PuzzleForge — Flashcards

A Spark & Anvil printable flashcard deck. Quiz yourself on the PuzzleForge 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. recursion — solve the smaller tower first (Tower of Hanoi)

✂ Question (fold →)	Answer
Nestor says: "A tower of 10 is just a tower of 9 with one more disk underneath." This helps because:	**if you can move a tower of 9, you can move a tower of 10** — That's the recursive idea: solving size n only needs the solution to size n-1 (twice) plus one big-disk move.
A tower of 4 disks needs 15 moves. If you make a mistake and use 20 moves but still finish, your solution is:	**correct but not the fewest-moves (optimal) solution** — Finishing legally is a valid solution, but 15 is the minimum — 20 moves works yet isn't optimal. (Minim would chase the 15!)
To move a tower of 3 disks, Nestor first moves the top TWO disks out of the way. This is an example of:	**solving a smaller version of the same problem first** — This is recursion / decomposition: to move n disks you first solve the smaller (n-1)-disk tower — the same problem, one size smaller.
The move-counts go 1, 3, 7, 15, … Each number is:	**double the one before, plus one** — Each count = 2 x (previous) + 1, because you move the smaller tower twice and the big disk once. That gives 1, 3, 7, 15, 31, …
The smallest disk in a Hanoi solve tends to move:	**very often — on every other move** — In an optimal solve the smallest disk moves on every other move — it's the busiest disk on the board.
What is the fewest number of moves to solve a tower of 3 disks?	**7** — 3 disks: move the top 2 to the spare peg (3 moves) + move the biggest to the goal (1) + move the 2 onto it (3) = 7 moves.
A tower of 3 takes 7 moves. Using the pattern (move the smaller tower, the big disk, then the smaller tower again), how many moves for 4 disks?	**15** — 4 disks = (moves for 3) + 1 + (moves for 3) = 7 + 1 + 7 = 15 moves.
What is the fewest number of moves to solve a tower of 2 disks?	**3** — 2 disks: move the small one aside (1), move the big one to the goal (2), move the small one on top (3) = 3 moves.
After you move the top 3 disks aside and move the biggest disk to the goal, what is the LAST big step?	**move the 3-disk tower on top of the biggest disk** — The three steps are: move (n-1) aside, move the biggest to the goal, then move the (n-1) tower onto the biggest — finishing the solve.
How many moves does it take to move a tower of just 1 disk to another peg?	**1** — A single disk takes exactly 1 move.
In the Tower of Hanoi, how many disks may you move at one time?	**exactly one** — Rule 1 of Hanoi: you move exactly one disk per move — always the top disk of a peg.
Which sequence correctly solves a 2-disk tower from peg A to peg C (B is spare)? Disks: 1=small, 2=big.	**disk 1 A→B, disk 2 A→C, disk 1 B→C** — Small to spare (A→B), big to goal (A→C), small onto big (B→C) — 3 legal moves, never big-on-small.
"Decomposition" in problem-solving means:	**breaking a problem into smaller parts you can solve** — Decomposition is splitting a big problem into smaller, easier sub-problems — exactly how Hanoi breaks into smaller towers.
Why is planning your strategy (before moving) helpful in Hanoi?	**it lets you follow the smaller-tower plan instead of guessing** — Naming the plan (solve n-1, move the big disk, solve n-1) keeps every move purposeful and reaches the fewest-moves solution.

The big idea of recursion is best described as:	**solving a big problem by solving smaller copies of the same problem** — Recursion means breaking a problem into smaller copies of itself until the smallest copy is trivial to solve.
How many pegs does the classic Tower of Hanoi use?	**three** — The classic puzzle has three pegs: the disks start on one, end on another, and the third is a helper.
If a tower of 6 disks takes 63 moves, how many for 7 disks?	**127** — 7 disks = 2 x 63 + 1 = 127 moves (also 2^7 - 1 = 128 - 1 = 127).
Which move is NOT allowed in the Tower of Hanoi?	**placing a larger disk on top of a smaller disk** — Rule 2: a larger disk may never rest on a smaller one. Smaller-on-larger and moving to an empty peg are both fine.
In a Hanoi solve, the spare (middle) peg is used to:	**hold disks temporarily while a bigger disk moves** — The spare peg is a temporary holding place for the smaller tower while the big disk moves to the goal.
The fewest moves for n disks equals 2 multiplied by itself n times, minus 1 (written 2^n - 1). For 3 disks this is:	**2 x 2 x 2 - 1 = 7** — 2^3 = 2 x 2 x 2 = 8; 8 - 1 = 7, matching the 3-disk answer.
To move 4 disks from the start peg to the goal peg, what do you do FIRST?	**move the top 3 disks to the spare peg** — You must clear the top 3 disks onto the spare peg before the biggest disk is free to move to the goal — then bring the 3 back on top.
How many moves for a tower of 5 disks?	**31** — 5 disks = 2 x 15 + 1 = 31 moves.
Why can't the biggest disk move until the disks above it are gone?	**you may only move the top disk of a peg** — Because you can only lift the TOP disk, the biggest disk (at the bottom) is stuck until everything above it is moved away.
The Tower of Hanoi teaches recursion because the SAME three-step plan works for:	**any number of disks** — The recursive plan (solve the smaller tower, move the big disk, solve the smaller tower again) works for any n — that generality is the power of recursion.
Which disk on a peg are you allowed to pick up?	**the top (smallest on that peg) disk** — You may only move the top disk of a peg — the ones below it are blocked.

2. optimality — the fewest moves (2^n - 1) and why it is the minimum

✂ Question (fold →)	Answer
If a solution is optimal, how many OTHER solutions use even fewer moves?	**zero — none can be shorter** — Optimal is the minimum, so by definition no legal solution uses fewer moves.
The powers of 2 go 2, 4, 8, 16, 32, ... What comes after 32?	**64** — Each power of 2 doubles: 32 x 2 = 64.
A tower of 5 disks takes 31 moves. How many for 6 disks?	**63** — 6 disks = 2 x 31 + 1 = 63 (also 2^6 - 1 = 64 - 1 = 63).
A legend says monks move a 64-disk tower. Using 2^n - 1, that is about:	**a truly enormous number (over 18 quintillion moves)** — 2^64 - 1 is over 18 quintillion — the doubling makes the count explode, which is why the legend says the world would end first.
Which is the smallest number that is a power of 2?	**2^1 = 2** — 2^1 = 2 is the first power of 2 in the doubling list 2, 4, 8, 16, 32...
Big idea: optimality asks not just "did I solve it?" but also:	**"could I have solved it in fewer moves?"** — Optimality is about efficiency: after solving, ask whether the same result could be reached with fewer moves.
Why is counting moves a good way to compare two puzzle strategies?	**fewer moves means a more efficient strategy** — Move-count is a clean measure of efficiency: the strategy that reaches the goal in fewer moves is more efficient.
What does it mean for a puzzle solution to be OPTIMAL?	**it uses the fewest possible moves** — An optimal solution reaches the goal legally in the fewest moves possible — no solution can do it in fewer.
The fewest-moves counts for towers of 1, 2, 3, 4 disks are:	**1, 3, 7, 15** — The optimal counts are 1, 3, 7, 15 — each is 2 times the previous plus 1.
The biggest disk in an optimal Hanoi solve moves exactly:	**one time** — In the fewest-moves solution the biggest disk goes straight from source to goal — exactly one move.
Two players both solve a 4-disk tower legally. Ana uses 15 moves, Ben uses 17. Who was optimal?	**Ana — 15 is the fewest for 4 disks** — The optimal count for 4 disks is 15, so Ana hit the minimum; Ben's 17 is legal but not optimal.
A tower of 8 disks: how many optimal moves?	**255** — 2^8 = 256; 256 - 1 = 255 moves.
2^7 - 1 equals how many moves (for 7 disks)?	**127** — 2^7 = 128; 128 - 1 = 127 moves.
Adding ONE more disk to a tower does what to the fewest-moves count?	**a little more than doubles it (doubles and adds one)** — Each extra disk roughly doubles the work: the count goes from M(n-1) to 2 x M(n-1) + 1.
Minim's motto is "never a wasted move." Which habit best matches it?	**plan the fewest-moves route before touching a disk** — Planning the recursive fewest-moves route (solve n-1, big disk, solve n-1) avoids wasted moves — Minim's motto in action.
The optimal move-count 2^n - 1	**odd** — A power of 2 is even, and even minus 1 is odd — so 1, 3, 7, 15, 31... are all

is always what kind of number?	odd.
A tower of 10 disks takes how many optimal moves (2^10 - 1)?	**1023** — 2^10 = 1024; 1024 - 1 = 1023 moves.
Minim solves a 3-disk tower in 9 moves. What can she say about her solution?	**it is legal but not optimal — 7 is the fewest** — 9 moves finishes legally, but the minimum is 7, so it is not optimal — Minim would trim it to 7.
Why can't a tower of 3 be solved in fewer than 7 moves?	**you must clear the top 2 (3 moves), move the big disk (1), then rebuild the 2 (3 moves)** — The 2-disk tower must be cleared (3 moves) and rebuilt (3 moves) around the single big-disk move — 3 + 1 + 3 = 7, so 7 is forced.
The rule 2^n - 1 means: multiply 2 by itself n times, then subtract 1. For n = 4 disks this is:	**2 x 2 x 2 x 2 - 1 = 15** — 2^4 = 16; 16 - 1 = 15, which matches the 4-disk count.
To move the BIGGEST disk to the goal, what must be true first?	**every smaller disk is off it and on the spare peg** — The biggest disk can only move once all smaller disks are cleared onto the spare peg, leaving the goal peg open for it.
Doubling the DISKS (from 3 to 6) changes the moves from 7 to 63. So doubling the disks:	**far more than doubles the moves** — Because every extra disk doubles the work, three extra disks multiply the count ~8x (7 -> 63), far more than doubling.
Why is each move-count double the previous one plus one?	**you move the smaller tower twice and the big disk once** — Solving n disks = move (n-1) aside + move the big disk + move (n-1) back = 2 x (count for n-1) + 1.
Which is the BEST evidence that a Hanoi solution is optimal?	**its move-count equals 2^n - 1 for that many disks** — The formula 2^n - 1 gives the true minimum, so a solution matching it exactly is optimal.
Which statement about the biggest disk is the KEY reason 7 moves is forced for 3 disks?	**the big disk needs the goal peg clear, so the 2 small disks must fully leave and return** — The big disk's single move requires the 2-disk tower to fully clear (3 moves) and rebuild (3 moves), forcing at least 7 total.

