Cracking the Coding Interview

Lecture 2: Geometric Series, Recursion, and Recursive Program Analysis

Proper Lecture Scribes — based on the lecture transcript and worked examples

Lecture Roadmap

So far we have seen **arithmetic sequences** under the umbrella of Big-O, Big-Ω, and Big-Θ (Lecture 1). Today's lecture adds a second, equally important tool:
geometric sequences → why they matter in real life → geometric series and their closed form → nested-loop pitfalls → recursion → the system stack and activation records.

1 Geometric Sequences: How Many Hops to Get There?

A geometric sequence starts at some value, and at every step it is multiplied by a fixed constant:

$$1,\ c,\ c^2,\ c^3,\ \ldots,\ \text{until it reaches (approximately) } n.$$

The natural question is:

> *Given that we keep multiplying by the constant c, how many hops (multiplications) does it take to reach n?*

The answer is a very famous quantity:

$$\boxed{\text{number of hops} = \log_c n}.$$

What "log" means here

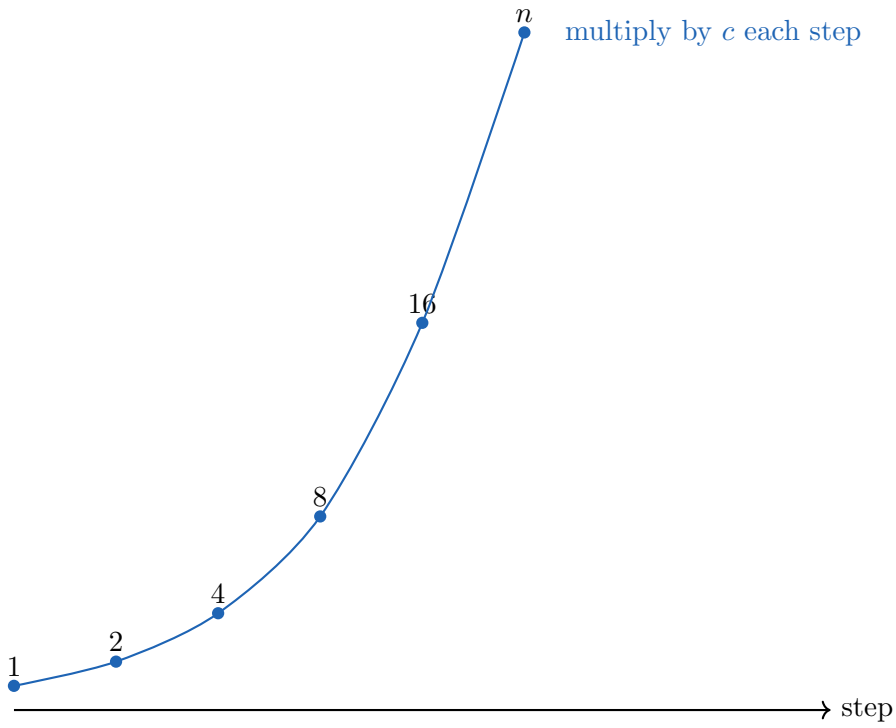
If I start from some value a (not necessarily 1) and keep multiplying by a constant c, the number of multiplications needed to reach n is $\log_c n$. The sequence does not have to start at 1 — the logarithmic hop-count is about the constant multiplicative factor, not the starting point.

1.1 The doubling example, both directions

$$\begin{aligned} 1,\ 2,\ 4,\ 8,\ \ldots,\ n \qquad & \text{(previous term before } n \text{ is } n/2) \\ n,\ n/2,\ n/4,\ \ldots,\ 8,\ 4,\ 2,\ 1 \qquad & \text{(same sequence, written backward)} \end{aligned}$$

Both directions take the same number of steps to go from one end to the other:

$$\boxed{\log_2 n}.$$



2 Loops That Multiply: $\Theta(\log_c n)$

This hop-count is exactly what a loop with a multiplicative update computes.

```
for (int i = 1; i <= n; i *= 2) {
    // constant work, cost = 1
}
```

2.1 Reading the cost directly off the sequence of i

Value of i	Cost of this pass
1	1
2	1
4	1
8	1
$\vdots$	$\vdots$
n	1

Every pass costs exactly 1, and the loop runs as many times as it takes the sequence $1, 2, 4, 8, \ldots$ to reach n. Hence the total cost is

$$\boxed{\Theta(\log_2 n)}.$$

2.2 Changing the multiplier

- `i *= 2` $\Rightarrow \Theta(\log_2 n)$.
- `i *= 3` $\Rightarrow \Theta(\log_3 n)$.
- `i *= b` for any constant $b > 1$ $\Rightarrow \Theta(\log_b n)$.

