is carried by the slide title and by which end of the tree holds the whole list — *not* by arrow direction. This is easy to get wrong.

Mathematical and logical notation

Truth tables (03b p6, p7). A 3×3 grid with visible black rules; the **operator name sits in the top-left cell in cyan monospace** (`and`, `or`); the header row and header column carry the operand values in black monospace; a **double rule** separates the headers from the body. The one semantically interesting result is coloured magenta — the single `True` for `and` (p6), the single `False` for `or` (p7).

Function signatures are given above the table, centred, mixing monospace names with maths: `and :  ×  →` . Also `> :  ×  →` (03b p5) and `P :  → {True, False}` (12a p11).

Blackboard bold for domains — booleans (03b p4, p5, p6, p7), integers (03b p5), naturals (12a p11, p13). `= {True, False}` is defined with the Python literals inside the set, in monospace (03b p4).

Python literals stay in monospace inside maths. `True` and `False` are never italicised into maths variables.

Implication and quantifiers use and : `(P(0) and (P(n)  P(succ(n))))  m  , P(m)` (12a p11). Note `and` stays as the Python keyword in monospace even inside a mathematical statement.

Contradiction is (02a p20).

Displayed maths is centred on its own line, in magenta when it is the point being made (02a p20, 12a p11).

Arrow chains stand in for flow diagrams. Simple pipelines are written inline as text with → rather than drawn: `state change → _check_invariants → valid` (08b p8), `parent → child → grandchild` (10b p12), `Task + TodoList → Model (domain)` (11a p8). *Consistency: three decks. This is why the course has so few real flow diagrams.*

Geometric and proof figures

The chessboard-tiling figures for the induction proof (12a p12, p14, p15):

- The construction is drawn in **cyan solid lines**.
- The underlying reference grid is **grey dashed**.
- The distinguished (uncovered) square is a **filled magenta disc**.
- Dimensions are annotated with cyan extension lines outside the figure, labelled in black serif maths (2, $2^{\,1}$), placed below and to the right.

Consistency: three pages, one deck, entirely self-consistent.

The one flow diagram

07a p17 is the **only control-flow diagram in the 726 pages**. It shows try/except/else/finally:

- Nodes are **rounded rectangles with a hand-drawn, wobbly grey stroke** (a sketch/xkcd style).
- The node label is the bare Python keyword in monospace, **coloured per keyword**: `try` teal, `except` magenta, `else` blue, `finally` purple.
- **`finally` is drawn with a dashed sketchy border**; the other three are solid.
- Flow runs **top to bottom**, with curved grey arrows looping out to the left.
- Edge captions are bold sans-serif, placed **below** the arrow they describe: *Run the code under try*, *Execute the code under except when there is an exception*, *No exceptions? run the code under else after try*, *Always run this code.*
- **Status glyphs sit outside the flow**: a purple circled beside the try→except path, a green circled beside the try→else path, a blue circled **i** beside `finally`.

> ⚠ Warning
>
> This is a **single instance**. It does not establish a general flowchart convention for the course — see Not established.

The LEGB diagram

07b p13 draws name resolution as a vertical cascade:

- Four levels listed top to bottom, each with its **initial letter bold** — **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.
- Between consecutive levels, a large **red hook arrow** with the small red caption `If not found` to its left.

- The cascade terminates in bold red **`Error!`**.

Red marks the failure path. *Single instance.*

Naming and capitalisation

The three rules the course states in words, not just in examples:

Kind	Rule	Evidence
Variables, functions	lowercase, words separated by underscores — `descriptive_variable_name`	03a p25
Classes	`CamelCaps` — "That is `ClassNamesAreLikeThis`"	07c p7
Constants	`SNAKE_CASE_CAPS`	07b p5

03a p24 states the course follows **Google's PEP**, "a variant of PEP8", with both linked as lavender rounded chips. 03a p26: lines must be **strictly less than 80 characters**.

These rules govern labels in diagrams too. UML boxes use `CamelCaps` class names (`DeliveryRobot`, `WTController`, `DroneFlight`) and snake_case members (`_check_invariants()`, `has_solar_panels`, `battery_level`) throughout 10a, 10b, 07c and 08a.

Do/don't pairs. Compared alternatives go in a **two-column table headed `Yes` and `No` in magenta italic**, with a single rule under the headers and no other lines — good on the left, bad on the right (03a p25, p28). The variant form labels two separate ruled listings *Not helpful* / *Helpful* in magenta italic (03a p27). Inside dark-theme code screenshots the same judgement is made with an all-caps comment — `# BAD!` / `# GOOD!` (08b p7).

