CodeForge — Flashcards

A Spark & Anvil printable flashcard deck. Quiz yourself on the CodeForge cast's curriculum — one card at a time.

How to use these cards

Fold each sheet down the middle line so the answers are hidden. Read the **question** on the left, say your answer out loud, then **unfold** to check the right side. Testing yourself — then checking — is one of the most powerful ways to remember. Get one wrong? That's the best kind of practice: try it again tomorrow.

Want real flashcards? Cut along the fold line and the rows into cards — question on one side, answer on the other.

---

1. Sequencing — the order in which statements run

✂ Question (fold →)	Answer
Indentation or braces in code typically show:	**which statements belong to a block** — Indentation/braces group statements into blocks (e.g. a loop or if body).
A single instruction to the computer (like an assignment or a print) is called a:	**statement** — A statement is one executable instruction in a program.
In a program, statements normally execute:	**in order, from top to bottom, one at a time** — Unless control flow says otherwise, statements run sequentially, top to bottom.
Two prints, print('1+1') then print(1+1), produce:	**1+1 then 2** — '1+1' is a string (printed literally); 1+1 is evaluated to 2.
Given: x = 3; x = 5; print(x). The output is:	**5** — The second assignment overwrites the first, so x holds 5 when printed.
An ALGORITHM is:	**a finite sequence of steps that solves a problem** — An algorithm is a well-defined, finite sequence of steps to accomplish a task.
Given: p = 4; q = p; p = 7; print(p + q). The output is:	**11** — q copied p's value 4; then p became 7; p + q = 7 + 4 = 11.
Why is understanding sequencing essential to programming?	**the order of steps determines what a program does** — Every program is fundamentally an ordered sequence of steps; getting the order right is the foundation of correctness.
A statement that produces no visible output but changes a value is a:	**assignment** — An assignment updates a variable's value without printing anything.
Sequencing, selection (conditionals), and iteration (loops) are the three basic:	**control structures of programming** — Sequence, selection, and iteration are the fundamental control structures that build any program.
A COMMENT in code is:	**a note for humans that the computer ignores** — Comments document code for people; the interpreter/compiler ignores them.
If a program should greet the user THEN ask their name, the greet statement goes:	**before the ask statement** — To greet first, its statement must come before the ask in the sequence.
If step 2 of a recipe-like program depends on step 1's result, then step 1 must come:	**before step 2** — A step that uses an earlier result must run after that result is produced.
Changing the ORDER of statements in a program:	**can change what the program does** — Because execution is sequential, reordering statements can change the result.
A program's execution flow can be changed from strict top-to-bottom by:	**control structures (conditionals, loops, function calls)** — Conditionals, loops, and calls redirect the otherwise sequential flow.
A program with only sequencing	**runs the same fixed set of steps in order** — Without conditionals or loops,

(no ifs or loops) always:	the program runs the identical steps every time.
Given: print('A'); print('B'); print('C'). The output order is:	**A, then B, then C** — Statements run in sequence, so the prints happen A, B, C in order.
The value of x after: x = 1; x = x + 2; x = x * 3 is:	**9** — x = 1 → 1 + 2 = 3 → 3 * 3 = 9.
Given: name = 'Ada'; greeting = 'Hi ' + name; print(greeting). The output is:	**Hi Ada** — name holds 'Ada', so 'Hi ' + name is 'Hi Ada'.
If two independent statements don't depend on each other, swapping them:	**gives the same result** — Statements with no data dependency can be reordered without changing the outcome.
The FIRST thing a program does when it runs is execute:	**its first statement** — Execution begins at the program's first (top) statement.
Given: total = 0; total = total + 10; print(total). The output is:	**10** — total starts at 0, then becomes 0 + 10 = 10, which is printed.
Given: a = 2; b = a; a = 9; print(b). The output is:	**2** — b was assigned a's value (2) at that moment; changing a later doesn't change b.
Swapping two statements where the second uses the first's variable will likely:	**cause an error or wrong result (the value isn't ready yet)** — If you use a variable before assigning it, the program errors or uses a wrong/undefined value.
Reading code to predict its exact behavior before running is called:	**tracing** — Tracing means mentally executing the code line by line to predict its output/state.

2. Variables and data types — assignment and values

✂ Question (fold →)	Answer
The statement x = x + 1 means:	**take x's current value, add 1, and store the result back in x** — The right side is evaluated first (old x + 1), then stored back into x — increment by one.
Dividing the integer 7 by 2 with true division gives:	**3.5 (a float)** — True division of 7 / 2 yields the float 3.5.
A good variable name is one that:	**clearly describes the value it holds (e.g. studentCount)** — Descriptive names make code readable and maintainable.
Which is a valid variable name in most languages?	**user_age** — Names can't start with a digit, contain spaces, or be keywords; user_age is valid.
A STRING type holds:	**text (a sequence of characters)** — A string is a sequence of characters, like 'hello'.
The value 42 (no quotes, no decimal) is an:	**integer** — A whole number with no decimal or quotes is an integer.
A variable's TYPE tells the computer:	**what kind of value it holds and what operations are valid** — The type determines valid operations (you can add ints, concatenate strings, etc.).
Converting the string '7' to an integer lets you:	**do arithmetic with it (e.g. add numbers)** — Converting '7' to the integer 7 enables numeric operations.
Uninitialized variables (used before assignment) typically cause:	**an error or undefined behavior** — Reading a variable before giving it a value is an error in most languages.
A boolean variable is most useful for:	**storing a yes/no or on/off state** — Booleans naturally represent two-state conditions (true/false, on/off).
The type of the value returned by the comparison 5 > 3 is:	**boolean** — A comparison evaluates to a boolean (true here).
Why are variables fundamental to programming?	**they let a program store, update, and reuse values as it runs** — Variables give programs memory — the ability to hold and manipulate changing data.
Adding the integer 3 and the float 2.5 typically gives:	**the float 5.5** — Combining an integer and a float usually promotes the result to a float: 5.5.
A BOOLEAN type holds:	**one of two values: true or false** — A boolean is either true or false — the basis of decisions.
The word for the '=' operation (store a value into a variable) is:	**assignment** — '=' performs assignment: store the right-hand value into the left variable.
A VARIABLE is best described as:	**a named storage location that holds a value** — A variable is a name bound to a storage location holding a value that can change.
Reassigning a variable to a new value:	**replaces the old value** — A new assignment overwrites the variable's previous value.
A FLOAT (or double) type holds:	**numbers with a decimal point (like 3.14)** — Floats represent real numbers with fractional parts.

The value 3.0 is a:	**float** — A number written with a decimal point (3.0) is a float, even though it equals 3.
After a = 10; a = 20, the variable a holds:	**20 (only its most recent value)** — A variable holds only its latest assigned value; the earlier value is overwritten.
The value true is a:	**boolean** — true (and false) are boolean values.
An INTEGER type holds:	**whole numbers (like −3, 0, 42)** — Integers are whole numbers without a fractional part.
The value 'hello' (in quotes) is a:	**string** — Text in quotes is a string.
Adding the strings 'cat' + 'dog' typically gives:	**'catdog' (concatenation)** — The + operator joins (concatenates) strings, giving 'catdog'.
The statement x = 5 means:	**store the value 5 into the variable x** — '=' is assignment (store the right side into the left), NOT a permanent math equality.