Constant-base rule

$\log_2 n$, $\log_3 n$, and $\log_b n$ all belong to the **same** asymptotic class $\Theta(\log n)$: changing the base of a logarithm only changes it by a constant factor ($\log_b n = \log_2 n / \log_2 b$), exactly the way changing the step size in an arithmetic loop only changed the constant in Lecture 1.

3 Why This Is One of the Most Important — and Most Neglected — Ideas

Geometric sequences and series are not merely a textbook curiosity. As a well-known remark from an old (circa 1990s) talk puts it, someone **should not graduate** without understanding them, because they are among the most important yet most neglected tools for reasoning about growth. Failing to understand them can be genuinely dangerous when reasoning about real processes that grow multiplicatively.

4 Real-World Geometric Growth

Several concrete stories make the danger of "linear thinking about multiplicative growth" vivid.

4.1 Viral and bacterial reproduction

A virus or bacterium reproduces by splitting itself into two—a general splitting process described here loosely as mitosis-like reproduction. If it starts as a single cell and doubles every fixed interval:

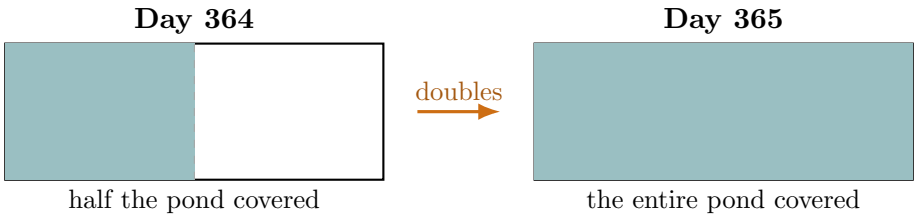
$$1 \to 2 \to 4 \to 8 \to \cdots$$

After 10 doublings there are roughly $2^{10} \approx 1024$ copies; after 20 doublings, roughly $2^{20} \approx 10^6$; after 30 doublings, roughly $2^{30} \approx 10^9$. It takes only a small number of doubling intervals to go from "one cell" to "you feel sick."

4.2 The algae pond puzzle

Imagine a pond in which algae doubles in coverage every fixed period (say, every day), starting from a tiny patch. Suppose it takes 364 days for the algae to cover **half** the pond.

Day 364 : half the pond	Day 365 : the entire pond



The trap

Anyone watching the pond up to day 364 would see only half the pond covered and might think there is still plenty of time left. In reality there is exactly **one day** left before the whole pond is gone. This is the essential danger of geometric growth: it looks harmless for a long time, and then it isn't.

4.3 Lahore's rainwater and aquifers

Lahore provides an extended case study of geometric growth, water depletion, and environmental damage:

- Lahore receives substantial monsoon rain every year. Underground, rainwater naturally recharges **aquifers** — the layers of rock/soil that store groundwater (the same water a household bore-well draws from).
- As the city urbanized, roads and housing societies (DHA-style developments) were built without preserving the natural drainage paths that let rain seep underground; instead the priority was directing rainwater into drains as fast as possible.
- Industries operating around Lahore—including the packaging industry and plastic/"shopper bag" production as a major pollutant—dumped waste directly into the ground where no organized waste-disposal system existed, contaminating groundwater with heavy metals and other pollutants.
- By contrast, older parts of the city such as Model Town, designed by Sir Ganga Ram, included sunken parks (*doonga ground*) at the center of each block specifically to collect rainwater and recharge the aquifer beneath the whole neighbourhood. This deliberate form of water-recharge engineering was later abandoned in newer development in favour of raising roads and rapidly draining rainwater away.
- The same story extends from Lahore to the entire Indus river system shared by Pakistan and India. Both countries treated rivers primarily as quantities of water to be divided, stored, diverted, and controlled. Dams, barrages, canals, and other large engineering works promised irrigation, electricity, and protection from floods, but the narrow language of "capturing" water concealed a larger question: *what happens to a living river when its natural flow is repeatedly interrupted?*
- A river carries much more than water. Its changing seasonal flow transports sediment and nutrients, replenishes floodplains and wetlands, connects tributaries, sustains fish and wildlife, recharges groundwater, and finally feeds the Indus delta. When too much water is withheld or diverted upstream, these connected habitats below the structures can shrink or disappear. Reduced freshwater and sediment reaching the delta also leave it more vulnerable to seawater intrusion, salinity, erosion, and the loss of the remarkable plant, animal, and human communities associated with the rivers of Punjab, Sindh, and Balochistan.
- This tragedy was produced not simply by one country or one dam, but by a shared human urge to dominate nature without understanding the whole system. Pakistan and India disputed who should control particular flows while often accepting the same underlying assumption: that a river is useful only when every drop has been assigned an immediate human purpose. A foreign academic advisor's research on flood estimation for the Indus illustrates this mindset: even research may begin by asking how to predict and restrain a river, rather than how much seasonal flooding, sediment movement, and ecological variation the river needs in order to remain alive.
- The same mistake appears underground. Aquifers are treated as invisible reservoirs that can be pumped indefinitely, while urban paving reduces recharge and industrial and household pollution spreads through groundwater. Extraction can grow geometrically as population, agriculture, and industry expand; contamination can also spread through connected underground water. Yet natural recharge and ecological recovery may be far slower. By the time wells run dry or poisoned water becomes visible, the accumulated damage may already be extremely difficult to reverse.
- The mathematical lesson is therefore also an ethical one: small percentage losses repeated year after year can become catastrophic resource depletion, just as small percentage increases in pollution, construction, pumping, or demand can rapidly overwhelm a finite system. Looking at a single dam, canal, housing society, factory, or bore-well in isolation hides the geometric accumulation of their combined effects. Human control may produce a short-term gain while quietly dismantling the river, delta, wetland, and aquifer systems on which long-term life depends.