Figure sourcing and attribution

No textbook is cited as a figure source anywhere. A full-text search of all 25 decks and the course profile for textbook, figure-source and adaptation language returns only the four items below. The course profile prescribes no textbook at all.

Attribution	Where	Scope
`Based on slides from Dr. Azadeh Ghari-Neiat`	Title page of 07a, 07b, 07c, 08a, 08b, 09a, 09b, 10a, 10b, 11a	Slide lineage, not figure lineage

Attribution	Where	Scope
`Based on slides from Dr. Azadeh Ghari-Neiat and Dr. Paul Vrbik`	07b title page	as above
`Note: The diagrams from today's lecture are taken from runestone.academy.`	12b p2	**All sorting figures in 12b**
`Image source: www.freepik.com`	09a p9	That slide's stock art
`Images are from WIRED`	09b p2, p3	Those slides' photos
`Inspired by the example of MIT6.100L slide.`	07c p10	The blueprint/house analogy — an idea, not a figure family

Placement. Credits are set in **tiny centred or bottom-right roman type** beneath the figure (09a p9, 07c p10) or in the bottom-right corner of the slide (09b p2, p3).

10b p9 refers to "the course notes" as the source of a code snippet — the course has its own written notes (delivered via Ed Lessons) which are **not** among these materials and may carry conventions this note cannot see.

Page furniture

- **Slide titles**: dark blue `#3030B0`, serif, upper-left, sentence case with title-case content words. `Optional Content:` is used as a prefix for non-examinable material (10b p5).
- **Footer**: `n / N` in tiny grey at the bottom-right. **Unreliable** — see How to read the citations.
- **Title slide**: course code and title in blue, lecture number and name in magenta italic, lecturer, date, then the attribution line. (10a p1, 09a p1, 09b p1, 10b p1, 11a p1)
- **QR-code slides**: a large centred QR code with the URL in small serif beneath, flanked by decorative 3D clip art, no title (07a p19, 07b p18, 09b p19).

Inconsistencies

Places where the course does the same thing more than one way. **Both readings are recorded; a decision is needed before drawing.**

1. Two entirely different UML idioms inside one deck

10a p4–p11 draws three-compartment black-outlined class boxes with magenta relationship lines, orthogonal routing and no fills. 10a p13 draws single-compartment pastel-filled draw.io boxes with dashed dependency arrows. Same deck, same lecture, same subject. 10b p19 adds a third variant: single-compartment boxes with **navy `#103040`** outlines. **Decide: compartments or not, fills or not, box outline colour.**

2. Class boxes are drawn four incompatible ways before week 10

Three-compartment black box (10a p6); attributes-box-above-methods-box with the name outside and magenta outlines (07c p6); attributes-left/methods-right around a central black name box (08a p2); plain filled rectangle with name only (09b p2, p4). **Decide which is canonical for pre-UML material.**

3. Inheritance connectors: three styles

Orthogonal bracket with **no arrowhead** (09b p2, p3); **straight blue lines each with its own arrowhead**, overlapping at the parent (09b p4, p5, p6); orthogonal bracket with a **filled triangular arrowhead** (10a p4, p7–p11; 10b p19). **Decide.**

4. Hierarchy box colour: uniform vs. per-child

09b p2/p3 fill every box the same steel blue. 09b p4/p5/p6 make the root black and give each child its own colour. The two appear two pages apart. And 10a p13 uses a third scheme where colour may track "has subclasses". **Decide whether fill colour means anything at all.**

5. `Warning` heading: red or purple

Red `#F00000` in 12a p10 and 04b p14. Purple `#7030A0` in 07b p6. **Decide.**

6. Two greens for the same role

`Question` / `Exercise` / `Example` / `Task` are `#109050` in the LaTeX decks (02a p5, 03b p11) but `#00B050` in the PowerPoint ones (10a p12, 11a p8). Same meaning, two hexes. Additionally 01a p25 sets `Question` in a magenta/crimson rather than either green (observed visually; my colour sample of that page missed the heading). **Decide one green.**

7. Boxed-token highlight colour is unfixed

Purple (07c p9, p16), magenta (07a p18), blue-and-purple keyed to two referents (09b p9). **Decide.**