3. Conditionals — making decisions with if/else

✂ Question (fold →)	Answer
Given score = 60: if score >= 90: print('A') elif score >= 60: print('D') else: print('F'). Output:	**D** — score >= 90 is false; score >= 60 is true, so it prints 'D' and skips the rest.
A condition is just:	**a boolean expression that evaluates to true or false** — Every condition reduces to a boolean value that selects a branch.
Given: if 5 > 3: print('yes') else: print('no'). The output is:	**yes** — 5 > 3 is true, so the if-branch runs, printing 'yes'.
A NESTED conditional is:	**an if statement inside another if statement** — Nesting places one conditional inside another to test combined conditions.
An 'elif' (else-if) is used to:	**test another condition when the previous ones were false** — elif adds additional conditions checked only if the earlier ones failed.
The result of the condition (10 == 5) is:	**false** — 10 does not equal 5, so the test is false.
A condition that combines two tests with AND is true only when:	**both tests are true** — An AND condition requires both sub-conditions to be true.
Given: if 2 > 7: print('A') else: print('B'). The output is:	**B** — 2 > 7 is false, so the else-branch runs, printing 'B'.
An if statement with NO else, when the condition is false:	**simply does nothing and continues** — If the condition is false and there's no else, execution just moves on.
The operator >= means:	**greater than or equal to** — '>=' is true when the left value is greater than OR equal to the right.
Nesting an if inside another if lets you:	**test a second condition only when the first is true** — The inner if is only reached (and checked) when the outer condition is true.
An IF statement runs its block:	**only when its condition is true** — The if block executes only if the condition evaluates to true.
Why are conditionals essential to programming?	**they let programs respond differently to different situations** — Decision-making via conditionals is what lets software adapt its behavior to inputs and states.
The comparison operator for 'not equal' is:	**!=** — '!=' tests whether two values are not equal.
A condition that is always true (like 1 == 1) makes the if-block:	**always run** — A perpetually true condition always executes the if-block.
In a chain if/elif/else, once a condition is TRUE:	**its block runs and the rest are skipped** — The first true branch runs; the remaining elif/else branches are skipped.
In an if/else statement, exactly how many branches run?	**one (the if-branch if true, otherwise the else-branch)** — An if/else selects exactly ONE branch based on the condition — never both.
The operator < means:	**less than** — '<' tests whether the left value is less than the right.

Given x = 10: if x > 5: print('big'). The output is:	**big** — x is 10, and 10 > 5 is true, so it prints 'big'.
Conditionals let a program:	**choose different actions based on data (branch)** — Conditionals give programs decision-making — different behavior for different inputs.
Given x = 3: if x > 5: print('big'). The output is:	**nothing (the condition is false and there is no else)** — 3 > 5 is false and there's no else, so nothing prints.
Given temp = 30: if temp > 25: print('hot') else: print('mild'). Output:	**hot** — 30 > 25 is true, so the if-branch prints 'hot'.
The operator <= means:	**less than or equal to** — '<=' is true when the left value is less than OR equal to the right.
The operator that TESTS equality (not assignment) is:	**== (double equals)** — '==' compares for equality; '=' assigns a value — confusing them is a classic bug.
The result of the condition (10 == 10) is:	**true** — 10 equals 10, so the equality test is true.

4. Boolean logic — and, or, not, and truth tables

✂ Question (fold →)	Answer
The result of (true OR false) is:	**true** — OR is true when AT LEAST ONE operand is true; here one is true, so the result is true.
A condition (x == 5 OR x == 10) is true when x is:	**5 or 10** — The OR is true if x equals either value.
Boolean logic is used inside __ to decide what runs:	**conditionals and loops** — Conditions in if statements and loops are boolean expressions that control execution.
(NOT false) evaluates to:	**true** — NOT flips false to true.
The OR operator is sometimes written as:	**||** — Many languages write OR as '||'.
Short-circuit evaluation of (false AND anything) means the second part:	**is not evaluated (the result is already false)** — If the first operand of AND is false, the result is false regardless, so the second isn't evaluated.
The expression (NOT (5 > 3)) evaluates to:	**false** — 5 > 3 is true; NOT true is false.
Given x = 15: the expression (x > 0 AND x < 10) is:	**false** — 15 > 0 is true but 15 < 10 is false, so AND gives false.
Given x = 5: the expression (x > 0 AND x < 10) is:	**true** — 5 > 0 is true AND 5 < 10 is true, so the whole expression is true.
The AND operator is sometimes written as:	**&&** — Many languages write AND as '&&' (OR as '||', NOT as '!').
Given a = true, b = false: (a AND NOT b) evaluates to:	**true** — NOT b is true; a AND true is true.
(true AND true) evaluates to:	**true** — AND is true when both operands are true.
The result of (true AND false) is:	**false** — AND is true only when BOTH operands are true; here one is false, so the result is false.
NOT (a AND b) is equivalent to (De Morgan's law):	**(NOT a) OR (NOT b)** — De Morgan's law: negating an AND turns it into an OR of the negations.
Why is boolean logic foundational to computing?	**all decisions and digital circuits reduce to true/false logic** — Every conditional, loop, and even the hardware's logic gates operate on boolean true/false values.
The result of (false OR false) is:	**false** — OR is false only when both operands are false.
The result of (NOT true) is:	**false** — NOT flips the value: not true is false.
De Morgan's law says NOT (a OR b) equals:	**(NOT a) AND (NOT b)** — Negating an OR turns it into an AND of the negations.
The result of (false AND false) is:	**false** — AND with both false is false.
A common boolean misconception comes from everyday language: programming OR is INCLUSIVE, meaning (true OR true) is:	**true** — Unlike everyday 'either...or', logical OR is inclusive: true OR true is true.
AND returns true only when:	**both operands are true** — Logical AND requires both inputs to be true for a true result.

OR returns true when:	**at least one operand is true** — Logical OR is true if either (or both) inputs are true.
A truth table lists:	**the output for every combination of inputs** — A truth table enumerates all input combinations and the resulting boolean output.
The value of (5 > 3) AND (2 > 4) is:	**false** — 5 > 3 is true but 2 > 4 is false; true AND false is false.
(true OR true) evaluates to:	**true** — OR is true whenever at least one operand is true (here both are).