- The core lesson drawn from this section: the city's **urban expansion itself has been geometric** — a society that was nearly empty a year ago can be completely built up within months — and a resource-management policy that assumes slow, linear change will fail against a geometrically growing city.

Food for thought: can restoration also be geometric?

If geometric growth in extraction, pollution, and construction helped create this crisis, what would it mean to answer it with a **geometric increase in resource creation and restoration**? Could societies begin by understanding the mathematics of compounding, then multiply rainwater harvesting, aquifer recharge, wastewater treatment, wetland and floodplain restoration, environmental river flows, efficient irrigation, and cooperation across borders? The question is not merely how humans can control more water, but how quickly we can learn to restore the living systems that create, clean, carry, and renew it.

4.4 Population growth and energy consumption

Pakistan's population has historically tended to double roughly every 10 years, while energy consumption rises with population, causing energy demand to grow on a similar timescale. If supply does not keep pace, available energy per person is effectively halved every cycle. A "solar revolution" in parts of southern Punjab provides a counter-example of adaptation: many households installed solar panels and greatly reduced their dependence on grid electricity, illustrating both the geometric problem and a locally geometric-scale solution.

4.5 Cancer growth

Cancer growth is another geometric process: early growth looks small and slow, but once past a certain point it accelerates and becomes very difficult to stop or reverse—illustrating again why catching geometric growth *early* matters far more than catching arithmetic growth early.

4.6 Compound interest

If a loan or deposit of Rs. 100 accrues 10% interest per year, compounded annually:

$$100, \quad 100(1.1), \quad 100(1.1)^2, \quad 100(1.1)^3, \ldots$$

$$100 \to 110 \to 121 \to \cdots$$

This is a geometric sequence with $a = 100$ and common ratio $r = 1.1$; after t years the amount is $100(1.1)^t$ — the same $a\,r^{\,t}$ shape as every other example in this section.

The thread connecting all of these

Biology, urban planning, economics, and computer science all reduce to the same mathematical object: $a,\ ar,\ ar^2, \ldots$. The practical skill is recognizing this shape—and specifically recognizing when a process is geometric rather than arithmetic.

5 Geometric Series: Summing the Terms

5.1 A concrete pattern first

$$1 + 2 + 4 = 7, \qquad \text{next term} = 8.$$

$$1 + 2 + 4 + 8 = 15, \qquad \text{next term} = 16.$$

$$1 + 2 + 4 + 8 + 16 = 31, \qquad \text{next term} = 32.$$

The pattern: the sum of all doubling terms up to (but not including) some term is always exactly **one less** than the next term.

5.2 The general claim

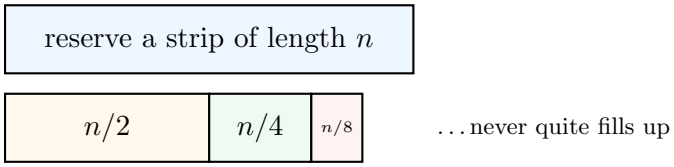
If the terms go $1, 2, 4, \ldots, n/2$ (so $n/2$ is the last term reached, and n is the *next* one), then

$$\boxed{1 + 2 + 4 + \cdots + \frac{n}{2} = n - 1.}$$

Consequently, if we instead summed all the way up to and including n itself,

$$1 + 2 + 4 + \cdots + n \;\leq\; 2n.$$

5.3 Why: a geometric (visual) proof



Reserve a strip of length exactly n. Shade a block of length $n/2$ (the largest term below n) — a strip of length $n/2$ remains unshaded. Shade the next term, $n/4$, inside that remaining strip — a strip of length $n/4$ remains. Continue: at every step, exactly **half** of what is left gets shaded, so the shaded total **never exceeds** n, no matter how many terms are added. Since we reserved a strip of length n for just the terms below n, the full running sum (terms below n, plus n itself) is bounded by

$$n + n = 2n.$$

6 Different Common Ratios, Different Constants

The same style of bound applies for other ratios, with a different (and generally smaller) constant.