8. Three code presentations

Ruled + line-numbered LaTeX listings (most decks); **dark-theme screenshots** of an IDE (09b p8–p13, p16; 10b p2, p4, p5; 08b p7); and plain unruled, unnumbered monospace (07a p16, p20). 09b switches between the first two within a single deck. **Decide.**

9. `Summary` styling

Purple block heading (02a p28, 07a p21, 07b p19); blue slide title (10a p14, 11a p14); magenta (09b p18). **Decide.**

10. Image credits are given inconsistently

09a p9 and 09b p2/p3 credit their stock photography. 01a p23–p29 (Ada Lovelace, Grace Hopper, the Jacquard loom, punch cards, the first computer bug), 07c p6/p10, 08a p2/p4 and 12a p4–p9 (the meme sequence) credit nothing. **Decide a credit policy.**

11. Bit-string reading order

02a p7–p9 enumerate n-bit patterns by toggling the **leftmost** cell fastest (`off-off`, `on-off`, `off-on`, `on-on`), which reads the leftmost cell as least significant. 02a p11 then interprets a word as $1\cdot2^{3\,2} + 1\cdot2^{3\,1} + \ldots + 1\cdot2$, reading the **leftmost cell as most significant**. **If you draw a numbered bit string, decide which end is the high bit.**

12. `Exercise` as heading vs. divider

A green block heading in 03b p11, 12b p5, 09b p17, 10a p12, 11a p8 — but a full magenta italic divider slide in 08b p11. **Decide.**

Not established by the source

Conventions a diagram in this course will plausibly need, that the materials simply do not settle. This is where the risk of invention lies.

No general flowchart convention. The only flow diagram in 726 pages is 07a p17's sketch-style try/except figure. There is no established shape vocabulary (no decision diamond, no terminator, no process box), no arrow style, and no rule for labelling branches. **If you need a flowchart, you are inventing one.** Note that the course's habit is to write flow as an inline `→` chain (08b p8, 10b p12, 11a p8) rather than draw it — consider doing that instead.

No MVC diagram. The entire MVC deck (11a, 14 pages) contains **no architecture diagram**. The Model/View/Controller relationship is given as a six-item numbered list (11a p6, p13) and a four-column table (11a p5). The only place the three appear as boxes is 10a p13,

where they are three unremarkable draw.io rectangles among thirteen. There is **no established shape, layout, arrow direction, or colour** for an MVC diagram in this course.

No hand-drawn reference or memory diagram. The course never draws its own variable/object box-and-arrow picture — it screenshots Python Tutor (07b p7/p12, 07c p13/p14). If you need a reference diagram that Python Tutor cannot produce, there is no house style to follow.

No list or aliasing diagram. 04b (non-primitive data, 45 pages) teaches aliasing, copying and pass-by-reference **entirely through REPL transcripts** — mutating `ys` and observing `xs` change (p10, p12, p13). Not one box-and-arrow picture of a list. The idea of "two names, one list" is never drawn outside Python Tutor.

Grey-level semantics in the sorting traces are never legended. 12b p4, p7, p10 use at least three shades — white, light grey, darker grey — and no slide says which is "being compared", which is "sorted", and which is "just moved". Since these figures are Runestone's, the course has no stated rule to appeal to.

No stated meaning for fill colour in class boxes. 09b p5's brick red, teal and orange appear decorative; 10a p13's green *may* mark classes with subclasses but this is one figure and `WTView`'s orange contradicts any simple reading. **There is no colour key anywhere.**

No convention for showing a method body, call stack, or sequence of calls. Despite 12a (recursion, 47 pages) there is no call-tree or stack-frame diagram of the course's own making — recursion is taught with memes (p4–p9), induction on chessboards (p12–p15) and code. No UML sequence diagram, activity diagram, state diagram or object diagram appears anywhere in the course.

No rule for typing method returns in UML. 10a p6 writes `+info(): str`; 10a p7–p11 write `+info()`. Both are the same method on the same class in the same deck.

No hex values are ever published. Every colour in this note was sampled from the rendered PDFs. The course states no palette, so a redrawn figure matching these hexes matches the slides by measurement, not by declaration.

Abstract classes have no notation. 10b p3–p5 teaches abstract base classes in prose and code only. UML's italic-name convention for abstract classes is never used; `Shape` is never drawn.

Line weights, font sizes and box dimensions are never specified, and vary between the two slide families.

The eight unread decks. 00-welcome, 01b, 02b, 04a, 05a, 05b, 06a, 06b were not read as images. They contain no embedded raster images (except 00-welcome's six photo pages) and are in the LaTeX family, so they are unlikely to hide a new idiom — but this note cannot rule it out.