5. Loops — repeating actions with iteration

✂ Question (fold →)	Answer
Using a loop instead of copy-pasting code 100 times is better because it is:	**shorter, clearer, and easier to change** — A loop avoids duplication, making the code concise and maintainable.
A 'break' statement in a loop:	**immediately exits the loop** — break stops the loop entirely and continues after it.
The body of a loop is:	**the block of code that repeats** — The loop body is the repeated block of statements.
Reversing the order of a loop's iterations is done by:	**counting down instead of up** — Iterating from high to low (counting down) reverses the order.
An INFINITE loop happens when:	**the loop's condition never becomes false** — If nothing ever makes the condition false, the loop runs forever (an infinite loop).
Why are loops powerful in programming?	**they let a few lines of code process large or repeated amounts of work** — Loops let short code handle huge datasets and repetitive tasks efficiently.
An OFF-BY-ONE error is:	**looping one time too many or too few** — Off-by-one (fencepost) errors come from miscounting loop boundaries — running once too many or too few.
Given: total = 0; for i in range(1, 4): total = total + i; print(total). Output:	**6** — total = 0 + 1 + 2 + 3 = 6.
The loop 'for i in range(1, 4)' produces the values:	**1, 2, 3** — range(1, 4) yields 1, 2, 3 (start inclusive, end exclusive).
A LOOP is used to:	**repeat a block of code** — Loops repeat a block of statements, avoiding copy-pasting the same code.
A NESTED loop is:	**a loop inside another loop** — Nesting places one loop inside another, useful for grids and combinations.
A for loop 'for i in range(5)' runs its body:	**5 times** — range(5) yields 0,1,2,3,4 — five iterations.
How many times does this print? for i in range(3): print(i) (range(3) gives 0,1,2)	**3 times** — range(3) yields 0, 1, 2 — three values — so it prints three times.
To repeat an action once for each item in a list, you use a:	**loop (e.g. a for-each loop)** — A for-each loop iterates over each element of a collection.
A 'continue' statement in a loop:	**skips to the next iteration** — continue skips the rest of the current iteration and proceeds to the next.
A while loop whose condition starts false runs its body:	**0 times** — If the condition is false at the first check, the body never runs.
A nested loop where the outer runs 3 times and the inner runs 4 times executes the inner body:	**12 times** — The inner body runs 4 times per outer iteration × 3 outer iterations = 12 times.
	repeat a specific number of times (or over a collection) — A for

A FOR loop is typically used to:	loop iterates a set number of times or over each item in a collection.
Given: i = 0; while i < 3: print(i); i = i + 1. The last value printed is:	**2** — It prints 0, 1, 2; when i becomes 3, 3 < 3 is false and the loop stops.
A counter variable in a loop is typically:	**initialized before the loop and updated each iteration** — A counter starts before the loop and is updated (e.g. incremented) each pass.
To sum the numbers 1 to 10, a loop should:	**add each number to a running total** — Accumulate each value into a total variable inside the loop.
A loop that processes each item of a 10-item list runs:	**10 times** — A for-each loop runs once per element — 10 times.
In a while loop, if the loop variable is never updated inside the loop, the loop:	**may run forever (infinite loop)** — If the condition variable never changes, the condition stays true and the loop never ends.
A WHILE loop repeats:	**as long as its condition remains true** — A while loop keeps running its body while the condition is true, re-checking each pass.
A loop's CONDITION is checked:	**before each iteration (for a while loop)** — A while loop re-evaluates its condition before every iteration.

6. Functions and abstraction — reusable named blocks

✂ Question (fold →)	Answer
Calling a function that doesn't return anything (only prints) and trying to store its result gives:	**no useful value (often 'None'/null)** — A print-only function returns no meaningful value, so storing its result gives None/null.
Why are functions and abstraction central to writing large programs?	**they manage complexity by breaking programs into reusable, understandable pieces** — Abstraction and functions let programmers build and reason about large systems without holding every detail at once.
You run a function by:	**calling it (by its name, with any arguments)** — Defining a function creates it; calling it (name + arguments) runs it.
ABSTRACTION in programming means:	**hiding complex details behind a simple interface** — Abstraction lets you use a function by its name without knowing its internal details.
Calling a function multiple times with the same arguments (no side effects) gives:	**the same result each time** — A pure function returns the same output for the same inputs every call.
Giving a function a clear, descriptive NAME helps because it:	**communicates what the function does** — A good name documents the function's purpose, aiding readability.
A FUNCTION (procedure) is:	**a named, reusable block of code** — A function bundles code under a name so it can be reused by calling it.
A RETURN value is:	**the result a function sends back to the caller** — return hands a value BACK to the caller (usable in further code); print just displays text — they're different.
A function can take:	**zero, one, or many parameters** — Functions may have any number of parameters, including none.
Passing 3 and 4 to a function def f(a, b), the parameter a receives:	**3 (the first argument)** — Arguments fill parameters left to right: a = 3, b = 4.
Given def square(n): return n * n, the call square(4) returns:	**16** — n = 4, so it returns 4 * 4 = 16.
A PARAMETER is:	**a named input a function receives** — Parameters are named inputs listed in the function definition.
The keyword commonly used to send a value back from a function is:	**return** — return hands a value back to the caller.
An ARGUMENT is:	**the actual value passed to a function when you call it** — Arguments are the concrete values supplied for the parameters at call time.
A variable defined INSIDE a function is usually:	**local — it exists only within that function** — Local variables are scoped to the function and don't exist outside it.
Given a function add(a, b) that returns a + b, the call add(2, 3) returns:	**5** — The arguments 2 and 3 fill a and b; the function returns 2 + 3 = 5.

Helper functions are small functions that:	**do one focused subtask used by larger functions** — Helpers handle focused subtasks, keeping larger functions clean.
Decomposing a big problem into smaller functions makes code:	**easier to understand, test, and reuse** — Breaking a task into functions (decomposition) improves clarity, testability, and reuse.
A function that RETURNS a value can be used:	**in an expression (e.g. x = add(2, 3))** — A returned value can be stored or used anywhere a value is expected.
A function's RETURN value differs from PRINT because return:	**gives a value back to be used; print only displays it** — return produces a usable value; print just shows text on screen.
Functions support the DRY principle, which means:	**Don't Repeat Yourself — avoid duplicating code** — DRY (Don't Repeat Yourself) — encapsulate repeated logic in a function instead of copying it.
Abstraction lets you call a function like sqrt(x) without:	**knowing how it computes the square root internally** — You use the function by its interface, ignoring its internal implementation.
A function defined but never CALLED:	**does not run** — Defining a function only creates it; it runs only when called.
The main benefit of functions is:	**reuse and abstraction (write once, use many times)** — Functions let you write logic once, reuse it, and hide (abstract) its details behind a name.
The same function called with different arguments:	**can produce different results** — A function's output depends on its arguments, so different inputs give different outputs.

7. Lists — ordered collections and indexing

✂ Question (fold →)	Answer
Iterating a list and summing gives you its:	**total** — Adding each element yields the sum (total) of the list.
The list [1, 2, 3] has a length of:	**3** — It contains three elements.
Appending 4 to the list [1, 2, 3] gives:	**[1, 2, 3, 4]** — Append adds the new element to the end.
A list can hold:	**many values, accessed by index** — Lists store multiple values, retrieved by their index positions.
A LIST (array) is:	**an ordered collection of values** — A list stores multiple values in order, accessed by position (index).
An empty list has a length of:	**0** — An empty list contains no elements, so its length is 0.
To process every element of a list, you use a:	**loop** — A loop lets you visit and process each element in turn.
Modifying scores[0] = 95 on the list [90, 85, 100] makes it:	**[95, 85, 100]** — Assigning to index 0 replaces the first element with 95.
In most programming languages, list indexing starts at:	**0** — Most languages use 0-based indexing: the first element is at index 0.
Why are lists a foundational data structure?	**most programs work with collections of data, and lists organize them by position** — Ordered collections are everywhere in programming, and lists are the basic tool for storing and indexing them.
Removing an element from a list makes its length:	**one smaller** — Removing an element decreases the list's length by one.
Given scores = [90, 85, 100], scores[1] refers to:	**85** — Index 1 is the second element, 85.
The number of elements in a list is its:	**length** — A list's length is how many elements it contains.
Given the list [10, 20, 30], the element at index 2 is:	**30** — Index 2 is the third element (0, 1, 2), which is 30.
Accessing a negative index like list[-1] in many languages gives:	**the last element** — In several languages, index −1 refers to the last element.
Given the list [10, 20, 30], the element at index 0 is:	**10** — Index 0 is the first element, which is 10.