Ratio 3

$$1 + 3 + 9 + 27 + \cdots + n.$$

If the last term is n, the previous term is $n/3$, and the one before that is $n/9$. Using the same "reserve a strip and shade a shrinking fraction" argument (now shading $1/3$ at each step instead of $1/2$), the sum is bounded by a constant multiple of n that is noticeably **smaller** than $2n$.

The key takeaway regardless of ratio

For any fixed ratio $r > 1$, a geometric series whose last term is n satisfies

$$1 + r + r^2 + \cdots + n = \Theta(n),$$

because the sum is always dominated by its **last** term, up to a constant that depends on r (bigger r gives a bigger constant, smaller r gives a smaller constant, but the growth class stays $\Theta(n)$).

7 The General Closed-Form Formula

For an exact value, rather than merely a bound, use the same shift-and-subtract trick credited to Gauss.

Let

$$S = a + ar + ar^2 + \cdots + ar^{n-1}.$$

Multiply every term by r:

$$rS = ar + ar^2 + \cdots + ar^{n-1} + ar^n.$$

Subtract S from rS — every interior term cancels:

$$rS - S = ar^n - a.$$

$$S(r-1) = a(r^n - 1) \quad \Longrightarrow \quad \boxed{S = \frac{a\,(1 - r^n)}{1 - r}}.$$

7.1 Sanity check against the doubling example

Take $a = 1$, $r = 2$, and suppose the number of terms is $n' = \log_2 n$ (so the last term reached is exactly n):

$$S = \frac{1\,(1 - 2^{\log_2 n})}{1 - 2} = \frac{1 - n}{-1} = n - 1.$$

This matches the pattern found by hand in Section 5.2 exactly: summing $1, 2, 4, \ldots$ up to the term just below n gives $n - 1$.

8 Nested Loops: Why You Cannot Just Multiply Loop Counts Blindly

One of the most important warnings is that seeing two nested loops does **not** automatically tell you the complexity—you must trace what each loop actually does.

8.1 Example 1 — a common mistake: this is $\Theta(n)$, not $\Theta(n \log n)$

```
for (int i = 1; i <= n; i *= 2)
    for (int j = 1; j <= i; j++)
        // constant work
```

The outer loop takes the values $i = 1, 2, 4, 8, \ldots, n$ — that is $\Theta(\log n)$ passes. But the *inner* loop's work is not constant: it runs i times. So the total work is

$$1 + 2 + 4 + 8 + \cdots + n,$$

exactly the geometric series from Section 5! By Section 5.2,

$$\boxed{1 + 2 + 4 + \cdots + n \le 2n = \Theta(n)}.$$

The trap

People often guess $\Theta(n \log n)$ here because there is a $\log n$-length outer loop and an inner loop that looks like it depends on n. The correct answer is $\Theta(n)$. Counting the number of outer passes is not the same as summing the actual work done across those passes.

8.2 Example 2 — true $\Theta(n \log n)$: the loops must be independent

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j *= 2)
        // constant work
```

Here the inner loop's bound (n) does **not** depend on i — the two loops are independent. The outer loop runs n times; for each of those, the inner loop independently runs $\Theta(\log n)$ times. Hence

$$\boxed{\Theta(n \log n)}.$$

The difference between this example and Example 1 is entirely about **dependence**: whether the inner loop's bound depends on the outer loop's current variable.

8.3 Five closely related geometric-loop patterns

The following examples look almost identical, but small changes to the bounds and update expressions produce different complexity classes. Assume $n \geq 1$ and that every loop body performs constant work.

A multiplicative loop must not start at zero

A loop written as `for (int i = 0; i <= n; i *= 2)` never progresses: multiplying zero by two still gives zero. It is therefore an infinite loop for $n \geq 0$. Every multiplicative loop below starts at 1.

Pattern 1: linear outer loop, logarithmic inner loop

```
for (int i = 0; i <= n; i++)
    for (int j = 1; j <= n; j *= 2)
        // constant work
```

The outer loop runs $n + 1 = \Theta(n)$ times. On every pass, the inner loop takes the values $1, 2, 4, \ldots, n$, so it runs $\Theta(\log n)$ times. The loops are independent:

$$T(n) = (n+1)\big(\lfloor \log_2 n \rfloor + 1\big) = \boxed{\Theta(n \log n)}.$$

Pattern 2: replacing the inner bound by n^2