3. sliding-tile (15-puzzle) — blank-adjacent moves and when a scramble is solvable

✂ Question (fold →)	Answer
The blank's job in the puzzle is best described as:	**the empty seat every tile takes turns sliding into** — The blank is the moving gap; every slide is a tile taking the blank's seat, so tracking the blank tracks the whole puzzle.
In a sliding-tile puzzle, what actually lets a tile move?	**an empty (blank) space next to it** — Tiles slide into the single blank space, so only a tile next to the blank can move.
In the row 2, 1, 3, how many inversions are there?	**1** — Only 2 comes before 1 (out of order); 3 is after both, so there is exactly 1 inversion.
Which pair of ideas do Wend and Evan together teach on the sliding puzzle?	**explore the reachable states (search) and test if the goal is reachable (parity)** — Wend maps reachable states (search) and Evan checks the parity invariant (is the goal reachable at all) — the two halves of solving a sliding puzzle wisely.
Legal slides change the number of inversions by an even amount on an odd-width board. That's why:	**the even-or-odd nature of the inversion count stays fixed** — Because slides shift inversions by an even amount, the count's even/odd nature never flips under legal play — that fixed value decides solvability.
Evan checks solvability WITHOUT solving. This is powerful because:	**you can know a puzzle is impossible before wasting time trying** — Solvability is an invariant you can test up front, so Evan knows whether a scramble can ever work before making a single move.
A common strategy for the 15-puzzle is to solve the top row and left column first, then:	**shrink the problem to a smaller puzzle below-right** — Solving and 'locking' the top row and left column reduces the puzzle to a smaller grid — a decomposition strategy.
Wend 'maps out reachable positions.' The set of all boards you can slide to is called the:	**state space** — The state space is the collection of every arrangement reachable by legal moves — Wend's map of where you can go.
You scramble a puzzle only by legal slides from the solved board. The result is:	**always solvable — you can slide back** — If you only used legal slides, you can reverse them all to return to solved — so any board reached that way is solvable.
If you physically PULL OUT two tiles and swap them, the puzzle's solvability:	**flips — a solvable board becomes unsolvable and vice versa** — Swapping two tiles changes the parity of the arrangement, flipping solvability — the classic prank that makes a puzzle unsolvable.
Which is the SAFEST first step when facing a hard scramble, in Wend's style?	**figure out which nearby positions you can reach and where they lead** — Wend surveys reachable positions before committing — mapping the state space keeps the search purposeful.
When the blank is in a corner, how many moves are possible?	**two** — A corner blank has only two neighbors (along the two edges), so only two tiles can slide in.
Moving a tile into the blank does what to the blank?	**the blank moves to where that tile was** — Each move swaps a tile with the blank, so the blank travels to the spot the tile left.
	if you can reach a state, you can always get back — so you can explore

Why does reversibility matter for solving?	**safely** — Because every move can be undone, exploring is safe — you can try a path and reverse it, which is why search works here.
A single legal slide moves the blank one step. Sliding the blank left then right returns the board to:	**exactly where it started** — Every move is reversible; sliding the blank one way then back returns the exact starting board.
How many tiles are next to (can slide into) a blank in the middle of the board?	**up to four** — A blank in the middle touches tiles above, below, left, and right — up to four possible moves.
The goal state of a solved sliding puzzle has how many inversions?	**zero** — In the solved board every number is in order, so there are zero inversions.
In the row 3, 2, 1, how many inversions are there?	**3** — The out-of-order pairs are (3,2), (3,1), and (2,1) = 3 inversions.
Two puzzles differ only by swapping tiles 14 and 15. One is solvable. The other is:	**unsolvable** — Swapping exactly two tiles flips solvability, so if one board is solvable the swapped board must be unsolvable.
Evan says some scrambled boards can NEVER be solved. This means:	**no sequence of legal slides will ever reach the goal** — Exactly half of all tile arrangements are unreachable from the goal by legal slides — no amount of sliding will solve those.
A 4-by-4 sliding puzzle has 15 numbered tiles and how many blank spaces?	**one** — A 4x4 board has 16 cells; 15 hold tiles and exactly 1 is blank.
An "inversion" is a pair of tiles where:	**a larger number comes before a smaller one in reading order** — An inversion is any pair where a bigger number appears before a smaller one in reading order — counting them tells you about solvability.
Big idea: on a sliding puzzle, whether a scramble can be solved depends on:	**the arrangement's fixed even/odd property, not how hard you try** — Solvability is set by the arrangement's invariant parity — a fixed property the starting scramble has, independent of effort.
For the 15-puzzle, what fraction of all possible tile arrangements is actually solvable?	**one half** — Exactly half of all arrangements are solvable; the other half form a separate, unreachable group.
Why is it useful to know a puzzle is UNSOLVABLE before trying?	**you avoid wasting effort on something that can never work** — Knowing impossibility up front saves effort — you re-set the board (a legal reset) instead of chasing a solution that cannot exist.

4. state-space search — states, moves, and which positions are reachable

✂ Question (fold →)	Answer
Big idea: turning a puzzle into states-and-moves lets you:	**plan a route to the goal instead of guessing move by move** — Modeling a puzzle as a state space turns blind guessing into route-planning toward the goal.
In puzzle-solving, a 'state' means:	**one complete arrangement of the puzzle at a moment** — A state is a full snapshot of the puzzle — where every piece is right now.
A 'path' from start to goal is:	**an ordered list of moves connecting them** — A path is the ordered sequence of moves that leads from the start state to the goal state.
If two different paths both reach the goal, you should usually prefer:	**the one with fewer moves** — When multiple paths reach the goal, the shorter (fewer-move) one is more efficient.
'Backtracking' means:	**undoing recent moves to try a different path** — Backtracking is reversing recent moves to a decision point so you can explore an untried path.
Why is it helpful that Hanoi and sliding puzzles have REVERSIBLE moves during search?	**you can always undo a move and try another, so exploring is safe** — Reversible moves make backtracking possible — you can safely explore because any step can be undone.
'Searching' a puzzle means:	**following moves through states to find a path to the goal** — Search is tracing a path of moves from the start state to the goal state through the state space.
Two states are 'neighbors' when:	**one legal move turns one into the other** — Neighbors are states connected by exactly one legal move — one step apart in the state space.
Thinking of a puzzle as states-and-moves helps because:	**it turns 'what do I do?' into 'which neighbor gets me closer?'** — The state-space view reframes solving as repeatedly choosing the neighbor that moves you toward the goal.
Wend's core habit is best summarized as:	**map where you can go before deciding where to step** — Wend surveys the reachable states first — mapping before moving is state-space search in action.
Why do puzzle designers love reversible, rule-bound puzzles for teaching search?	**the clean states and moves make the search map easy to reason about** — Well-defined states and reversible moves make a clean, explorable state space — ideal for learning how search works.
In a river-crossing puzzle, a 'state' would record:	**who is on each bank right now** — A river-crossing state records exactly who is on each bank (and where the boat is) — the full current situation.
Because there can be so many states, a good searcher tries to:	**follow moves that get closer to the goal, not every move blindly** — With huge state spaces, guiding the search toward the goal is far more efficient than blindly checking every state.
If the goal state is NOT reachable from the start, the puzzle is:	**unsolvable from that start** — If no legal path connects start to goal, the goal is unreachable and the puzzle can't be solved from there.
Some neighbor states in a river-crossing are ILLEGAL. A good	**never steps into an illegal state** — Search prunes illegal states — they can't be

search:	part of any valid solution, so you never step into them.
Wend explores neighbors before committing. Why explore first?	**to see which moves lead toward the goal and which are dead ends** — Exploring neighbors first reveals which moves progress toward the goal, avoiding blind moves into dead ends.
A puzzle with only ONE reachable state besides the start (and it's not the goal) is:	**stuck — unsolvable from there** — If the goal isn't among the reachable states, there's no path to it — the puzzle is unsolvable from that start.
A 'dead end' in a state-space search is a state where:	**no path forward leads to the goal** — A dead end is a state from which no continuation reaches the goal, so you backtrack and try a different branch.
The SHORTEST path in a state space is the one with:	**the fewest moves** — The shortest path uses the fewest moves — the search-map version of an optimal solution.
The whole map of states connected by moves is called the:	**state space** — The state space is the network of all states joined by legal moves — the map Wend explores.
Which question captures state-space thinking best?	**"From here, which reachable state is closest to the goal?"** — State-space thinking constantly asks which reachable neighbor moves you closest to the goal.
A state that has already been visited during a search should usually be:	**skipped, so you don't loop forever** — Marking visited states prevents loops — re-exploring the same state wastes effort and can loop forever.
The number of possible states in a puzzle tends to be:	**very large, growing fast with more pieces** — State counts explode with piece count (the 15-puzzle has over 10 trillion states), which is why smart search beats brute force.
A 'reachable' state is one that:	**you can get to from the start using legal moves** — A reachable state has at least one legal path from the start state to it.
Marking visited states as you search is like:	**leaving breadcrumbs so you don't retrace your steps** — Marking visited states is like breadcrumbs — it stops you from re-exploring the same places and looping.