The written course notes. 10b p9 cites "the course notes" as a source. Those notes are delivered through Ed Lessons and are not part of these materials. If they contain diagrams, they may establish conventions that contradict this note.

See also

- 2025-10-07-uml[23] — Lecture 10A's content
- uml[24] — the UML reference note
- python-pep8-style-guide[25] — the naming rules cited above
- csse1001[26] — course index

Python Recursion

A **recursive function** is a function that calls itself. Every recursion needs at least one of each:

- **Base case**: a case defined outright, with no further recursive call (`if base_case: return constant`).
- **Recursive step**: a line where the function calls itself on "smaller" input, making progress toward a base case.

Canonical example: factorial

```
def fact(n: int) -> int:
    if not n:              # base case
        return 1
    return n * fact(n-1)   # recursive step
```

Evaluation has two phases:

- **Winding**: recursive calls build up a stack of pending computations (`4 × (3 × (2 × (1 × fact(0))))`). Each call pushes an *activation record*; they complete last in, first out.
- **Unwinding**: once the base case is hit, each pending computation resolves from the inside out.

[23] courses/csse1001/2025-10-07-uml.pdf
[24] courses/csse1001/uml.pdf
[25] courses/csse1001/python-pep8-style-guide.pdf
[26] courses/csse1001/csse1001.pdf

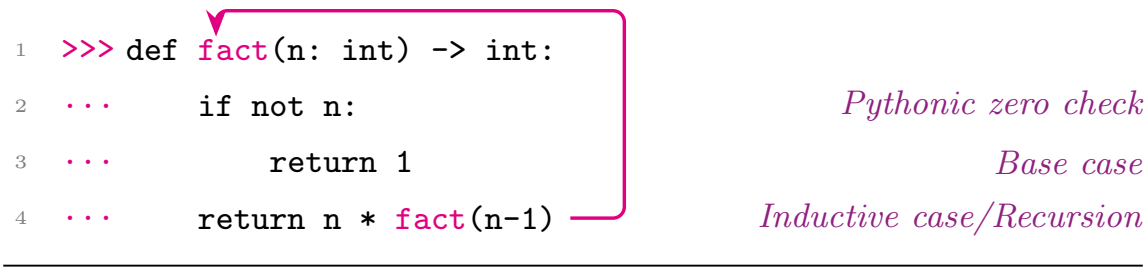


Figure 8: The anatomy of a recursive function. The purple notes on the right name each part. The two occurrences of fact are coloured magenta and joined by an arrow, to show the one thing the definition of recursion turns on: line 4 calls the function being defined on line 1.

See 2025-10-20-recursion[27] for the full winding/unwinding trace of `fact(4)` drawn out line by line.

If a base case is never reached, the recursion **bottoms out** with a `RecursionError: maximum recursion depth exceeded` once Python's call stack limit (1000 by default) is hit — `fact(-1)` does this, since `n` only gets more negative. The limit can be raised with `sys.setrecursionlimit(n)`:

```
>>> import sys
>>> sys.setrecursionlimit(1500)
```

Overflowing the stack and bottoming out are **different bugs with the same error message**. Raising the limit fixes a recursion that legitimately needs more depth than Python allows (`fact(999)` works once the limit is 1001); it never fixes a missing or unreachable base case (`fact(-1)` fails at any limit).

Choosing base cases

Type	Typical base case
`int` (counting down)	`0`
`list`	`[]`
`str`	`''` (empty string)

[27] courses/csse1001/2025-10-20-recursion.pdf

For less obvious base cases, take an example that makes a *single* recursive call and ask what the base case **must be** for that example to come out right. For example, $2^k = 2 \cdot 2^{k-1}$ leaves open what "2 multiplied 0 many times" is. Since $2^1 = 2$ and $2^1 = 2 \cdot 2^0$, the base case must be $2^0 = 1$.

The same advice stated the other way round: set the base cases to whatever makes the **singleton** case (the second-smallest one) work.

Linear vs. tree recursion

Most recursive functions make a single recursive call per invocation (linear recursion, e.g. `fact`, `is_palindrome`, `increasing`, `sum_of_digits`) and so unwind along one chain. Some make more than one (tree recursion), e.g. checking whether any sublist sums to a target:

```
def sublist_sum(xs: list[int], target: int) -> bool:
    if not xs:
        return not target
    return sublist_sum(xs[1:], target-xs[0]) or sublist_sum(xs[1:], target)
```

Two calls per invocation means 2^n leaf calls for a list of n elements. See 2025-10-20-recursion[28] for the call tree drawn out, and for the other branching examples (`num_paths`, `lcs`, `hanoi`, naive `fib`).