Iterating a list of 5 items with a loop runs the loop body:	**5 times (once per element)** — A for-each loop runs its body once per element — five times for a 5-item list.
For the list [5, 6, 7, 8], the LAST valid index is:	**3 (length 4, indices 0–3)** — A length-4 list has indices 0, 1, 2, 3, so the last valid index is 3 (length − 1).
Lists are useful because they let you:	**store and manage many related values under one name** — Lists group related values so a program can handle collections with one variable.
To ADD an element to the end of a list, you typically:	**append it** — Appending adds a new element to the end of the list.
Accessing index 4 in the list [5, 6, 7, 8] causes:	**an out-of-bounds error** — Valid indices are 0–3; index 4 is out of bounds and errors.
To find the largest value in a list, you typically:	**loop through, tracking the maximum seen so far** — Scan the list, updating a running maximum whenever a larger value appears.
The first element of a list is at index:	**0** — Zero-based indexing puts the first element at index 0.
The INDEX of an element is its __, while the VALUE is the __:	**position; content stored there** — The index is the position number; the value is the data stored at that position.
Given items = ['a', 'b', 'c'], items[2] is:	**'c'** — Index 2 is the third element, 'c'.

8. Maps and dictionaries — key→value lookup

✂ Question (fold →)	Answer
Looking up a missing key 'zzz' in a map often results in:	**an error or a default 'not found' value** — An absent key typically raises an error or returns a not-found default.
Adding a new pair 'grape': 2 to a map:	**inserts a new key–value entry** — Assigning a value to a new key adds that key–value pair.
Assigning map['apple'] = 10 when 'apple' already exists:	**overwrites the old value (now 10)** — Assigning to an existing key replaces its value; keys stay unique.
Given ages = {'Sam': 12, 'Kai': 15}, ages['Kai'] is:	**15** — The key 'Kai' maps to 15.
A map is ideal when you need to:	**look up a value quickly by a meaningful key** — Maps give fast lookup by key (like a phone book: name → number).
Why are maps an important data structure?	**they provide fast, meaningful lookup that models real relationships (name→data)** — Maps model key-based relationships and enable efficient lookup, central to countless real programs.
You retrieve a value from a map using its:	**key** — Unlike a list (indexed by position), a map is accessed by key.
Keys in a map must be:	**unique (no duplicates)** — Each key appears once; assigning to an existing key overwrites its value.
A map/dictionary is accessed by __ rather than by numeric position:	**key** — Maps look up values by key, not by positional index.
Given the map {'apple': 3, 'pear': 5}, looking up 'pear' gives:	**5** — The key 'pear' maps to the value 5.
Adding pen: 2 to the map {book: 5}:	**adds a new key 'pen' with value 2** — Assigning a value to a new key inserts that pair.
Compared to searching a list, looking up a value by key in a map is usually:	**faster** — Map (hash) lookup is typically much faster than scanning a list for a value.
Looking up a key that does NOT exist in a map typically:	**gives an error or a default 'not found' value** — Accessing an absent key usually raises an error or returns a not-found sentinel, depending on the language.
The difference between a list and a map is that a list uses __ while a map uses __:	**numeric indices; keys** — Lists are indexed by position (0,1,2…); maps are indexed by arbitrary unique keys.
Choosing a map over a list is wise when your data is naturally:	**labeled by a key rather than ordered by position** — When you access data by a meaningful label (a name, an id) rather than a position, a map fits best.
A MAP (dictionary) stores data as:	**key–value pairs** — A map/dictionary associates each unique key with a value for fast lookup.

To count how many times each word appears in a text, a good structure is a:	**map (word → count)** — A map from each word to its count is the natural tool for tallying frequencies.
Two keys in a map:	**must be different (unique)** — Keys are unique; a repeated key overwrites the previous value.
Iterating a map lets you visit:	**each key–value pair** — Looping over a map visits every key (and its associated value).
A map is a good choice for storing:	**a mapping like username → score** — Key→value relationships (username→score) are naturally maps.
A map is also commonly called a:	**dictionary (or hash map / associative array)** — Maps go by dictionary, hash map, or associative array in different languages.
Another name for a map is a:	**dictionary** — A map is commonly called a dictionary (or hash map).
A map's VALUES:	**can be duplicated (only keys must be unique)** — Two different keys can map to the same value; only keys must be unique.
Iterating a map lets you access:	**each key and its value** — Looping a map visits each key–value pair.
The values in a map:	**can repeat across different keys** — Different keys can share the same value; only keys must be unique.

9. Strings — text as data, slicing and formatting

✂ Question (fold →)	Answer
The LENGTH of the string 'code' is:	**4** — 'code' has four characters: c, o, d, e.
String comparison is usually CASE-sensitive, so 'Cat' == 'cat' gives:	**false** — Uppercase and lowercase differ, so 'Cat' and 'cat' are not equal.
The string 'hi' repeated 3 times ('hi' * 3 in many languages) gives:	**'hihihi'** — Multiplying a string by 3 repeats it: 'hihihi'.
Joining two strings together is called:	**concatenation** — Concatenation combines strings end-to-end (e.g. 'a' + 'b' = 'ab').
A single character like 'x' is:	**a string of length 1** — In many languages a single character is just a length-1 string.
Formatting a message like 'Score: ' + str(score) requires converting the number because:	**you can't concatenate a number directly onto a string** — To join a number into a string with +, convert it to a string first.
A SLICE of a string gives:	**a substring (a range of characters)** — Slicing extracts a contiguous portion (substring) of a string.
Why are strings a crucial data type?	**text is everywhere — names, messages, files, and web content are all strings** — Nearly every program handles text; strings are the fundamental way to store and manipulate it.
The number 100 and the string '100' are:	**different types (a number vs text)** — 100 is an integer; '100' is a string — different types with different behavior.
String FORMATTING is used to:	**insert values into text (e.g. 'Hi, ' + name)** — Formatting builds text with embedded values, like personalized messages.
Looping over a string lets you process:	**each character in turn** — A loop visits each character of a string one at a time.
Accessing a single character by position in a string uses:	**indexing (like a list, usually from 0)** — Strings are indexed like lists; index 0 is the first character.
A common string misconception is that '5' + '3' gives 8. Actually it gives:	**'53' (string concatenation, not addition)** — With strings, + concatenates: '5' + '3' = '53', not the number 8.
A STRING is:	**a sequence of characters (text)** — A string is ordered text — a sequence of characters like 'hello'.
Strings are indexed like lists, so 'abc'[1] is:	**'b'** — Index 1 is the second character, 'b'.
The length of the empty string '' is:	**0** — An empty string has no characters, so its length is 0.
In 'DATA', the character at index 0 is:	**'D'** — Index 0 is the first character, 'D'.
Joining a list of words ['a', 'b', 'c'] with spaces gives:	**'a b c'** — Joining with a space separator produces 'a b c'.
	true (identical strings) — Two identical strings are equal, so the comparison

Comparing 'apple' == 'apple' gives:	is true.
Checking if 'cat' contains 'a' typically returns:	**true** — 'a' appears in 'cat', so the containment check is true.
Converting the number 42 to a string gives:	**'42' (text that can't be used in arithmetic)** — Converting to a string makes '42' text; you can't do arithmetic with it until you convert back.
The string comparison 'apple' < 'banana' is often true because:	**'a' comes before 'b' alphabetically** — Strings compare character by character; 'a' < 'b', so 'apple' < 'banana'.
'Hello' + ' ' + 'World' produces:	**'Hello World'** — Concatenating the three strings (including the space) gives 'Hello World'.
To use '5' + '3' as numbers, you must first:	**convert the strings to numbers** — Convert '5' and '3' to integers first, then 5 + 3 = 8.
Converting 'Hello' to uppercase gives:	**'HELLO'** — An uppercase conversion makes every letter capital: 'HELLO'.