5. look-ahead planning — clear a path by thinking several moves ahead (sliding-block / Rush Hour)

✂ Question (fold →)	Answer
Look-ahead planning is useful far beyond Rush Hour, for example in:	**packing a bag or scheduling steps that depend on each other** — Look-ahead applies anywhere steps depend on each other — packing, cooking, scheduling — do the enabling steps first.
Which best matches Faro's planning habit?	**trace the exit lane, list the blockers, then order their moves** — Faro traces the target's path, lists what blocks it, and sequences the clearing moves before touching a car.
A move that doesn't help clear the target's path is called a:	**wasted (or unnecessary) move** — A move that doesn't advance the clearance plan is wasted — planning ahead avoids these.
If clearing the path needs moves in the order C, B, A but you do A first:	**A may get stuck because B and C haven't made room yet** — When moves depend on each other, doing them out of order traps pieces — you must make the enabling moves first.
In a Rush Hour sliding-block puzzle, the goal is to:	**slide the target car out through the exit** — The goal is to slide the target car out of the exit; other cars are obstacles to move aside.
Look-ahead and Wend's search are related because both:	**explore future states before committing to moves** — Look-ahead is mental state-space search — the same idea as Wend's mapping, done a few moves deep before acting.
The deepest 'chain of blockers' tells you:	**how many cars must move before the target can escape** — The chain of blockers (each stuck behind the next) shows how many cars must move first — the depth of your plan.
The main difference between guessing and planning is:	**planning predicts the result of moves before making them** — Planning predicts outcomes before acting; guessing acts and hopes. Prediction is what look-ahead adds.
To free a blocker, you often need to move it into:	**an empty space along its lane** — A blocker can only move into open space along its lane, so open squares are the resource your plan must use.
A blocking car can't move because ANOTHER car is in ITS way. This means you must:	**move the second car first, then the blocker, then the target** — You order moves so each clears the way for the next: free the blocker's blocker first, then the blocker, then the target.
If a blocker can move EITHER left or right to clear the path, you should pick the direction that:	**doesn't create a new block for another needed move** — When a blocker can go two ways, look ahead and choose the direction that doesn't block a later move you need.
Faro clears a path 'before touching a piece.' The benefit is:	**the plan is checked in her head, avoiding trapped positions** — Planning the whole clearance in her head lets Faro spot traps before committing, so her real moves flow.
Big idea: look-ahead turns a sliding-block puzzle from 'push and hope' into:	**'clear the path in the right order, then the target just rolls out'** — Look-ahead reframes the puzzle as ordering the clearing moves; once the path is clear, the target simply exits.
'Looking ahead' means:	**imagining several future moves before you make the first one** — Looking ahead is mentally simulating the next several moves so your first move fits a

	working plan.
Faro says 'I move the car in my mind first.' This mental step is called:	**simulation — imagining the move's effect** — Imagining a move's effect before making it is simulation — the heart of looking ahead.
Why is moving a car WITHOUT looking ahead risky?	**it might block a car you need to move later** — Without planning, a move can accidentally block a car you'll need later, forcing wasted undo moves.
Why do sliding-block puzzles get HARD even on a small board?	**blockers chain together, so many moves must happen in a careful order** — Even small boards are hard because blockers interlock, forcing long, carefully-ordered sequences of moves.
Cars in Rush Hour can move:	**only forward or backward along their own lane** — Each car slides only along its own row or column, never turning or jumping.
Planning ahead in Rush Hour is a form of:	**searching future states in your head** — Looking ahead is mentally searching future states — the same state-space idea, done in your head before moving.
In planning, a 'dependency' between moves means:	**one move can't happen until another makes room for it** — A dependency means one move requires another first — respecting dependencies is what correct ordering is about.
A puzzle where the target has THREE blockers, each needing one clearing move, needs at least:	**four moves (three clears + the target)** — Three blockers each need one clearing move (3), plus the target's own escape move (1) = at least 4 moves.
A good habit before your FIRST move is to:	**picture the last move (target escaping) and work backward to what must clear** — Working backward from the target's escape reveals exactly what must be cleared — a planning shortcut Faro uses.
The target car can't reach the exit. The FIRST thing to figure out is:	**which cars are blocking its path** — Planning starts by identifying the cars blocking the target's path to the exit.
Which is the BEST sign your plan is efficient (Minim + Faro together)?	**every move you make advances the clearance — none are undone** — An efficient plan has every move advancing the clearance with no undo — planning (Faro) meets optimality (Minim).
If you plan ahead and STILL get stuck, the smartest response is to:	**backtrack and try a different clearing order** — If a plan stalls, backtrack to a decision point and try a different order of clearing moves.

6. constraint satisfaction — never break the rule (wolf-goat-cabbage; missionaries-cannibals)

✂ Question (fold →)	Answer
Big idea: constraint satisfaction is solving a puzzle where:	**the challenge is staying legal at every step, not just reaching the goal** — Constraint satisfaction is about reaching the goal while every single step obeys the rules — staying legal is the real challenge.
A good general rule for constraint puzzles is:	**handle the most-restricted piece first** — Tackling the most-constrained element first (like the goat) is a classic constraint-satisfaction heuristic.
Which pair must NEVER be left alone together without the farmer?	**the wolf and the goat (and the goat and the cabbage)** — The forbidden pairs are wolf+goat and goat+cabbage; the wolf and cabbage are safe together.
If a move would create an illegal state, Skiff:	**rejects that move and looks for another legal one** — An illegal move is never made, even momentarily — Skiff looks for a legal alternative instead.
After the farmer drops the goat and rows back EMPTY, the boat is:	**ready for the next item, both banks still legal** — Rowing back empty resets the boat for the next crossing while both banks remain in legal states.
Skiff's core habit is:	**check that EVERY move lands in a legal state before making it** — Skiff checks that a move leaves a legal state (no forbidden pair unattended) before committing to it.
In missionaries-and-cannibals, the rule is that on either bank cannibals must never:	**outnumber the missionaries there** — The constraint: on neither bank may cannibals outnumber the missionaries present (unless no missionaries are there).
Which mindset does Skiff model best?	**patience — check the rule on every single move** — Skiff models patient, rule-first thinking: check every move against the constraints before ferrying.
The goat is the 'troublemaker' because:	**it forms a forbidden pair with BOTH other items** — The goat is in both forbidden pairs (wolf+goat, goat+cabbage), so it must be kept apart from the others — that's why it's shuttled.
The full wolf-goat-cabbage solution takes how many crossings (one-way trips)?	**seven** — The classic solution uses seven one-way crossings, including the clever move of bringing the goat back.
A state that breaks a constraint (like wolf alone with goat) is called:	**an illegal state** — A state that violates a constraint is illegal; a valid solution never passes through one.
Why is an EMPTY return trip a legal, useful move?	**the farmer must reposition the boat, and an empty boat breaks no rule** — The farmer needs the boat back on the other bank; carrying nothing keeps every state legal, so the empty trip is valid and necessary.
A 'constraint' in a puzzle is:	**a rule that must never be broken** — A constraint is a rule that every state must satisfy — it can never be violated, even for one moment.
Recognizing the goat is in two forbidden pairs helps you:	**plan to keep the goat with the farmer most of the time** — Spotting the most-constrained piece (the goat) guides the plan: keep it near the farmer and shuttle it as needed.

In the wolf-goat-cabbage puzzle, the farmer's boat can carry:	**the farmer plus one item** — The boat holds the farmer and at most one item, so only one item crosses per trip.
The deepest lesson of river-crossing puzzles is:	**sometimes you must go backward to make legal forward progress** — The key insight is that a temporary backward move (returning the goat) can be the only way to keep advancing legally.
Constraint thinking shows up in real life when you:	**schedule tasks that can't overlap (like one oven, many dishes)** — Any limited resource is a constraint; planning around it (one oven, many dishes) is exactly constraint satisfaction.
Why does having FEWER legal moves sometimes make a puzzle EASIER to plan?	**constraints narrow the choices, so the right move stands out** — Tight constraints prune most moves, so the few legal options make the correct next step easier to spot.
If someone claims a river puzzle is impossible, the honest check is:	**is there ANY path of legal states from start to goal?** — Solvability depends on whether a legal path exists at all, not on effort — the same reachability idea as Wend and Evan use.
'Pruning' illegal states means:	**removing rule-breaking states from consideration** — Pruning discards illegal states so the search only explores valid ones, shrinking the work.
Constraint puzzles are solved by:	**finding a path of legal states from start to goal** — You find a path where every intermediate state is legal — constraints prune the illegal states from the search.
Constraint satisfaction and Wend's state-space search are connected because:	**constraints prune the state space to only legal states** — Constraints shrink the state space by removing illegal states, so search only explores the legal part of the map.
The classic first move is to take the GOAT across because:	**the wolf and cabbage are safe left together** — Taking the goat first leaves the wolf and cabbage, which is a legal (safe) state, so it's the correct opening.
The clever trick in this puzzle is that the farmer sometimes:	**brings an item BACK to avoid breaking a rule** — To stay legal, the farmer sometimes ferries the goat back — a temporary 'backward' move that keeps every state safe.
Which is a legal state in wolf-goat-cabbage (farmer on the near bank)?	**wolf and cabbage on the near bank, goat across** — Wolf+cabbage together (goat safely across) breaks no rule — a legal state. The others leave a forbidden pair unattended.