```
for (int i = 0; i <= n; i++)
    for (int j = 1; j <= n * n; j *= 2)
        // constant work
```

The inner loop now runs $\Theta(\log(n^2))$ times. By the logarithm rule,

$$\log_2(n^2) = 2 \log_2 n = \Theta(\log n).$$

Squaring the bound changes only the constant factor inside the logarithm, not the asymptotic class. Therefore

$$T(n) = \Theta(n) \cdot \Theta(\log(n^2)) = \boxed{\Theta(n \log n)}.$$

Pattern 3: two independent logarithmic loops

```
for (int i = 1; i <= n; i *= 2)
    for (int j = 1; j <= n; j *= 2)
        // constant work
```

Both loops independently run $\Theta(\log n)$ times. Each outer pass executes the complete inner loop, so their counts genuinely multiply:

$$T(n) = \Theta(\log n) \cdot \Theta(\log n) = \boxed{\Theta((\log n)^2)}.$$

Pattern 4: logarithmic outer loop, but growing inner work

```
for (int i = 1; i <= n; i *= 2)
    for (int j = 1; j <= i; j++)
        // constant work
```

The inner loop does not run n times on every outer pass. It runs $1, 2, 4, 8, \ldots$ times as i doubles. Hence the total is a geometric sum:

$$T(n) = 1 + 2 + 4 + \cdots + 2^{\lfloor \log_2 n \rfloor}.$$

The largest power of two in this sum lies between $n/2$ and n, and a doubling series is less than twice its largest term. Thus

$$T(n) = \boxed{\Theta(n)},$$

not $\Theta(n \log n)$. This is the same dependence principle demonstrated in Example 1.

Pattern 5: growing inner work up to n^2

```
for (int i = 1; i <= n * n; i *= 2)
    for (int j = 1; j <= i; j++)
        // constant work
```

The outer loop has only $\Theta(\log(n^2)) = \Theta(\log n)$ passes, but the inner work grows with i. The total work is

$$1 + 2 + 4 + \cdots + 2^{\lfloor \log_2(n^2) \rfloor}.$$

Its largest term is within a constant factor of n^2, and the whole geometric sum is within a constant factor of that largest term. Therefore

$$T(n) = \boxed{\Theta(n^2)},$$

not $\Theta(n^2 \log n)$.

Comparison rule

If the inner bound is independent of the outer variable, multiply the two iteration counts. If the inner bound depends on the current outer value, write the work as a sum over the actual outer values. For $i = 1, 2, 4, \ldots$, this often produces a geometric series dominated by its largest term.

8.4 Example 3 — a hidden cost inside a loop condition

```
int strlen(char *s) { // Theta(n) -- counts up to the null terminator
    int i = 0;
    while (s[i] != '\0') i++;
    return i;
}

for (int d = 2; d <= strlen(s); d++) {
    // constant work
}
```

It looks like a single loop running roughly n times, where n is the string length. But the loop *condition* `d <= strlen(s)` is re-evaluated on **every single iteration**, and each call to `strlen` costs $\Theta(n)$ on its own. So the true cost is

$$n \text{ iterations} \times \Theta(n) \text{ per condition check} = \boxed{\Theta(n^2)}.$$

The trap

A single visible `for` loop is not automatically $\Theta(n)$. Any function call sitting inside the loop condition (or inside the loop body) must be analyzed on its own and its cost added in, every single time it is invoked.

8.5 Example 4 — loops do not simply "multiply": add up each line's own cost

Suppose a helper function does $\Theta(n^4)$ internal work and then returns just a single value n (say, by returning the fourth root of an accumulated quantity that grew to size n^4):

$$A(n): \ \Theta(n^4) \text{ work}, \quad \text{returns } n.$$

Now consider a loop whose *guard condition itself* calls $A(n)$:

```
for (int i = 1; i <= A(n); i++) {
    for (int j = 1; j <= n; j *= 2) {
        // constant work
    }
}
```

- The outer guard `i <= A(n)` is re-checked on every pass, exactly like Example 3. The outer loop runs n times (since $A(n)$ returns n), and each check costs $\Theta(n^4)$, giving $\Theta(n \cdot n^4) = \Theta(n^5)$ just from evaluating the guard repeatedly.
- The inner loop body, on its own, is the independent $\Theta(\log n)$ loop from Example 2, run once per outer pass: $n \cdot \Theta(\log n) = \Theta(n \log n)$.

These two costs are added, not multiplied against each other in some other combination, because they are two separate sources of work happening across the same structure:

$$T(n) = \Theta(n^5) + \Theta(n \log n) = \boxed{\Theta(n^5)},$$

by the bottleneck principle from Lecture 1 — the fastest-growing term dominates.

The lesson of Section 9

Do not assume nested loops' complexities simply multiply, and do not assume a loop is cheap just because you can't see an explicit inner loop. Identify *every* independent source of cost — including function calls hidden in guards — analyze each on its own, and only then combine them (by **addition** for sequential/repeated costs, by **multiplication** only when one genuinely nested count multiplies another).

8.6 A brief aside on Python's `for` loops

Python's `for i in range(n)` builds its range representation once at the start and then iterates over it, so the "bookkeeping" cost is itself linear. This does not change the asymptotic class of surrounding code—$\Theta(2n)$ or $\Theta(3n)$ are both still $\Theta(n)$—but it is a reminder that even a language's own loop construct has a cost that, in principle, should be accounted for.

9 Introduction to Recursion

Where the idea comes from

Recursion is a direct borrowing from mathematics. Consider the mathematical definition of factorial:

$$n! = n \cdot (n-1)!.$$

This defines $n!$ in terms of a *smaller* instance of the same problem, $(n-1)!$.

Written this way, the definition is **incomplete** — there must be a stopping criterion, or the definition never bottoms out. The complete mathematical definition adds a base case:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

Translating this directly into code

Exactly the same two-part structure carries over into programming:

1. A **stopping (base) case**, which in a program must be checked first — the value the function returns with no further recursive call.
2. A **recursive case** that calls the function again, but always on a strictly smaller version of the problem ($n-1$ instead of n).

9.1 Factorial in C++

```
int fact(int n) {
    if (n == 0) return 1; // base / stopping case
    return n * fact(n - 1); // recursive case
}

int main() {
    int q = 5;
    int answer = fact(q); // answer becomes 120
}
```

9.2 Factorial in Python

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

q = 5
answer = fact(q)
```

10 The System Stack: Activation Records

The mechanical fact demonstrated with a debugger

Every function call — recursive or not — creates a new entry on the **system (call) stack** called an **activation record**. Stepping through `fact(5)` in a debugger shows a brand-new local variable `n` appear on the stack every time `fact` calls itself, and a small reserved space is left alongside it.

That reserved space holds exactly two things:

1. The **return address**: which line (and column) of code called this frame, i.e. where control must go back to.
2. The **return value slot**: the box that will eventually hold the value this call sends back to its caller.

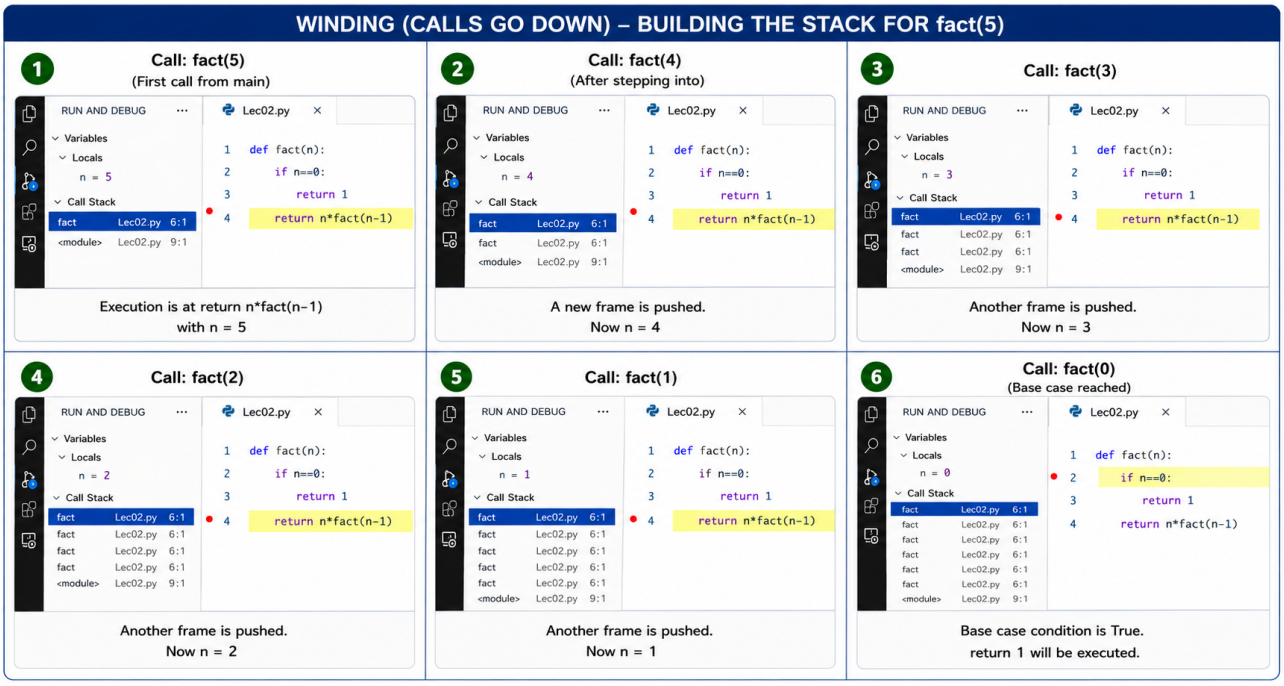
10.1 Winding: building the stack for `fact(5)`

The base case here is $n = 0$, so calling `fact(5)` pushes activation records for $n = 5, 4, 3, 2, 1, 0$ — six frames in total.