10. Search algorithms — linear vs binary

✂ Question (fold →)	Answer
A key REQUIREMENT for binary search is that the data is:	**sorted** — Binary search only works on sorted data; otherwise halving the search space is invalid.
Binary search on a sorted list of 1000 items takes at most about:	**10 comparisons ($\log_2$ 1000 ≈ 10)** — Each step halves the range, so ~$\log_2(1000)$ ≈ 10 comparisons suffice.
For a large sorted list, binary search is generally __ than linear search:	**much faster** — By halving the search space each step, binary search is far faster on large sorted data.
For a tiny list of 5 items, linear vs binary search performance is:	**roughly similar (size is too small to matter much)** — On tiny inputs the difference is negligible; complexity matters at scale.
After comparing to the middle, binary search discards:	**the half that cannot contain the target** — It eliminates the half where the target can't be, halving the search each step.
If a list is UNsorted, the appropriate simple search is:	**linear search** — Binary search needs sorted data; for unsorted data, linear search is the direct choice.
A SEARCH algorithm is used to:	**find whether (and where) a value is in a collection** — Search locates a target value within a collection (or reports it's absent).
The 'guess a number 1–100, higher/lower' game is an example of:	**binary search** — Halving the range with each higher/lower guess is exactly binary search.
Linear search on a list of 1000 items takes at most:	**1000 comparisons (checking every element)** — In the worst case linear search checks all 1000 elements.
Choosing the right search algorithm depends on:	**whether the data is sorted and how large it is** — The best search depends on the data's order and size — the core of algorithmic thinking.
The main advantage of linear search over binary search is that it:	**works on unsorted data with no preparation** — Linear search needs no sorted precondition, so it works anywhere with zero setup.
Why is understanding search algorithms important?	**finding data efficiently is one of the most common and important tasks in computing** — Efficient search underlies databases, web engines, and countless applications — algorithm choice matters at scale.
Searching for a value that is NOT in a list with linear search:	**checks every element, then reports not found** — Linear search scans the whole list before concluding the value is absent.
Binary search discards HALF the remaining elements at each step, which is why it's:	**logarithmic (very efficient) for large data** — Halving each step gives logarithmic growth — a huge speedup for large inputs.
Linear search can be used on data that is:	**sorted OR unsorted** — Linear search makes no assumption about order, so it works on any collection.
The trade-off of binary search	

is that it requires the data to be:	**kept sorted (which itself takes work)** — Binary search's speed assumes sorted data; maintaining that sort has its own cost.
BINARY search works by:	**repeatedly halving a SORTED collection to zero in on the target** — Binary search checks the middle of a sorted range and discards the half that can't contain the target.
Linear search is simplest to implement because it:	**just checks each element in order** — Linear search needs no preconditions — just a straightforward scan.
A search that returns the POSITION of a found item reports its:	**index** — A search commonly returns the index (position) where the item was found.
Binary search fails to work correctly if the list is:	**not sorted** — On unsorted data, discarding a half is invalid, so binary search gives wrong results.
LINEAR search works by:	**checking each element one by one from the start** — Linear search examines each element in order until it finds the target or reaches the end.
If you'll search a list many times, it can pay off to:	**sort it once, then use binary search repeatedly** — One sort enables many fast binary searches — worthwhile for repeated queries.
Binary search on the sorted list [1,3,5,7,9] for 7 first checks:	**the middle element (5)** — Binary search checks the midpoint (5) first, then searches the right half.
Binary search is a 'divide and conquer' algorithm because it:	**repeatedly splits the problem in half** — Splitting the search space in half each step is a divide-and-conquer strategy.
Binary search returns quickly when the target is:	**found at (or near) a midpoint it checks** — If the target is a midpoint it checks early, binary search finds it in few steps.

11. Sorting and algorithmic complexity (Big-O intuition)

✂ Question (fold →)	Answer
For large n, which grows the SLOWEST (most efficient)?	**O(log n)** — Logarithmic growth is far slower than linear, quadratic, or cubic — the most scalable here.
Efficient general-purpose sorts (like merge sort) run in about:	**O(n log n)** — Good comparison sorts achieve O(n log n) — much better than $O(n^2)$ for large data.
Complexity analysis usually ignores constant factors because it focuses on:	**how the work SCALES, not the exact operation count** — Big-O captures scaling behavior; constant factors matter less as n grows large.
An algorithm with a nested loop over the same n elements is typically:	**quadratic time, $O(n^2)$** — A loop inside a loop over n elements does ~n × n = n^2 work — $O(n^2)$.
O(1) (constant time) means the work:	**does not depend on the input size** — Constant time takes the same amount of work regardless of input size (e.g. one array access).
O(n) means the number of operations grows:	**in proportion to the input size** — Linear O(n): double the input, roughly double the work.
A SORTING algorithm:	**arranges elements into order (e.g. ascending)** — Sorting rearranges a collection into a defined order.
An algorithm doing n × n comparisons on n items has complexity:	**$O(n^2)$** — n × n work is quadratic — $O(n^2)$.
Binary search's complexity is:	**O(log n)** — Halving the search space each step gives logarithmic O(log n) time.
An algorithm that checks every element once is:	**linear time, O(n)** — One pass over n elements is O(n) — linear time.
Sorting a list is often done BEFORE:	**binary search (which needs sorted data)** — Because binary search requires sorted data, sorting is a common prerequisite.
Reasoning about complexity lets a programmer:	**predict whether an algorithm will stay fast as data grows** — Complexity reasoning predicts scalability — essential for real, large-scale software.
An algorithm that always takes the same time regardless of input size is:	**O(1) (constant time)** — Constant time (like one array access) doesn't depend on n.
A simple sort like bubble/selection sort is typically:	**$O(n^2)$** — Basic comparison sorts compare many pairs, giving $O(n^2)$ time.
For large n, which grows the FASTEST (least efficient)?	**$O(n^2)$** — Among these, quadratic $O(n^2)$ grows fastest, becoming slow for large inputs.
For very large data, an O(n log n) sort is preferred over $O(n^2)$ because it:	**does far less work as n grows** — O(n log n) scales much better than $O(n^2)$, making it the practical choice for big data.
Why is understanding complexity essential for a programmer?	**it lets you predict and compare how algorithms perform as data grows** — Reasoning about complexity is how programmers choose algorithms that stay fast as inputs scale.