7. GCD / number reasoning — measure exact amounts by pouring between two jugs

✂ Question (fold →)	Answer
Why can pouring never produce an amount that is NOT a multiple of the GCD?	**every fill, empty, and pour keeps amounts as combinations of the jug sizes, which are all multiples of the GCD** — Every amount is a sum/difference of the jug capacities; since both are multiples of the GCD, so is every reachable amount.
With a 4-liter and a 6-liter jug, which amounts CAN you measure?	**2, 4, and 6 liters (the even amounts)** — The measurable amounts are the multiples of 2 up to the largest jug: 2, 4, and 6.
To get 4 liters with 3- and 5-liter jugs, one path is: fill the 5, pour into the 3 (leaves 2 in the 5), empty the 3, pour the 2 in, then:	**fill the 5 again and top off the 3 (needs 1), leaving 4 in the 5** — With 2 in the 3-jug, refilling the 5 and topping off the 3 (adds 1) leaves exactly 5-1 = 4 liters in the 5-jug.
GCD stands for:	**greatest common divisor** — GCD is the greatest common divisor — the largest number that divides both amounts evenly.
Which statement is the CLEANEST test of measurability for target t with jugs a and b?	**t is reachable if t is a multiple of gcd(a,b) and t is at most max(a,b)** — A target is reachable exactly when it's a multiple of gcd(a,b) and no larger than the biggest jug.
With coprime 7- and 11-liter jugs, measuring 1 liter is:	**possible (gcd is 1)** — Since gcd(7,11)=1, the smallest measurable amount is 1 liter — it's possible (though it takes several pours).
Two jug sizes whose GCD is 1 are called:	**relatively prime (coprime)** — When gcd = 1 the sizes are relatively prime (coprime), and every whole amount up to the larger jug is measurable.
With 6- and 9-liter jugs (gcd 3), which amount is IMPOSSIBLE?	**4 liters** — Measurable amounts are multiples of gcd=3 (3, 6, 9); 4 is not a multiple of 3, so it's impossible.
The GCD of 7 and 11 is:	**1** — 7 and 11 are both prime and different, so their only common divisor is 1 — they're coprime.
A quick way to find the GCD is Euclid's idea: keep replacing the bigger number with:	**the remainder when the bigger is divided by the smaller** — Euclid's algorithm replaces the larger number with the remainder repeatedly; the last nonzero value is the GCD.
The GCD of 8 and 12 is:	**4** — Common divisors of 8 and 12 are 1, 2, 4; the greatest is 4.
Water-pouring puzzles connect to which everyday skill?	**combining fixed measuring cups to reach an unmarked amount** — Using two fixed cup sizes to reach a target amount is exactly the pouring puzzle — GCD decides which totals are possible.
In the famous puzzle, you must measure 4 liters using a 3-liter and a 5-liter jug. This is possible because:	**4 is a multiple of gcd(3,5)=1** — Because gcd(3,5)=1, every amount from 1 to 5 (including 4) is measurable.
If both jugs are even sizes, can you ever measure an ODD amount?	**no — the GCD is even, so all measurable amounts are even** — Two even sizes have an even GCD, so every measurable amount (a multiple of that GCD) is even — no odd amount is reachable.
In a water-pouring puzzle you have	**fill a jug, empty a jug, or pour one into another until full or empty** —

jugs with no markings. The only actions are:	The three legal actions are fill, empty, and pour-between-jugs; there are no markings to read.
Gill's defining move is:	**pour back and forth to reach an exact target** — Gill measures exact amounts by systematic pouring — the hands-on face of GCD reasoning.
With a 4-liter and a 6-liter jug, can you measure exactly 3 liters?	**no — 3 is not a multiple of gcd(4,6)=2** — gcd(4,6)=2, so only even amounts (2, 4, 6) are measurable — 3 is impossible.
Big idea: water-pouring puzzles show that hidden inside a hands-on task is:	**a clean number rule (the GCD) that decides what's possible** — Behind the pouring is a clean number-theory rule: the GCD determines exactly which amounts can ever be measured.
The exact amounts you can measure with two jugs are always:	**multiples of the GCD of the two jug sizes** — Only multiples of gcd(a,b) can be measured — this is the central number-theory fact behind pouring puzzles.
If you wanted a puzzle where MANY amounts are possible, you'd choose jug sizes that are:	**coprime (gcd 1)** — Coprime sizes (gcd 1) make every whole amount up to the larger jug reachable — the most flexible design.
With a 3-liter jug and a 5-liter jug, the SMALLEST exact amount you can measure into a jug is:	**1 liter** — gcd(3,5)=1, so the smallest measurable amount is 1 liter — and therefore any whole amount up to 5 is reachable.
If two jugs are coprime, which whole amounts can you measure?	**every whole amount up to the larger jug** — Since gcd=1, every whole number is a multiple of 1, so all whole amounts up to the larger jug are reachable.
Before pouring, the FASTEST way to know if a target is reachable is to:	**check whether the target is a multiple of the GCD (and not bigger than the largest jug)** — Checking 'is the target a multiple of the GCD, and at most the largest jug?' tells you reachability before a single pour.
The GCD of 6 and 9 is:	**3** — Common divisors of 6 and 9 are 1 and 3; the greatest is 3.
Using Euclid on 12 and 8: 12 divided by 8 leaves remainder 4; 8 divided by 4 leaves remainder 0. The GCD is:	**4** — The last nonzero remainder in Euclid's steps (4) is gcd(12,8) = 4.

8. binary / Gray-code counting — free the rings by changing exactly one at a time

✂ Question (fold →)	Answer
Big idea: the Chinese-rings puzzle shows that binary/Gray-code counting is:	**a real, physical way to step through every combination one change at a time** — The rings physically realize Gray-code counting — stepping through all combinations with exactly one change each time.
In the 3-bit Gray code 000,001,011,010,110,111,101,100, the FIRST and LAST states differ by:	**one bit (000 vs 100)** — 000 and 100 differ in one bit — a 'cyclic' Gray code even wraps around with a single change.
Ordinary binary counting from 01 to 10 changes:	**two bits at once** — Going 01 to 10 flips both bits, so plain binary counting is NOT one-change-at-a-time — Gray code fixes that.
Why is a one-change-at-a-time order useful beyond puzzles?	**changing one thing at a time makes each step easy to check and undo** — Single-change steps are easy to verify and reverse, which is why Gray codes appear in dials, sensors, and error-avoidance.
Representing a puzzle state as bits is powerful because it lets you:	**count, compare, and order states using number rules** — Encoding states as bits turns the physical puzzle into numbers you can count, compare, and order with math.
Gray and Evan both use invariants/patterns. Their shared big idea is:	**a state's structure (its bits/parity) tells you how it behaves** — Both read a state's underlying structure (bits, parity) to predict what's possible — number-structure over trial and error.
If a ring puzzle has 5 rings, the number of possible states is:	**32** — 5 rings, 2 states each = 2^5 = 32 possible states.
In the rings puzzle, you may only add or remove a ring when the rings before it are in a special set position. This forces you to:	**change one ring at a time in a fixed order** — The mechanical rules only ever allow one ring to change at a time, which is precisely a Gray-code sequence.
In the Chinese-rings puzzle, each ring is either:	**on the bar or off the bar** — Each ring has two states — on the bar or off — just like a switch that is 0 or 1.
Solving the Chinese rings is HARD mostly because:	**the number of one-at-a-time steps grows quickly with more rings** — Because only one ring changes per step and there are ~2^n states, the step count grows fast as rings are added.
The binary string 000 for the rings means:	**all rings off** — 000 means every ring is off (0) — often the goal state of the puzzle.
How many bits (rings) do you need to count up to the value 8 in binary?	**4 (since 8 is 1000)** — 3 bits count 0-7; representing 8 (1000) needs a 4th bit.
Between two consecutive Gray-code states, the NUMBER of rings that change is always:	**exactly one** — By definition, consecutive Gray-code states differ by exactly one bit (one ring).
The rings puzzle teaches that a long, tricky task can be:	**broken into a fixed order of tiny one-step changes** — The lesson is that a complex task decomposes into a definite sequence of single, simple, reversible changes.

Which real device uses Gray-code-style one-change-at-a-time to avoid errors?	**a rotary position sensor (encoder) on a dial** — Rotary encoders use Gray code so only one bit changes between adjacent positions, preventing misreads at the boundary.
A 'Gray code' is a counting order where each step changes:	**exactly one bit** — In a Gray code, consecutive values differ in exactly one bit — one change at a time.
If you're mid-solve and forget your place, the one-change rule helps because:	**you can compare the current state to the neighbors that differ by one ring** — Since legal moves change one ring, only a couple of neighbor states are reachable — comparing them tells you where you are and where to go.
In binary, the number after 011 (counting up) is:	**100** — 011 is 3; adding 1 gives 4, which is 100 in binary.
How many different states does a 3-ring puzzle have?	**8** — With 2 choices per ring and 3 rings, there are 2^3 = 8 possible states.
A Gray code for n rings visits all 2^n states while:	**changing only one ring between each state** — A Gray code is a tour through all 2^n states where each step changes exactly one bit — no state repeated.
The 2-bit Gray code order is 00, 01, 11, 10. Between 01 and 11, which bit changed?	**the left bit (0 to 1)** — 01 -> 11 changes only the left bit from 0 to 1 — exactly one change, as a Gray code requires.
The binary number 101 equals which decimal value?	**5** — 101 = 1x4 + 0x2 + 1x1 = 5.
Gray's defining act is:	**freeing the rings by changing exactly one at a time in a fixed order** — Gray changes exactly one ring at a time in the fixed Gray-code order — one-bit-at-a-time counting made physical.
The 3-bit Gray code starts 000, 001, 011, 010, ... What comes next?	**110** — 010 -> 110 flips only the left bit, continuing the 3-bit Gray code (000,001,011,010,110,111,101,100).
Because each ring is on OR off, the puzzle state is naturally written as:	**a string of 0s and 1s (binary)** — With each ring as 0 (off) or 1 (on), the whole state is a binary string — one bit per ring.