Tree recursion can recompute the same subproblem many times. **Dynamic programming** — caching previous results — avoids this when subproblems overlap heavily (the classic example being naive recursive Fibonacci, which is exponential without a cache but linear with one). Here the cache is a module-level dict seeded with the base cases:

```
fib_cache = {0: 1, 1: 1}
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]
```

[28] courses/csse1001/2025-10-20-recursion.pdf

Relationship to induction

Recursion is essentially the Principle of Mathematical Induction realised as code: a base case (the $P(0)$ case) plus a step that reduces a general case to a smaller one already known to work (the $P(n) \implies P(\text{succ}(n))$ step). The recursive call *is* the appeal to the induction hypothesis — which is easiest to see in the Towers of Hanoi proof, where the hypothesis is used twice and the code makes exactly two recursive calls.

See python-scope[29] for how local variables/recursive calls interact with scope, and python-composition[30]/python-inheritance[31] for other structuring tools. See towers-of-hanoi[32] for a worked induction/recursion example, and algorithmic-complexity[33] for the vocabulary that makes "exponential without a cache, linear with one" precise — merge sort and quick sort in sorting-algorithms[34] are both recursive.

Sorting Algorithms

Five ways to solve one problem — sorting a list. Each is derived from a different restriction: *suppose the only thing you could do was X, could you still sort?* Taken together they are the course's worked example for comparing algorithms, which is what algorithmic-complexity[35] formalises.

Sorting is also the course's very first example of an algorithm at all (2025-07-29-a-brief-history-of-computing[36]). The selection-sort trace below needs no code beside it: *what* is being expressed matters more than the language it is expressed in.

Selection sort

> Suppose we could only *select* (find the index of) the largest element from a list.

Find the largest member of `xs` and make it the last element. Then find the next largest and make it the second last, and so on. After k passes the last k elements are in their final positions.

Two things the trace shows that the one-line description does not. The list of nine takes only eight passes — once the last eight are placed the first element has nowhere else to go, which is the `17 ok` row. And the *work* is invisible in the picture: each row costs a full scan of the

[29] courses/csse1001/python-scope.pdf
[30] courses/csse1001/python-composition.pdf
[31] courses/csse1001/python-inheritance.pdf
[32] courses/csse1001/towers-of-hanoi.pdf
[33] courses/csse1001/algorithmic-complexity.pdf
[34] courses/csse1001/sorting-algorithms.pdf
[35] courses/csse1001/algorithmic-complexity.pdf
[36] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