ALGORITHMIC COMPLEXITY (Big-O) describes:	**how an algorithm's work grows as the input size grows** — Big-O expresses how runtime (or memory) scales with input size n.
Complexity analysis focuses on LARGE inputs because:	**differences between algorithms matter most as n grows** — For small inputs any algorithm is fast; complexity reveals which scales as data grows large.
Between O(n) and O(n^2), for large n the more EFFICIENT is:	**O(n)** — Linear O(n) grows far slower than quadratic O(n^2) for large inputs.
An O(n) algorithm on 1,000,000 items does roughly __ operations vs an O(n^2) doing roughly __:	**a million; a trillion** — n = 10^6 gives n = 10^6 (linear) vs n^2 = 10^{12} (quadratic) — a million vs a trillion operations.
The 'Big' in Big-O refers to describing:	**the growth rate for large inputs (an upper bound)** — Big-O characterizes how the work scales as input grows large (an asymptotic upper bound).
Choosing an efficient algorithm matters because:	**an inefficient algorithm can be unusably slow on large data** — The right algorithm can turn an impossible computation into an instant one at scale.
Sorting is often a PREPROCESSING step because it enables:	**faster subsequent operations like binary search** — Sorting once unlocks efficient later operations (binary search, easy min/max/median).
Selection sort repeatedly:	**finds the smallest remaining element and places it next** — Selection sort picks the minimum of the unsorted part and moves it into place each pass.

12. Recursion — solving a problem in terms of a smaller copy of itself

✂ Question (fold →)	Answer
The call stack in recursion holds:	**each pending function call waiting for its result** — Each recursive call is pushed onto the stack until the base case returns and the stack unwinds.
Without a base case, a recursive function will:	**recurse forever (until it crashes / stack overflow)** — Missing a base case means the function never stops calling itself — a stack overflow.
Computing factorial(3) recursively expands to:	**3 × 2 × 1 × (0! = 1) = 6** — factorial(3) = 3 × factorial(2) = 3 × 2 × factorial(1) = 3 × 2 × 1 = 6.
The key question to check any recursion is:	**does it have a base case and make progress toward it?** — Every correct recursion needs a base case and shrinking subproblems — verify both.
Compared to iteration, recursion for a simple count is often:	**less efficient (extra call overhead) but sometimes clearer** — Recursion adds call overhead; it's chosen for clarity on self-similar problems, not raw speed.
A recursive Fibonacci is elegant but can be SLOW because it:	**recomputes the same subproblems many times** — Naive recursive Fibonacci repeats overlapping subproblems, causing exponential work (a case for memoization).
Recursion expresses a problem in terms of:	**a smaller instance of the same problem** — A recursive solution reduces to a smaller version of itself.
Why is recursion an important problem-solving technique?	**it elegantly solves problems with self-similar or nested structure** — Recursion expresses naturally self-similar problems clearly and is fundamental to many algorithms and data structures.
The two required parts of any recursive function are:	**a base case and a recursive case** — Every recursion needs a base case (to stop) and a recursive case (to shrink and self-call).
The simplest input a recursive function handles directly is the:	**base case** — The base case is solved directly without further recursion.
Recursion is natural for a folder-inside-folder file system because it is:	**a nested, self-similar structure** — Folders containing folders are self-similar — a perfect fit for recursion.
countdown(3) that prints n then calls countdown(n−1) with base case n==0 prints:	**3, 2, 1** — It prints 3, 2, 1, then n==0 stops before printing 0.
RECURSION is when a function:	**calls itself to solve a smaller version of the problem** — A recursive function solves a problem by calling itself on a smaller subproblem.
A recursive definition of a problem often mirrors:	**the natural self-similar structure of the problem** — Recursion fits problems that contain smaller copies of themselves (trees, factorials, fractals).
Every correct recursion needs a BASE CASE, which is:	**a condition that stops the recursion without another self-call** — The base case is the simplest case, solved directly, that stops the recursion.
Each recursive call gets its OWN copy of the:	**parameters and local variables** — Each call has its own stack frame with its own parameters/locals.

A recursive function that never reaches its base case will:	**crash (stack overflow)** — Endless self-calls overflow the call stack.
A factorial function n! can be defined recursively as:	**n × (n−1)!, with base case 0! = 1** — n! = n × (n−1)!, and the base case 0! = 1 stops the recursion.
The RECURSIVE case:	**breaks the problem into a smaller subproblem and calls itself** — The recursive case reduces the problem toward the base case via a self-call.
For recursion to terminate, each recursive call must move:	**closer to the base case** — Each self-call must shrink the problem so it eventually reaches the base case.
Recursion is especially natural for problems like:	**traversing trees or nested structures** — Hierarchical/nested structures (trees, folders) map naturally onto recursion.
A common recursion misconception is that it's 'magic.' Actually each recursive call:	**is a normal function call that returns a value used by the caller** — Recursion is just ordinary function calls stacking up and returning — no magic, just self-calls.
Any problem solvable with recursion can generally also be solved with:	**iteration (loops)** — Recursion and iteration are equally powerful; each fits some problems more naturally.
The sum 1+2+...+n can be defined recursively as:	**n + sum(n−1), with base case sum(0) = 0** — sum(n) = n + sum(n−1); base sum(0) = 0.
A recursive call that does NOT shrink the problem toward the base case:	**risks infinite recursion** — Without progress toward the base case, recursion may never terminate.

13. Debugging and tracing — finding and fixing faults

✂ Question (fold →)	Answer
A common debugging misconception (the 'superbug') is believing the computer:	**figures out what you MEANT — it only does exactly what you wrote** — The computer executes your literal code, not your intent; bugs come from a mismatch between the two.
Changing MANY things at once while debugging is risky because:	**you can't tell which change fixed (or broke) things** — Isolating one variable at a time keeps cause and effect clear.
The BEST way to debug is often to:	**reproduce the bug reliably, then narrow down its cause** — Reproducing the bug and isolating the cause (changing one thing at a time) is the systematic approach.
A RUNTIME error is:	**an error that occurs while the program runs (e.g. dividing by zero)** — Runtime errors happen during execution, like dividing by zero or accessing a bad index.
A 'rubber duck' debugging technique means:	**explaining your code line by line (to a duck/person) to spot the flaw yourself** — Explaining code aloud forces you to examine each step, often revealing the bug.
To fix a bug efficiently, first:	**reproduce it consistently** — A reliably reproducible bug can be isolated and fixed; an intermittent one is much harder.
Why is a systematic approach better than random changes when debugging?	**it reliably finds the true cause instead of masking symptoms** — Systematic tracing and isolation find the root cause; random edits often just hide symptoms.
Adding print statements to inspect values is called:	**print debugging (a tracing technique)** — Inserting prints to reveal values at checkpoints is a simple, effective debugging technique.
The best mindset when a program misbehaves is to assume:	**the bug is in your code (the computer did exactly what you wrote)** — Since the computer executes your literal code, the fault is almost always in what you wrote.
Given: x = 5; if x = 3: print('hi'). This likely has a bug because:	**'=' (assignment) was used instead of '==' (comparison)** — Using '=' where '==' is needed is a classic bug — assignment vs equality test.
A LOGIC error is:	**code that runs but produces the wrong result** — A logic error runs without crashing but gives an incorrect result — often the hardest to find.
A debugger tool lets you:	**pause execution and inspect variables step by step** — A debugger steps through code and shows the program's state, aiding fault-finding.
A program that runs but outputs the wrong number has a:	**logic error** — Running but wrong means the logic is flawed — a logic error.
Tracing a loop's variable each iteration helps catch:	**off-by-one and update errors** — Watching the loop variable each pass exposes miscounts and missing updates.
A BUG is:	**an error in a program that makes it behave incorrectly** — A bug is a defect causing a program to produce wrong results or crash.
Writing small tests helps debugging by:	**checking whether each part behaves as expected** — Tests verify each piece's behavior, catching bugs early and localizing them.