9. reduction / monovariant — jump-and-remove toward a single survivor (peg solitaire)

✂ Question (fold →)	Answer
Vault's defining act is:	**jumping one peg over another and removing it, shrinking toward one survivor** — Vault jumps and removes pegs, steadily reducing the board toward a single survivor — reduction in action.
Reduction/monovariant thinking is valuable in computer science for:	**proving a program's loop will stop** — Programmers use a decreasing quantity (a monovariant / loop variant) to prove a loop must terminate — the peg-count idea applied to code.
A quantity that only ever moves in ONE direction (like the peg count) is called a:	**monovariant** — A monovariant is a quantity that changes in only one direction — here, the peg count only decreases.
The peg count can never INCREASE because:	**the only legal move removes a peg; nothing adds one** — No legal move adds a peg, and every jump removes one, so the count can only ever go down.
Since the number of jumps is fixed, what actually determines whether you SOLVE it?	**the ORDER of jumps — a good order avoids getting stuck** — The total jumps is fixed at N-1; success depends on the ORDER, which must avoid stranding pegs before the last jump.
Getting STUCK in peg solitaire means:	**no legal jumps remain but more than one peg is left** — You're stuck when several pegs remain but none has a legal jump — usually because pegs got isolated.
A legal peg-solitaire move is to:	**jump a peg over an adjacent peg into an empty hole, removing the jumped peg** — You jump a peg straight over a neighbor into an empty hole; the jumped peg is removed.
A monovariant argument answers which kind of question best?	**'will this process definitely end?'** — Monovariants are ideal for proving termination — showing a process must eventually stop.
If you start with 10 pegs and want to finish with 1, how many jumps do you make?	**9** — Each jump removes one peg, so going from 10 to 1 needs exactly 10 - 1 = 9 jumps.
If a start position has 20 pegs and you're now at 6 pegs, how many jumps have you made?	**14** — Each jump removes one peg, so 20 - 6 = 14 pegs removed means 14 jumps made.
A monovariant is a powerful proof tool because it lets you argue that a process:	**must stop (it can't decrease forever)** — A whole-number count that always decreases can't drop forever, so the process must terminate — the classic monovariant argument.
You have 4 pegs left and make 2 more jumps. How many pegs remain?	**2** — Two jumps remove two pegs: 4 - 2 = 2 pegs remain.
A jump needs THREE holes in a line: peg, peg, empty. After the jump they become:	**empty, empty, peg** — The jumper's start and the jumped peg become empty; the landing hole fills — peg,peg,empty becomes empty,empty,peg.
Peg solitaire is a REDUCTION puzzle	**shrinks (fewer pegs) with every move** — Every move reduces the peg count, so the problem shrinks toward the single-peg goal — the essence

because the board:	of reduction.
Each legal jump changes the number of pegs by:	**exactly minus one** — Every jump removes exactly one peg (the one jumped over), so the peg count drops by 1 each move.
The classic English peg board has 33 holes and starts with 32 pegs (center empty). To finish with 1 peg you make:	**31 jumps** — From 32 pegs to 1 peg is 32 - 1 = 31 jumps.
Which board start is EASIEST to reduce to one peg, in general?	**one where pegs stay densely connected as you jump** — Dense, connected pegs keep jumps available; scattered pegs strand and dead-end, so connectivity makes reduction easier.
Big idea: peg solitaire teaches that a task where every step SHRINKS the problem:	**is guaranteed to finish, so the challenge is ordering the steps well** — Because each step reduces the problem, it must end; the skill is ordering the jumps so you don't strand pegs before the finish.
Because the peg count only decreases, the puzzle CANNOT:	**loop forever — it must end** — Since the count strictly decreases, the puzzle can never loop and must terminate — a key use of a monovariant.
To AVOID getting stuck, a good habit is to:	**avoid stranding lone pegs far from others** — Isolated pegs can't be jumped or jump, so keeping pegs near others avoids dead ends.
A peg with NO empty hole beyond an adjacent peg:	**cannot make a jump right now** — A jump requires an empty hole to land in beyond the jumped peg; without one, that peg can't move.
Working BACKWARD from the final single peg can help because:	**you can see which earlier positions could lead to that finish** — Reasoning backward from the goal peg reveals the positions that could reach it — a planning trick (Retta's style, in the other app).
Which of these is ALSO a monovariant in a different puzzle?	**the number of disks not yet on the goal peg in Hanoi steadily reaching zero** — A quantity that steadily moves one direction toward a target (like disks reaching the goal peg) is a monovariant.
The number of jumps to solve is FIXED by the start, no matter the order. For a board of N pegs to 1 peg it is always:	**N minus 1** — Because each jump removes exactly one peg, reaching 1 peg always takes N - 1 jumps regardless of order.
Planning peg jumps to keep pegs connected is like Skiff's puzzle because both:	**must avoid moves that lead to a stuck (dead-end) state** — Both require choosing moves that keep a solution possible — avoiding dead-end states is shared strategy.

10. recursion — self-similarity, base case, and the recursive trace

✂ Question (fold →)	Answer
To move 3 disks A->C, the recursion first solves 'move 2 disks A->B.' That sub-call itself first:	**moves 1 disk A->C (its own base step)** — The 2-disk sub-call breaks into moving 1 disk (its base case) first — recursion descends to the base before building back up.
In recursion, solving n depends on n-1, which depends on n-2, and so on down to the base. This chain is called the:	**recursion (or call) stack** — The chain of paused sub-problems (n waiting on n-1 waiting on n-2...) is the recursion/call stack.
'Self-similarity' in recursion means:	**the smaller pieces look like tinier versions of the whole** — Self-similarity means each sub-problem is a smaller copy of the same problem — the defining feature of recursion.
Using M(1)=1 and M(n)=2M(n-1)+1, what is M(3)?	**7** — M(2)=2(1)+1=3, then M(3)=2(3)+1=7.
What happens if a recursion has NO base case?	**it never stops — it goes forever** — Without a base case the recursion never terminates — it keeps reducing forever, which is a bug.
For Tower of Hanoi, the base case is:	**moving a single disk (n = 1) directly** — The Hanoi base case is n = 1: move the single disk directly, no recursion needed.
Big idea: recursion is powerful because one small rule (base + recursive case) can:	**solve a problem of ANY size** — A single base+recursive rule handles every size, so recursion scales a tiny idea to problems of any size.
Which is the clearest sign a problem can be solved recursively?	**solving it needs the answer to a smaller version of the same problem** — If solving the problem depends on solving a smaller version of the SAME problem, recursion fits.
Why is recursion a good MATCH for Hanoi specifically?	**the puzzle is literally defined by a smaller copy of itself** — Hanoi's structure is self-similar (a tower contains a smaller tower), so recursion mirrors the puzzle exactly.
The doubling in M(n)=2M(n-1)+1 is why Hanoi move-counts:	**grow explosively as disks are added** — Because each level roughly doubles the work, the counts grow exponentially (2^n - 1) as disks are added.
The part that solves a big problem using a smaller copy of itself is the:	**recursive case** — The recursive case reduces the problem to a smaller instance of the same problem.
M(4) using the recurrence is:	**15** — M(4) = 2 x M(3) + 1 = 2 x 7 + 1 = 15.
Which everyday task is naturally recursive?	**opening nested boxes, each holding a smaller box** — Nested boxes are self-similar: open a box to find the same task on a smaller box, until the base case (an empty box).
A student writes a recursion that reduces n to n-2 each time, starting from an odd n, with base case n=0. This will:	**skip past 0 and never hit the base case (bug)** — Stepping down by 2 from an odd number never equals 0, so the base case is missed and the recursion runs away — a base-case bug.
A recurrence is useful because it describes:	**a whole family of answers with one short rule** — A recurrence captures the pattern for every size with one compact rule, generating the whole family of answers.

A good recursive design always includes BOTH:	**a base case AND progress toward it each recursive step** — Correct recursion needs a reachable base case and steps that always move closer to it, guaranteeing termination.
The closed-form for M(n)=2M(n-1)+1 with M(1)=1 is:	**M(n) = 2^n - 1** — The recurrence solves to the closed form 2^n - 1, letting you jump straight to the answer without unrolling.
The 'recursive leap of faith' means you may:	**assume the smaller case already works while you write the bigger case** — You trust the recursive call handles n-1 correctly, so you only design the single step that extends it to n.
The recurrence M(n) = 2 x M(n-1) + 1 means the work for n is:	**twice the work for n-1, plus one more step** — The recurrence says solving n costs two solves of n-1 (the smaller tower, twice) plus one big-disk move.
Recursion and decomposition are related because recursion is:	**decomposition where the smaller parts are the SAME problem** — Recursion is a special decomposition where every sub-problem is a smaller copy of the original.
The recursion 'unwinds' (finishes) when it:	**reaches the base case, then completes each waiting step back up** — Recursion descends to the base case, then unwinds back up, completing each waiting step in reverse order.
A 'closed form' is valuable because it lets you find M(50) without:	**computing all the earlier values first** — A closed form (2^n - 1) gives any term directly, so M(50) = 2^50 - 1 without computing M(1)...M(49).
A recursive solution has two parts. The part that stops the recursion is the:	**base case** — The base case is the smallest version solved directly, without further recursion — it stops the process.
'Unrolling' a recursion means:	**writing out the steps it would take, level by level** — Unrolling expands each recursive call into its sub-steps, showing the full sequence the recursion produces.
Nestor says a tall problem is just a shorter problem plus one step. This lets him:	**trust the smaller solution and only worry about the one extra step** — Recursion lets you assume the smaller case is solved and focus only on the one step that builds the bigger case.