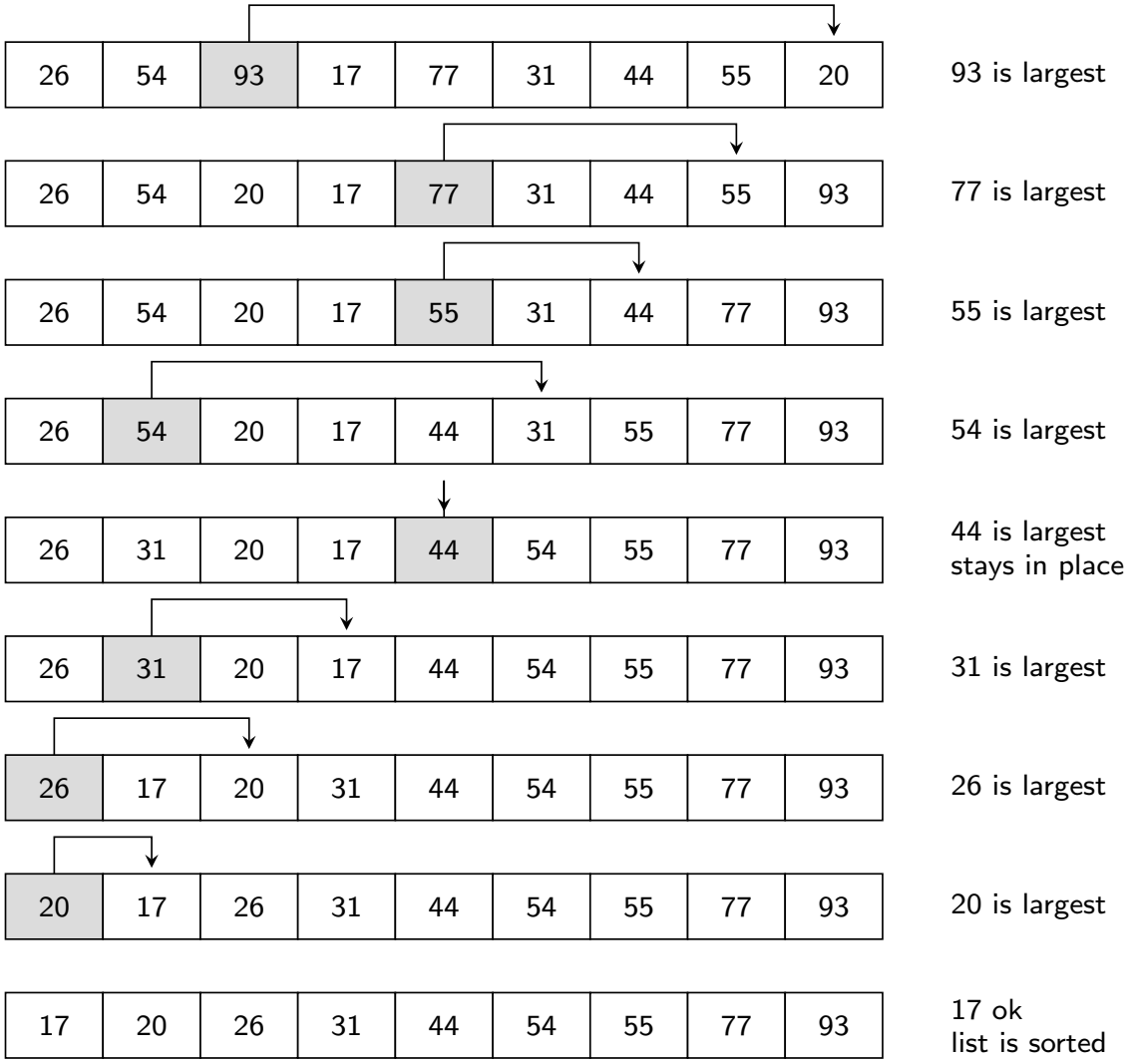


Figure 9: Selection sort on a list of nine. Each row is one pass; time runs downward. The grey cell is the largest element of the still-unsorted prefix and the arrow above the row takes it to the position it will keep, so after k rows the last k cells are final. In row five 44 is already where it belongs, so the arrow points straight down and nothing moves.

unsorted prefix to find its grey cell, and it is those scans, not the eight moves, that make selection sort quadratic.

Insertion sort

> Suppose we could only *place an element correctly into an already-sorted list*.

Sort the list by inserting elements repeatedly into the front portion of the list. After k steps the first k elements of the list are sorted (though not necessarily in their final positions relative to the rest).

Bubble sort

> Suppose the only operation we could perform was *swapping adjacent cells*.

Pass through the list swapping neighbouring elements so that larger elements move forward — larger elements "bubble" to the top. Make repeated passes until the list is sorted.

Merge sort

> Suppose the only operation we could perform was *merging two sorted lists into one sorted list*.

Break the list in half, sort each half (using merge sort — this is recursion[37]), then merge the two sorted halves.

The evaluation has the same two phases as any recursion: a **winding** phase where the list is repeatedly halved down to trivial pieces, and an **unwinding** phase where the sorted pieces are merged back up.

Quick sort

> Could we sort a list if all we could do was *split* a list into two pieces based on some predicate?

Divide the list into two lists — those greater than or equal to the head, and those less than or equal to the head. Sort those lists (recursively) and then just concatenate them.

[37] courses/csse1001/python-recursion.pdf

Which is best?

Rather than timing them, count how many times the characteristic operations are performed — for sorting, that means **comparisons** and **swaps**.

Algorithm	Complexity
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Bubble sort	$O(n^2)$
Merge sort	$O(n \ln n)$

Merge sort's divide-and-conquer structure is what buys it the better class: the halving contributes a $\log n$ factor and each level of the recursion does $O(n)$ merging work.

Exercise: implement selection, insertion, bubble and merge sort. The `hanoi` function in 2025-10-20-recursion[38] is the closest worked example of the same divide-and-conquer shape that merge sort and quick sort use.

[38] courses/csse1001/2025-10-20-recursion.pdf