Winding: six activation records, $n = 5, 4, 3, 2, 1, 0$

`fact(0)`	base case → returns **1** immediately
`fact(1)`	return address + return value *(reserved, empty)*
`fact(2)`	return address + return value *(reserved, empty)*
`fact(3)`	return address + return value *(reserved, empty)*
`fact(4)`	return address + return value *(reserved, empty)*
`fact(5)`	return address + return value *(reserved, empty)*

stack grows as each call is pushed



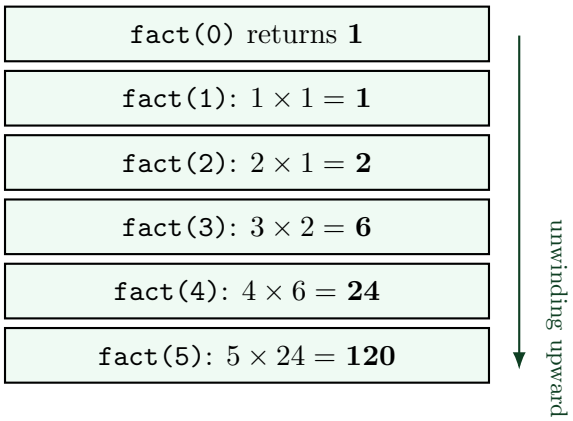
Each frame's local variable `n` is different $(5, 4, 3, 2, 1, 0)$, even though every frame is running the exact same code — this is why the same line `n * fact(n-1)` evaluates to something different in every frame.

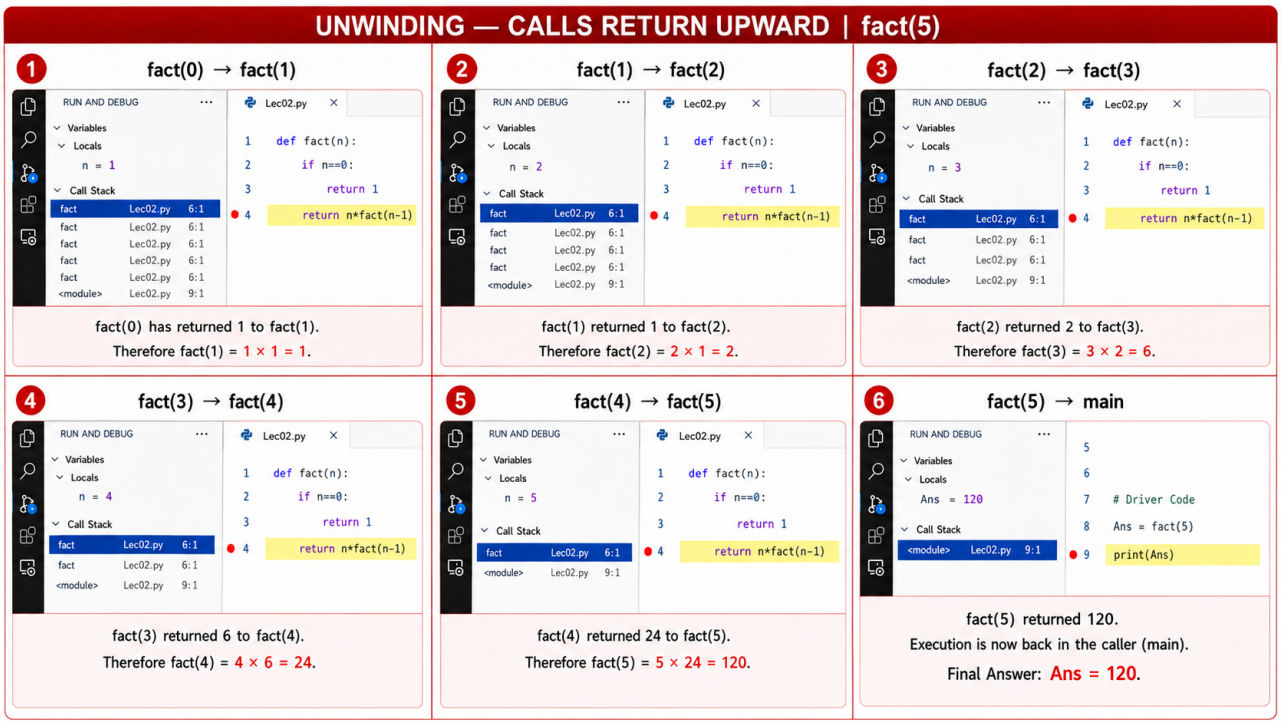
10.2 Unwinding: how the return address and return value are used