11. parity invariants — the mod-2 argument, generalized across puzzles

✂ Question (fold →)	Answer
Because a knight flips color each move, after an EVEN number of moves it is on:	**the same color it started on** — An even number of color-flips returns the knight to its starting color — parity tracks this instantly.
A checkerboard-coloring argument uses parity by:	**coloring squares black/white and tracking whether a piece is on black or white** — Coloring the board like a checkerboard turns 'which color a piece sits on' into a parity you can track across moves.
'Parity' means whether a number is:	**even or odd** — Parity is the even-or-odd nature of a number — the remainder when divided by 2.
Parity arguments are trusted in math because they:	**give certainty, not just 'I couldn't find a way'** — A parity invariant proves impossibility with certainty — very different from merely failing to find a solution.
Big idea: parity is powerful because a simple even/odd fact can:	**decide a whole puzzle's possibility in one step** — A single parity invariant can settle whether a puzzle is solvable at all — enormous power from one even/odd fact.
Which question does a parity check answer BEST?	**'can the goal ever be reached from here?'** — Parity answers reachability: whether the goal shares the invariant needed to be reachable from the start.
Adding an even and an odd number gives:	**an odd number** — Even + odd = odd (e.g., 3 + 4 = 7).
In the sliding puzzle, swapping exactly two tiles changes the inversion count by an ODD amount, so it:	**flips the parity (even to odd or odd to even)** — A single swap changes inversions by an odd amount, flipping parity — which flips solvability.
Evan checks parity to decide whether to even TRY a puzzle. This saves:	**time wasted on an impossible task** — A quick parity check reveals impossibility up front, so Evan avoids wasting effort on unsolvable scrambles.
A light switch flipped 7 times ends up:	**opposite to its starting position (7 is odd)** — An odd number of flips leaves one net flip, so the switch ends opposite to its start.
Parity and Gray's binary bits connect because:	**the last binary bit IS the parity (even ends in 0, odd in 1)** — A number's last binary bit is its parity (0 = even, 1 = odd) — Evan's parity and Gray's bits are two views of the same structure.
Adding two even numbers always gives:	**an even number** — Even + even = even (e.g., 4 + 6 = 10) — one of the basic parity rules.
The reason a single tile-swap is a good 'prank' to make a puzzle unsolvable is that it:	**flips the parity invariant so the original goal becomes unreachable** — One swap flips the parity invariant, so the goal's parity no longer matches — the puzzle can never be solved, undetectably.
If a scramble has an EVEN inversion parity and the goal also has even parity (on an odd-width board), the	**solvable** — Matching parity (invariant) between scramble and goal means the goal is reachable — the scramble is solvable.

scramble is:	
The famous 'mutilated chessboard' (remove two opposite corners) can't be tiled by dominoes because:	**the two removed corners are the same color, so black and white counts don't match** — Each domino covers one black and one white square; removing two same-color corners leaves unequal counts, so a perfect tiling is impossible — a parity proof.
A light switch flipped 10 times ends up:	**in its starting position (10 is even)** — An even number of flips returns the switch to its starting state — a tiny parity invariant.
Two things flip together and cancel out. This 'pairs cancel' idea is the heart of:	**parity (mod-2) reasoning** — Parity reasoning tracks whether things happen an even or odd number of times — pairs cancel, so only the leftover parity matters.
Evan's core habit across all these puzzles is to:	**find a quantity that never changes, then check the goal against it** — Evan finds an invariant (often a parity) and compares the goal to it — if they differ, the goal is unreachable.
A knight on a chessboard always moves from a square of one color to:	**the opposite color** — A knight always lands on the opposite color, so its square-color parity flips every move — a classic parity invariant.
Suppose a puzzle's 'blank' must return to its home corner, and each move flips its checkerboard color. To come home it needs:	**an even number of moves** — To return to its starting color (its home), the blank needs an even number of color-flipping moves.
Why are invariants useful for solvability?	**if the goal has a different invariant than the start, it's unreachable** — Since legal moves preserve an invariant, a goal with a different invariant value can never be reached — proving impossibility.
An 'invariant' is a fact about a puzzle that:	**never changes, no matter which legal moves you make** — An invariant is a property preserved by every legal move — it stays constant throughout the puzzle.
In the Tower of Hanoi, the smallest disk moves on every OTHER move. Its move-times follow:	**a fixed parity pattern (odd-numbered moves)** — In the optimal Hanoi solve the smallest disk moves on every odd-numbered move — a parity pattern in the move sequence.
To USE an invariant to prove impossibility, you must show the invariant is:	**the same for the start but different for the goal** — If a legal-move-preserved invariant differs between start and goal, no sequence of moves connects them — impossibility proven.
The power of a parity argument is that it can prove something is IMPOSSIBLE:	**without checking every possibility** — A parity invariant proves impossibility with a single argument, avoiding the need to test countless cases.

12. search strategies — breadth-first vs depth-first intuition

✂ Question (fold →)	Answer
A search that finds the goal but reports a 10-move path when a 6-move path exists is:	**correct but not optimal** — Reaching the goal is correct, but a longer-than-necessary path is non-optimal — a distinction Minim would flag.
Which real task is like DFS?	**exploring a maze by following one path to its end, then backtracking** — Following one maze path to its end, then backtracking, is DFS — deep before wide.
'Iterative deepening' combines the best of both by:	**running DFS with a small depth limit, then a bigger one, until the goal is found** — Iterative deepening repeats depth-limited DFS with growing limits, getting BFS-like shortest paths with DFS-like low memory.
For a Hanoi-style puzzle where you want the OPTIMAL solution, you'd prefer:	**BFS (or a smarter guided search), for the shortest path** — Because optimal means fewest moves, BFS (which finds shortest paths first) is the natural choice — echoing Minim's optimality.
BFS tends to use MORE memory than DFS because it:	**must remember all states at the current level (a wide frontier)** — BFS holds the whole current level (frontier) in memory, which can be huge; DFS only holds one path plus its choices.
Depth-first search (DFS) explores states:	**by following one path as deep as possible, then backtracking** — DFS plunges down one branch until it dead-ends, then backtracks to try another — deep before wide.
For the 15-puzzle, a common heuristic is counting how far each tile is from home. This helps the search:	**prefer moves that reduce total tile distance** — Summing each tile's distance-to-home estimates closeness to the goal, guiding the search toward lower-distance states.
To AVOID revisiting states, any good search should:	**keep a set of already-visited states** — Tracking visited states prevents loops and repeated work in both BFS and DFS.
Breadth-first search (BFS) explores states:	**level by level — all one-move-away states, then all two-move-away, and so on** — BFS explores in rings of distance: all states 1 move away, then 2 moves away, etc. — nearest first.
Big idea: search strategies show that solving isn't just about the puzzle, but also about:	**the smart method you use to explore the possibilities** — The strategy for exploring possibilities is as important as the puzzle itself — the heart of computational problem-solving.
BFS uses a 'queue' (first-in, first-out). This means it processes states:	**in the order they were discovered — oldest first** — BFS uses a FIFO queue, so it expands states in discovery order — which produces the level-by-level behavior.
Wend surveys reachable states before moving. In search terms she is:	**expanding the frontier to see options before committing** — Wend expands the frontier and weighs options before choosing — the disciplined core of any search.
The deepest lesson about search is that HOW you explore:	**changes how fast and how well you find a solution** — The same state space explored different ways yields different speed, memory use, and solution quality — strategy matters.

Why can't we just brute-force every state in a big puzzle?	**there are far too many states to check them all** — State counts explode (trillions), so exhaustively checking every state is infeasible — smart, guided search is needed.
The main reason to pick a search STRATEGY (not just search) is that different strategies:	**trade off memory, speed, and solution quality differently** — Strategies differ in memory, speed, and whether they find shortest solutions — you pick based on what matters for the task.
BFS explored level 1 (3 states) and level 2 (9 states) without finding the goal. The goal appears at level 3. The shortest solution length is:	**3 moves** — In BFS the level equals distance, so a goal first found at level 3 is exactly 3 moves away — the shortest solution.
DFS uses a 'stack' (last-in, first-out). This means it processes states:	**the most recently discovered first** — DFS uses a LIFO stack, so it dives into the newest state first — producing deep-first behavior.
DFS with a DEPTH LIMIT stops going deeper after a set number of moves. This prevents:	**diving forever down an endless or very deep branch** — A depth limit stops DFS from diving too deep; if no solution is found, the limit can be raised (iterative deepening).
If you only need SOME solution quickly and memory is tight, a reasonable choice is:	**DFS** — When any solution will do and memory is limited, DFS's small footprint makes it a practical choice.
Which real task is like BFS?	**checking all your closest friends first, then friends-of-friends, to pass a message** — Spreading a message to closest contacts first, then their contacts, is BFS — expanding by distance in rings.
DFS uses LESS memory but risks:	**going very deep down a wrong branch before backtracking** — DFS is memory-light but can waste time diving deep down an unproductive branch before it backtracks.
A 'guided' (heuristic) search improves on plain BFS/DFS by:	**preferring states that look closer to the goal** — A heuristic guides search toward states estimated closer to the goal, exploring promising options first.
A 'frontier' in search is:	**the set of states discovered but not yet explored** — The frontier is the boundary of discovered-but-unexplored states — what the search will expand next.
Why might DFS find a LONGER solution than necessary?	**it may reach the goal down a deep branch before checking shorter ones** — DFS may reach the goal along a long deep branch before ever exploring a shorter path, so its first solution can be non-optimal.
Which search is GUARANTEED to find the fewest-move solution first?	**breadth-first search** — BFS reaches states in order of distance, so the first time it hits the goal is via a fewest-move path.