To find WHERE a bug is, a useful technique is to:	**print variable values (or use a debugger) at key points** — Printing or inspecting values at checkpoints reveals where the program's state goes wrong.
Why is debugging a core programming skill?	**all non-trivial programs have bugs, and finding them is essential to making software work** — Every real program has defects; the ability to trace, isolate, and fix them is central to programming.
An off-by-one bug in a loop is an example of a:	**logic error (wrong number of iterations)** — Off-by-one runs fine but produces wrong results — a logic error.
Tracing a program by hand helps because it:	**reveals the exact point where the actual state diverges from the expected state** — Hand-tracing exposes where reality diverges from your mental prediction — the bug's location.
If a program crashes with 'index out of range', the likely cause is:	**accessing a list position that doesn't exist** — An out-of-range index accesses a nonexistent list position — a common runtime error.
A SYNTAX error is:	**code that breaks the language's rules (won't run)** — Syntax errors (like a missing bracket) violate the language grammar and prevent the program from running.
DEBUGGING is the process of:	**finding and fixing bugs** — Debugging locates the cause of incorrect behavior and corrects it.
A missing closing bracket usually causes a:	**syntax error (the program won't run)** — A missing bracket violates syntax, so the program fails to compile/run.
TRACING a program means:	**following the code step by step, tracking each variable's value** — Tracing executes the program mentally (or on paper), tracking state line by line.

14. Data and abstraction — modeling a thing as data

✂ Question (fold →)	Answer
Good abstraction makes a program:	**easier to understand and change** — Well-chosen abstractions hide complexity, improving readability and maintainability.
Choosing what NOT to include in a model is part of:	**abstraction** — Deciding which details to omit is a core act of abstraction.
The RIGHT level of abstraction:	**captures what the task needs and hides the rest** — Good abstraction includes exactly what the purpose requires, no more.
A list of student records is an example of:	**combining a data structure (list) with modeled data (records)** — Real programs combine structures (lists, maps) with modeled data to represent complex information.
Good data modeling makes later operations (searching, updating) generally:	**easier and more efficient** — A representation that matches how you'll use the data streamlines operations.
Abstraction and decomposition work together to:	**manage the complexity of large programs** — Decomposition breaks a problem into parts; abstraction hides each part's details — together they tame complexity.
A boolean field 'isActive' on a user record models:	**a two-state property (active or not)** — isActive naturally captures an on/off property as a boolean.
An abstraction 'leaks' when:	**its hidden details unexpectedly affect how you must use it** — A leaky abstraction forces the user to think about details it was supposed to hide.
Choosing to store a phone number as a STRING (not an integer) is wise because:	**it can preserve leading zeros and formatting** — Phone numbers aren't used arithmetically and may have leading zeros/symbols — a string fits.
Representing a TEMPERATURE reading, you would most likely use a:	**number (integer or float)** — A temperature is naturally a numeric value.
Choosing a POOR data representation can make a program:	**harder to write and less efficient** — The right representation makes operations natural and efficient; a poor one complicates everything.
A model that captures the WRONG details for a task is:	**a poor abstraction that makes the task harder** — Modeling the wrong attributes complicates the very operations the program needs.
Representing a person's list of hobbies, you would use a:	**list** — Multiple hobbies form a collection — a list.
A student could be modeled as data with attributes like:	**name, id, and grade** — Relevant attributes (name, id, grade) capture what a program needs about a student.
Deciding what to include and exclude when modeling data is guided by:	**the purpose the model must serve** — The task's purpose determines which attributes are relevant to model.
Why are data modeling and	**all software represents real things as data, and good abstractions make that**

abstraction central to computing?	**tractable** — Every program turns real-world things into data; abstraction is what makes representing and reasoning about them manageable.
A 'model' in computing is:	**a simplified representation of something real, capturing what matters** — A model deliberately simplifies reality to represent the aspects relevant to a purpose.
DATA ABSTRACTION lets a programmer:	**work with data by its meaning, not its low-level bits** — Abstraction lets you think in terms of 'a student' or 'a list', not raw memory.
Representing whether a light is ON or OFF, you would use a:	**boolean** — A two-state on/off is naturally a boolean.
Abstraction lets a team work on different parts by:	**agreeing on interfaces while hiding internal details** — Well-defined interfaces let people build parts independently, hiding internals.
Modeling a real-world thing as DATA means:	**representing its relevant attributes as values a program can use** — You capture a thing's relevant properties as data (numbers, text, booleans) the program manipulates.
Choosing WHICH attributes to represent is an act of:	**abstraction (keeping the relevant, hiding the irrelevant)** — Abstraction selects the details that matter for the task and omits the rest.
To model a deck of cards, a natural representation is a:	**list of card records** — A deck is a collection of cards, each with attributes — a list of records.
A record (or object) groups:	**related fields describing one thing** — A record bundles the related attributes of one entity (e.g. a student's name/id/grade).
Representing a date, a reasonable model is:	**separate fields (or a structured value) for year, month, day** — A date decomposes into year/month/day — modeled as structured numeric fields.

15. Algorithms — combining constructs to solve problems

✂ Question (fold →)	Answer
Testing an algorithm on EDGE cases (empty list, one element) helps ensure it:	**handles all inputs correctly, not just typical ones** — Edge cases expose bugs that typical inputs hide, verifying full correctness.
Given the same problem, an O(n) algorithm is generally preferred over an O(n^2) one because it:	**scales better to large inputs** — Both correct, the O(n) algorithm handles large inputs far better.
Expressing an algorithm clearly (pseudocode/flowchart) BEFORE coding helps because it:	**separates the logic from the syntax, catching flaws early** — Designing the algorithm first lets you get the logic right before wrestling with a language's syntax.
Given an unsorted list, an algorithm to find a specific value should use:	**linear search** — Unsorted data calls for linear search (binary needs sorted).
An algorithm that never terminates for some inputs is:	**incorrect (violates the finiteness requirement)** — Algorithms must be finite; one that can loop forever on some input is not a correct algorithm.
An algorithm must be UNAMBIGUOUS, meaning each step:	**has exactly one clear interpretation** — Every step must be precisely defined so it executes the same way each time.
When two correct algorithms exist, you usually prefer the one that is:	**more efficient (especially for large inputs)** — Given correctness, efficiency (and clarity) guide the choice, particularly at scale.
Writing an algorithm in pseudocode first lets you:	**focus on the logic before dealing with syntax** — Pseudocode separates the algorithm's logic from any language's syntax details.
The three questions to ask about any algorithm are: is it correct, does it terminate, and:	**how efficient is it?** — Correctness, termination, and efficiency are the core criteria for judging an algorithm.
Two different algorithms that solve the same problem can differ in:	**efficiency (speed and memory)** — Correct algorithms give the same answer but can differ greatly in how efficiently they get there.
Decomposing a problem before writing an algorithm means:	**breaking it into smaller subproblems** — Decomposition splits a complex problem into manageable subproblems, each easier to solve.
An algorithm's CORRECTNESS means it:	**produces the right output for all valid inputs** — Correctness is about producing the right answer for every valid input — separate from efficiency.
To find the largest number in a list, an algorithm can:	**track the biggest seen so far while scanning the list once** — Keep a running maximum, updating it whenever a larger element is found — one linear pass.
An algorithm's efficiency is measured by how its work grows with:	**input size** — Efficiency (complexity) describes how work scales with the size of the input.
An ALGORITHM combines constructs like sequence, selection, and iteration to:	**solve a problem in a finite number of well-defined steps** — An algorithm is a finite sequence of clear steps (using the basic constructs) that solves a problem.