When `fact(0)` returns, its return **value** (1) is written into the reserved slot left by its caller (`fact(1)`), and control jumps back to the reserved **return address** — the exact line inside `fact(1)` that made the call. `fact(1)` then re-executes its pending line, `n * fact(n-1)`, now that `fact(n-1)` finally has a concrete value. Its own frame is then popped (erased) from the stack, and the same process repeats one level up.

Order of return	Frame	Pending expression	Value received	Value returned
1	`fact(0)`	(base case)	—	1
2	`fact(1)`	$1 \times$ `fact(0)`	1	$1 \times 1 = 1$
3	`fact(2)`	$2 \times$ `fact(1)`	1	$2 \times 1 = 2$
4	`fact(3)`	$3 \times$ `fact(2)`	2	$3 \times 2 = 6$
5	`fact(4)`	$4 \times$ `fact(3)`	6	$4 \times 6 = 24$
6	`fact(5)`	$5 \times$ `fact(4)`	24	$5 \times 24 = 120$

Unwinding — frames popped from the top, in reverse call order





Order of evaluation is reversed from "code order"

The multiplication written first in the source ($n \times \texttt{fact}(n-1)$ for $n = 5$) is actually the **last** one to be computed. The innermost call resolves first; the outermost call resolves last, because it is the one waiting on everything else.

Two phases: one clean mental model

1. **Build the whole stack first**, all the way down to the base case, without computing anything.
2. **Compute from the base case upward**: the base case resolves immediately; every level above it re-executes its pending line only once the value below it is known, then pops and hands its own result up.

This is exactly what "execute" means for recursion — it is not compiled away in advance; it genuinely builds the stack, resolves the base case, and rolls back up one frame at a time. Recursion, at the implementation level, is simply the system stack being used as the underlying data structure.

11 What's Deliberately Left for Later

The *time complexity* of recursive programs remains for the next stage of the discussion. Turning a recursive trace like the one above into a formal bound follows the introduction of mathematical induction, the complementary tool for reasoning about recursive definitions.

12 Key Takeaways from Lecture 2

1. A geometric sequence multiplies by a fixed constant c at every step; the number of hops needed to go from a starting value to n is $\log_c n$.
2. A loop of the form `i *= c` runs $\Theta(\log_c n)$ times, and different constant bases all fall in the same class $\Theta(\log n)$.
3. Geometric growth is deceptive: a process can look barely-there for a long time and then complete almost immediately (the algae-pond puzzle: half the pond on day 364, the whole pond on day 365).
4. Biological reproduction, urban expansion, population/energy growth, cancer growth, and compound interest are all real examples of the same $a\,r^{\,t}$ shape.
5. The sum $1+2+4+\cdots+n$ is bounded by $2n$ — the running sum of a doubling series never exceeds twice its last term, shown by a shrinking-remainder (shading) argument.
6. Different common ratios give different constants but the same growth class: a geometric series whose last term is n is always $\Theta(n)$ for any fixed ratio $r > 1$.
7. The general closed form, derived by the shift-and-subtract trick, is $S = \dfrac{a(1-r^n)}{1-r}$.
8. Nested loops do not have an automatic complexity from "eyeballing" them: an outer $\log n$ loop combined with an inner loop that runs i times is $\Theta(n)$, not $\Theta(n \log n)$ — you must sum the actual work, not just count outer passes.
9. A function call hidden inside a loop's guard condition is re-evaluated on every iteration and can dominate the whole algorithm's cost (e.g., calling a $\Theta(n)$ `strlen` inside a loop condition turns an apparent $\Theta(n)$ loop into $\Theta(n^2)$).
10. Recursion is a direct translation of a mathematical recursive definition (like $n! = n \cdot (n-1)!$) into code, and always needs an explicit, reachable base case.
11. Every function call creates an activation record on the system stack holding a return address and a return-value slot; recursion first winds all the way down to the base case, then unwinds, resolving one pending computation per popped frame, in the reverse order the calls were made.

One sentence to remember

Multiplicative growth looks small for a long time and then stops looking small very suddenly — and recursion is just this same idea, implemented as a stack that builds all the way down before it computes anything at all.

Practice Direction from the Lecture

The next session introduces mathematical induction as the natural counterpart to recursive definitions, after which the formal time-complexity analysis of recursive programs (turning a call trace like `fact(5)`'s into a proper recurrence and bound) will be covered.