13. constraint satisfaction — modeling, pruning illegal states, and constraint propagation

✂ Question (fold →)	Answer
Constraint satisfaction appears in real software for:	**timetabling, seating charts, and resource scheduling** — CSPs power timetabling, scheduling, seating, and resource-allocation software — wherever rules limit valid combinations.
A Sudoku cell that can only be a 4 (all other digits pruned) should be filled with 4. This is a:	**forced move (no guess needed)** — When propagation leaves a single legal value, that placement is forced — certain, not a guess.
Skiff's patience ('check every move against the rules') matches the CSP habit of:	**verifying each partial assignment stays consistent** — Skiff's step-by-step legality checking is exactly CSP consistency-checking of each partial assignment.
Coloring a map so neighbors differ is a CSP. The constraint is:	**adjacent regions must have different colors** — Map coloring's constraint: any two regions sharing a border must be colored differently.
If propagation leaves a variable with ZERO legal values, you have found a:	**contradiction — this branch can't work, backtrack** — A variable with no legal values means the current path is impossible — you backtrack to the last choice.
Why is constraint satisfaction often FASTER than blind search?	**pruning cuts away huge numbers of illegal states before exploring them** — By pruning illegal states before exploring them, CSP shrinks the search space dramatically versus blind search.
When a guess IS needed, pairing it with propagation means:	**after guessing, immediately prune and check for contradictions** — After a guess you propagate and check for contradictions immediately, so a wrong guess is caught and undone quickly.
Scheduling classes so no student has two at once is a CSP where the constraint is:	**no two classes a student takes may share a time slot** — The constraint forbids overlapping time slots for classes a student shares — a real-world CSP.
The BEST summary of constraint satisfaction is:	**narrow the choices using the rules, then guess only where you must** — CSP solving narrows options via constraint propagation and reserves guessing for the few genuinely undetermined choices.
In Sudoku, a 'constraint' is:	**no repeated digit in any row, column, or box** — Sudoku's constraints forbid any repeated digit within a row, column, or 3x3 box.
Handling the 'goat' (most-constrained piece) first in river-crossing is an example of:	**the most-constrained-first heuristic** — Prioritizing the goat (the most-restricted piece) is the most-constrained-first heuristic applied to a river-crossing.
In river-crossing, Skiff prunes states where:	**a forbidden pair is left unattended** — Skiff prunes any state leaving a forbidden pair unattended — the same pruning idea CSPs use everywhere.
The difference between constraint satisfaction and plain search is that CSP:	**uses the rules to eliminate options, not just to check final states** — CSP uses constraints proactively to prune options during search, not merely to test the end state.
Good CSP solving reduces	**propagating constraints to force as many values as possible first** — Propagating constraints fills all forced values first, so guesses are needed only

GUESSING by:	where genuinely undetermined.
A CSP solver that checks consistency after every assignment is doing:	**constraint checking to fail fast** — Checking consistency after each assignment lets the solver fail fast and backtrack before wasting effort.
A constraint satisfaction problem (CSP) has variables, values, and:	**constraints that restrict which value combinations are allowed** — A CSP assigns values to variables subject to constraints that forbid certain combinations.
Big idea: constraints don't just make a puzzle HARDER — used well, they:	**guide you to the answer by eliminating the impossible** — Constraints are a tool: by eliminating impossible options they steer you toward the solution — the essence of CSP.
'Constraint propagation' means:	**one decision narrows the options for related variables** — Constraint propagation spreads the effect of one assignment: placing a value prunes it from every related variable.
As you solve a CSP well, variable domains generally:	**shrink as constraints eliminate options** — Good propagation shrinks domains as constraints eliminate options, driving variables toward forced values.
The 'domain' of a variable is:	**the set of values it is still allowed to take** — A variable's domain is its set of still-legal values; propagation prunes it toward a single value.
Why does handling the most-constrained choice first help?	**it catches forced moves and dead ends early, avoiding wasted work** — Tight variables reveal forced moves and dead ends soonest, so you avoid deep wasted exploration.
Modeling a real problem as a CSP means writing down:	**the variables, their possible values, and the constraints between them** — CSP modeling captures the variables, their domains (possible values), and the constraints linking them.
'Pruning' a value means:	**removing a value that would break a constraint** — Pruning removes values that can't work because they'd violate a constraint, shrinking the search.
The 'most-constrained variable first' heuristic says to fill in:	**the cell/choice with the FEWEST legal options next** — Filling the most-constrained variable first (fewest legal values) reduces guesswork and catches forced moves early.
If two constraints CANNOT both be satisfied, the problem is:	**over-constrained (no solution exists)** — When constraints conflict so no assignment satisfies all, the CSP is over-constrained and has no solution.

14. optimality proofs — lower bounds and why you cannot do better

✂ Question (fold →)	Answer
Big idea: proving optimality means showing your answer is BOTH:	**achievable (a real solution) and unbeatable (a matching lower bound)** — Optimality is proven when a solution is both achievable (upper bound) and unbeatable (matching lower bound).
A lower bound of 3 and a found solution of 5 tells you the true minimum is:	**somewhere between 3 and 5 (not yet pinned down)** — With a lower bound of 3 and an upper bound of 5, the minimum lies in [3,5]; you must tighten a bound to pin it down.
Minim finds a 15-move Hanoi-4 solve AND cites the lower-bound argument. He has proven:	**15 is exactly optimal for 4 disks** — A 15-move solution (upper bound) plus the matching lower-bound argument proves 15 is exactly optimal.
Which statement is a valid LOWER-bound claim?	**'no solution uses fewer than 7 moves'** — 'No solution uses fewer than 7' is a lower-bound (floor) claim; 'I found one in 7' is an upper-bound (ceiling) claim.
You PROVE a solution is optimal when the lower bound and upper bound:	**meet at the same number** — When a lower bound (at least N) meets an upper bound (a solution in N), the minimum is exactly N — proven optimal.
Why do computer scientists care about lower bounds?	**they tell you when to STOP improving — you've hit the best possible** — A lower bound tells you the best achievable result, so once you reach it there's no point trying to improve further.
An 'upper bound' is:	**a solution you actually found, showing it CAN be done in that many moves** — An upper bound is a demonstrated solution: it proves the task can be done in at most that many moves.
The gap between your best solution and your best lower bound is called the:	**optimality gap** — The optimality gap is the difference between the upper and lower bounds; closing it to zero proves optimality.
Why is finding a solution NOT enough to claim optimality?	**a shorter solution might still exist unless you prove a matching lower bound** — A solution only shows a task CAN be done in that many moves; optimality needs a lower-bound proof that nothing does better.
To PROVE the minimum is exactly 4 in that situation, you could:	**find a 4-move solution AND argue at least 4 are required** — Showing a 4-move solution (upper bound 4) and arguing at least 4 are needed (lower bound 4) proves the minimum is exactly 4.
A lower bound is called 'tight' when:	**a real solution actually achieves it** — A tight lower bound is achievable — some real solution meets it exactly, proving optimality.
Optimality proofs connect Minim to Evan because both:	**use a careful argument (not trial) to establish certainty** — Minim proves a minimum and Evan proves (im)possibility — both reach certainty by argument, not trial and error.
The Hanoi lower bound uses the SAME recursive structure as the solution. This shows:	**the optimal plan and the impossibility of doing better share one idea** — Hanoi's recursion both constructs the optimal solution and proves the lower bound — one elegant structure does both.
To CLOSE an optimality gap you can	**find a shorter solution or prove a higher lower bound** — Close the gap

either:	by improving the solution (lower the upper bound) or strengthening the argument (raise the lower bound).
For Hanoi, the recurrence M(n)=2M(n-1)+1 gives BOTH bounds because it shows:	**a solution in 2^n - 1 moves AND that fewer is impossible** — The recursive plan achieves 2^n - 1 (upper bound) and the must-clear-and-rebuild argument forces at least 2^n - 1 (lower bound) — so it's exactly optimal.
Minim's motto 'never a wasted move' is really a claim that his solution:	**matches the lower bound (is optimal)** — 'No wasted move' means his solution equals the lower bound — it's provably optimal.
In a sliding puzzle, if 6 tiles are out of place and each move repositions one tile toward home, a quick lower bound is:	**at least 6 moves** — Each misplaced tile needs at least one move to reach home, giving a lower bound of at least 6 (often more).
A 'proof by contradiction' for a lower bound would:	**assume a shorter solution exists, then show it breaks a rule or count** — Assume a shorter solution exists, then show it forces an impossibility (breaks a count or rule) — proving no shorter solution can exist.
If someone claims a 3-disk Hanoi in 6 moves, you know it's WRONG because:	**the proven minimum is 7, so 6 is below the lower bound (impossible)** — The lower bound for 3 disks is 7; a claimed 6-move solve is below the proven floor, so it can't be a valid legal solve.
A simple lower-bound argument in peg solitaire: to go from N pegs to 1 you must remove N-1 pegs, so you need at least:	**N-1 jumps** — Each jump removes one peg, so removing N-1 pegs requires at least N-1 jumps — and exactly N-1 suffices.
A 'lower bound' on the number of moves is:	**a number you can prove no solution can beat** — A lower bound is a proven minimum: no solution can use fewer moves than it.
A 'counting argument' for a lower bound might say:	**each move can fix at most one thing, and k things are broken, so at least k moves are needed** — If each move can repair at most one 'broken' thing and k are broken, at least k moves are forced — a counting lower bound.
The lower-bound argument for Hanoi rests on the fact that the biggest disk:	**must move at least once, and that requires clearing then rebuilding the smaller tower** — The biggest disk must move, which forces at least M(n-1) moves to clear the smaller tower and M(n-1) to rebuild it — the lower bound.
Real software uses lower-bound thinking to:	**know when a route or schedule is already as good as possible** — Route planners and schedulers use lower bounds to recognize when a solution is optimal and stop searching.
The most rigorous mindset Minim models is:	**'I won't call it best until I can prove nothing beats it'** — Minim insists on a proof (a matching lower bound) before calling a solution optimal — rigor over assumption.