An algorithm to check if a number is even can test whether:	**the number divided by 2 has remainder 0** — A number is even when n mod 2 == 0 (no remainder).
The steps to compute an average of a list are:	**sum all elements, then divide by the count** — Accumulate the sum in one pass, then divide by the number of elements.
Breaking 'make breakfast' into 'toast bread', 'fry egg', 'pour juice' is:	**decomposition** — Splitting a task into concrete subtasks is decomposition.
A flowchart represents an algorithm using:	**boxes and arrows showing the flow of steps** — Flowcharts diagram an algorithm's steps and decision flow with shapes and arrows.
Combining a loop with a conditional lets an algorithm:	**make a decision on each item as it iterates** — Iterating with an embedded conditional processes each element and branches per item.
A good algorithm should be:	**correct, finite, and well-defined** — Algorithms must give correct results, terminate, and have unambiguous steps.
PSEUDOCODE is:	**an informal, language-independent description of an algorithm** — Pseudocode expresses an algorithm's logic in plain, structured terms without a specific language's syntax.
To count how many even numbers are in a list, an algorithm should:	**loop through and increment a counter when a number is even** — Iterate, testing each number's evenness, and count the matches.
Why is algorithmic thinking the heart of computer science?	**designing correct, efficient step-by-step solutions is what programming ultimately is** — At its core, computer science is about devising algorithms — correct, efficient procedures — to solve problems.
An algorithm to reverse a list could:	**swap elements from the ends toward the middle** — Swapping the outermost pair and moving inward reverses the list in place.

16. Capstone — decompose, build, and debug a real program

✂ Question (fold →)	Answer
Refactoring code means:	**improving its structure without changing its behavior** — Refactoring cleans up code (clarity, reuse) while preserving its behavior.
Choosing the right DATA STRUCTURE (list, map) for a task affects:	**how easy and efficient the program is to write** — Matching the data structure to the access pattern (position vs key) makes the program natural and efficient.
DECOMPOSITION helps program-building by:	**turning a big task into smaller, solvable pieces** — Decomposition makes a large program tractable by splitting it into manageable functions/modules.
When a program gives a wrong result, you should:	**debug it by tracing to find where the state goes wrong** — Tracing locates where actual behavior diverges from expected, guiding the fix.
The empowering idea of computer science is that:	**with a few constructs you can build tools that solve real problems** — A small set of constructs, combined thoughtfully, lets anyone build software that solves real problems.
An efficient algorithm matters in a real program when:	**the input can be large** — For large inputs, algorithm choice determines whether the program is usable — a real design concern.
Testing your finished program with unexpected input checks its:	**robustness (handling edge cases gracefully)** — Feeding unusual inputs verifies the program handles edge cases without breaking.
The single biggest habit CodeForge builds is to:	**decompose, predict, run, and debug — reasoning about code before and after running it** — Computational thinking — decompose, predict, run, debug, reflect — is what turns coding into genuine problem-solving.
After writing code, you should __ before assuming it works:	**test it (including edge cases)** — Testing — especially edge cases — verifies the program actually behaves correctly.
Combining sequencing, conditionals, loops, and functions lets you:	**build programs that solve complex, realistic tasks** — The basic constructs, combined, are enough to express any computation a real task needs.
Combining everything — decomposition, data structures, algorithms, and debugging — is what it takes to:	**build a complete, working program** — Real programs integrate all the core skills together into a working whole.
If your program's output is close but slightly off, suspect a:	**logic error like an off-by-one or wrong operator** — Running-but-slightly-wrong points to a logic error, often off-by-one or a wrong operator.
Before coding a program, sketching the algorithm (pseudocode/flowchart) helps you:	**get the logic right before syntax gets in the way** — Designing the algorithm first separates logic from syntax and catches flaws early.
The habit of predicting a program's output before running it builds:	**a mental model of how code executes (the notional machine)** — Predicting execution strengthens your mental model of how the computer runs code.
Good comments and clear names in a program help:	**you and others understand and maintain the code later** — Readable code (names, comments) is easier to understand, debug, and extend over time.

A program that reads input, processes it, and produces output follows the:	**input–process–output pattern** — Most programs take input, transform it, and produce output — a fundamental structure.
Testing edge cases in a finished program (empty input, extremes) ensures it:	**works for ALL inputs, not just the common ones** — Edge cases catch bugs that typical inputs miss, giving confidence the program is robust.
Why is learning to build real programs the goal of computer science education?	**programming lets you turn ideas into working tools that solve real problems** — Building programs is the payoff of CS: transforming problem-solving ideas into real, working software.
Reflecting after building a program (what worked, what didn't) helps you:	**improve as a programmer over time** — Reflecting on the process turns each project into learning that improves future work.
A program that reads two numbers and prints their sum uses:	**input, a computation, and output** — It follows input → process (add) → output — the fundamental pattern.
Building a program incrementally (a piece at a time, testing each) is better than:	**writing it all at once and testing only at the end** — Building and testing in small pieces localizes bugs and keeps progress verifiable.
The 'predict–run–debug' loop means you:	**predict what the code should do, run it, and fix any mismatch** — Predicting first makes a surprising result a signal to debug — the core computational-thinking loop.
The FIRST step in building a program to solve a problem is usually to:	**understand the problem and decompose it into smaller parts** — Understanding and breaking the problem into subproblems precedes writing any code.
Writing each subproblem as a FUNCTION helps because it:	**isolates and reuses logic, making the program easier to test** — Functions encapsulate each subproblem, so you can build, test, and reuse each independently.
Choosing a list vs a map for your program depends on whether you access data by:	**position (list) or by a meaningful key (map)** — Access-by-position favors a list; access-by-label favors a map.

Keep going

You practiced **400 cards** from CodeForge. Come back to the ones you missed — spaced-out practice makes them stick.

Spark & Anvil is a 501(c)(3) public charity. Every app, story, and book is free, forever — no ads, no in-app purchases, no tracking.

Keep exploring online

Play it now	More free books
	
`https://spark-and-anvil.org/play/codeforge`	`https://spark-and-anvil.org/books`

Scan a code with any phone camera, or type the address. Every app, story, and book is free, forever — no account, no tracking.