15. number reasoning — gcd, modular arithmetic, and binary behind puzzle moves

✂ Question (fold →)	Answer
If a target amount is NOT a multiple of the gcd, then in mod terms:	**target mod gcd is not zero, so it is unreachable** — A reachable amount must be a multiple of the gcd (target mod gcd = 0); a nonzero remainder means it's unreachable.
Two numbers are coprime when their GCD is:	**1** — Coprime numbers share no factor except 1, so their GCD is 1.
Parity (even/odd) is really arithmetic mod:	**2** — Even/odd is the remainder mod 2 (0 or 1) — parity is arithmetic mod 2.
What is 17 mod 5 (the remainder when 17 is divided by 5)?	**2** — 5 goes into 17 three times (15) with 2 left over, so 17 mod 5 = 2.
Number reasoning lets you answer 'is this reachable?' WITHOUT:	**trying every possible sequence of moves** — A number-theory check (gcd, parity, mod) answers reachability directly, replacing brute-force trial of every sequence.
Which pairing correctly matches idea to puzzle?	**GCD -> water pouring; binary -> Chinese rings; parity -> sliding tiles** — GCD governs pouring (Gill), binary governs the rings (Gray), parity governs sliding-tile solvability (Evan).
Hanoi's move-count 2^n - 1 in binary is a string of:	**n ones (e.g., 3 disks -> 111 = 7)** — 2^n - 1 is n ones in binary (7 = 111, 15 = 1111) — a neat binary fact behind Hanoi.
Binary place values double each step: 1, 2, 4, 8, 16, ... The 7th place value is:	**64** — Counting places 1,2,4,8,16,32,64, the 7th place value is 2^6 = 64.
Euclid's algorithm finds a GCD by repeatedly taking:	**remainders (mod) until one is zero** — Euclid's algorithm repeatedly replaces the pair with (smaller, remainder) until the remainder is 0 — modular arithmetic in action.
A clock shows 9. Seven hours later it shows 4, because 9 + 7 = 16 and:	**16 mod 12 = 4** — Clock time is mod 12: 16 mod 12 = 4, so 9 + 7 hours reads 4 o'clock.
The deep reason number reasoning is so powerful in puzzles is that numbers:	**capture the structure that moves must respect** — Numbers capture invariant structure that legal moves must respect, so number rules predict what's possible.
In a puzzle where a piece cycles through 4 positions, its position after 10 steps is found by:	**10 mod 4 = 2 (the 3rd position, counting from 0)** — With a cycle of 4, position = 10 mod 4 = 2 — modular arithmetic tracks repeating cycles.
The GCD of two numbers is always:	**a divisor of both numbers** — By definition the GCD divides both numbers evenly — it's their greatest common divisor.
'Modular arithmetic' (mod) is about:	**the remainder after dividing** — Modular arithmetic keeps only the remainder after division, wrapping around like a clock.
With 6- and 10-liter jugs, the	

measurable amounts are multiples of gcd(6,10). That gcd is:	**2** — 6 = 2x3 and 10 = 2x5 share only the factor 2, so gcd(6,10) = 2 — only even amounts measurable.
The binary number 1101 in decimal is:	**13** — 1101 = 8 + 4 + 0 + 1 = 13.
Why do so many different puzzles hide the SAME number ideas?	**states and moves are really numbers, so number rules govern them** — Once you encode states and moves as numbers, the same arithmetic (gcd, mod, binary) governs many different-looking puzzles.
In a water-pouring puzzle, measurable amounts are exactly the multiples of:	**the GCD of the jug sizes** — Only multiples of gcd(a,b) are measurable — the central number fact of pouring puzzles.
A puzzle piece returns to start every 6 moves. After 100 moves it is at position:	**100 mod 6 = 4 moves into the cycle** — With a 6-move cycle, 100 mod 6 = 4, so the piece is 4 moves into the cycle after 100 moves.
With coprime jugs, EVERY whole amount up to the larger jug is measurable because:	**every whole number is a multiple of 1 (the gcd)** — Since the gcd is 1 and every whole number is a multiple of 1, all whole amounts up to the larger jug are measurable.
Big idea: behind hands-on puzzles lies:	**clean number rules (gcd, mod, binary) that decide what's possible** — Underneath the physical play, clean number rules (gcd, modular arithmetic, binary) determine what is possible — the unifying lesson.
A binary number represents a value using only:	**0s and 1s (powers of two)** — Binary uses only 0 and 1, with place values that are powers of two (1, 2, 4, 8, ...).
The reason a Gray code changes ONE bit per step relates to which number idea?	**binary representation of states** — Gray codes are a binary counting order where each step flips one bit — rooted in binary representation.
The GCD of 18 and 24 is:	**6** — 18 = 2x3x3, 24 = 2x2x2x3; the shared factors give 2x3 = 6.
Gill and Gray together show that a hands-on puzzle can be understood by:	**translating it into numbers and applying arithmetic rules** — Both translate a physical puzzle into numbers (gcd, binary) and let arithmetic explain what's possible.

16. synthesis — mixed review connecting recursion, search, parity, constraints, optimality, gcd, and reduction

✂ Question (fold →)	Answer
Every move removes one piece and the board shrinks toward one survivor. This is:	**reduction / monovariant (Vault)** — A steadily shrinking count toward one piece is reduction / a monovariant — Vault's primitive.
A puzzle state is written as bits and you count through all states changing one bit at a time. This connects:	**binary representation (Gray) with state-space search (Wend)** — Encoding states as bits (Gray) and stepping one bit at a time is literally touring the state space (Wend) — two primitives, one idea.
With 9- and 12-liter jugs, which amount is IMPOSSIBLE to measure?	**5 liters** — gcd(9,12)=3, so only multiples of 3 (3,6,9) are measurable — 5 is impossible.
Across the whole app, 'a good puzzler gets stuck on purpose' means:	**being stuck is part of learning; you try a strategy, learn, and adjust** — Sprocket normalizes being stuck: it's where you notice a strategy isn't working and choose a better primitive — productive, not shameful.
A puzzle is solved by breaking it into a smaller copy of itself. This calls for:	**recursion (Nestor)** — Solving via a smaller copy of the same problem is recursion — Nestor's primitive.
You're told a 4-disk Hanoi was solved in 12 moves. This is:	**impossible — the minimum is 15, so 12 is below the lower bound** — The proven minimum for 4 disks is 15; a legal solve can't use fewer, so 12 is impossible (Minim's lower bound).
You want the FEWEST-move solution and to prove none is shorter. This is:	**optimality with a lower-bound proof (Minim)** — Finding the fewest moves and proving nothing beats it is optimality — Minim's primitive.
The unifying thread across ALL PuzzleForge families is:	**make your solving strategy visible and choose the idea the situation needs** — The through-line is making your strategy visible and matching the right primitive to the situation — Sprocket's workshop frame.
Which TWO primitives combine when you search a state space AND want the fewest moves?	**search (Wend) + optimality (Minim)** — Searching for the shortest path combines Wend's state-space search with Minim's optimality.
A ring puzzle where you change exactly one ring at a time, in a fixed order, uses:	**binary / Gray-code counting (Gray)** — One-change-at-a-time over on/off rings is Gray-code / binary counting — Gray's primitive.
A 5-disk Hanoi tower takes how many optimal moves?	**31** — 2^5 - 1 = 32 - 1 = 31 optimal moves.
You want to know if a scramble can EVER be solved, before trying. Use:	**a parity/invariant check (Evan)** — Checking reachability up front is a parity/invariant question — Evan's primitive.
In a river-crossing, focusing on the MOST-restricted item first is a heuristic shared with:	**Sudoku's most-constrained-cell-first (Skiff)** — Handling the most-constrained element first is a constraint-satisfaction heuristic — the goat and the tightest Sudoku cell, both Skiff.
You must measure exactly 4 liters using a 3-liter and 5-liter jug. The idea that decides if it's possible is:	**gcd (Gill)** — Measurability by pouring is a gcd question — Gill's primitive (gcd(3,5)=1, so 4 is reachable).

Which primitive best proves a process MUST END?	**a monovariant / reduction (Vault)** — A monovariant (a strictly decreasing count) proves termination — Vault's reduction idea.
You map out which positions you can reach before committing to a move. This is:	**state-space search (Wend)** — Mapping reachable states before moving is state-space search — Wend's primitive.
Starting peg solitaire with 15 pegs, to reach 1 peg you make:	**14 jumps** — Each jump removes one peg, so 15 to 1 is 15 - 1 = 14 jumps.
A target car can't leave until blockers move in the right order. This is:	**look-ahead planning (Faro)** — Ordering moves to clear a path is look-ahead planning — Faro's primitive.
BFS is preferred over DFS when you specifically need:	**the shortest (fewest-move) solution** — BFS explores by distance, so it finds the fewest-move solution first — the reason to prefer it (Wend + Minim).
A tower of 6 disks: optimal move count?	**63** — 2^6 - 1 = 64 - 1 = 63 optimal moves.
Which situation is best solved by working BACKWARD from the goal?	**figuring out what must be cleared for a target car to exit** — Reasoning backward from the target's escape reveals what must be cleared — Faro's backward look-ahead.
You must ferry items across a river without ever leaving a forbidden pair. This is:	**constraint satisfaction (Skiff)** — Keeping every state legal (no forbidden pair) is constraint satisfaction — Skiff's primitive.
Someone pulls two tiles out of a solved sliding puzzle and swaps them. The puzzle is now:	**unsolvable (parity flipped)** — A single swap flips the parity invariant, turning a solvable board unsolvable — Evan's warning.
Big idea of PuzzleForge: every puzzle hides ONE key rule, and solving well means:	**finding that rule so the moves fall into place** — PuzzleForge's thesis: find the one hidden rule (recursion, parity, gcd, constraint, ...) and the moves fall into place.
The best reason to check an INVARIANT before solving is:	**to avoid spending effort on an impossible puzzle** — Checking an invariant first tells you if the goal is reachable, saving effort on impossible puzzles (Evan).

Keep going

You practiced **400 cards** from PuzzleForge. 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. Released under Creative Commons BY-NC-SA 4.0 for educational reuse.

Keep exploring online

Play it now	Meet the cast	More free books
`https://spark-and-anvil.org/play/puzzleforge`	`https://spark-and-anvil.org/cast/puzzleforge`	`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.