Chapter 2: Games and Representations

Algorithmic Game Theory

Basic concepts | Normal and extensive form | Classification | Prisoner's Dilemma | Classic 2×2 games | Guess 2/3 of the average | Combinatorial games and minimax | Backward induction | A look at mechanism design

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/agt/slides/Chapter_02_Games_and_Representations.html

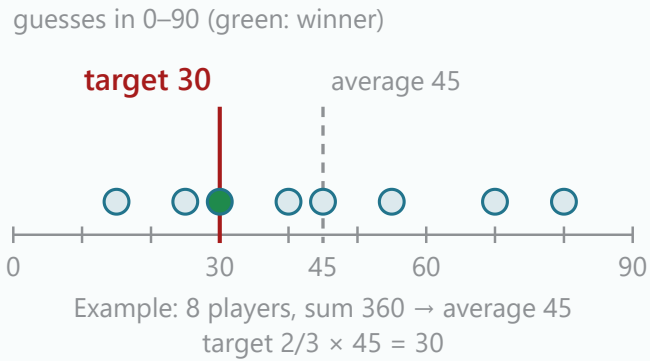
Warm-up Game: Guess 2/3 of the Average

The rules

Everyone secretly writes a number (integer or decimal) between 0 and 90; the guess closest to 2/3 of the average of all guesses wins (ties share the prize).

Think about it

- What do you think 2/3 of the average will be in this class?
- If everyone thinks as carefully as you, which number wins?



[Guess 2/3 of the average →]

Learning Outcomes

Concepts

Define players, strategies, payoffs, information, rationality and common knowledge on an example.

Representation and classification

Use the normal form (matrix) and the extensive form (tree); classify games by timing, binding agreements, total payoff, symmetry, information and horizon.

Equilibrium and efficiency

Explain Nash equilibrium intuitively; show that defection is dominant in the Prisoner's Dilemma and that the equilibrium is not Pareto optimal.

Classic games

Find the pure equilibria of six classic 2×2 games using best responses; interpret their conflict and coordination structure.

Algorithms

Explain level-k reasoning; solve small games by hand and in code with minimax and Nim's XOR strategy.

Sequential games

Find subgame-perfect equilibria by backward induction; interpret non-credible threats and the difference between finite and infinite horizons.

What Is Game Theory?

- A mathematical tool for situations in which decision makers interact **strategically**: my best decision depends on what you will do.
- The decision makers are called **players** (agents); the theory tries to explain how they behave and how the outcome takes shape.
- Players are assumed to act **rationally** and in their own interest.
- Widely used in economics, political science, biology, psychology and computer science.

Economics
Price competition between firms, auctions, bargaining.

Computer networks
Selfish routing, where every packet or driver picks its own route.

Artificial intelligence
Multi-agent systems, game-playing programs, learning agents.

Algorithmic game theory adds two questions: How hard is it to *compute* an equilibrium? How do we design *rules* (algorithms, protocols, auctions) that lead to good outcomes among selfish agents?

A Brief History: From Game Theory to Algorithmic Game Theory

Year	Development
1913	Zermelo: in finite perfect-information games such as chess, one side can force a win or both can force a draw.
1928	von Neumann: the minimax theorem for two-player zero-sum games.
1944	von Neumann and Morgenstern, *Theory of Games and Economic Behavior*: game theory becomes a field.
1950	Nash: every finite game has a mixed-strategy equilibrium. Same year: Flood–Dresher at RAND; Tucker's Prisoner's Dilemma story.
1961	Vickrey: truthful bidding is dominant in the second-price auction.
1994	Nobel Prize in economics to Nash, Harsanyi and Selten.
1999	Koutsoupias–Papadimitriou (worst-case equilibrium ratio, later the Price of Anarchy) and Nisan–Ronen (algorithmic mechanism design): algorithmic game theory is born.
2006	No fast method is known for Nash equilibria in general games: the problem (exact for two players, approximate for many) is PPAD-complete (Daskalakis–Goldberg–Papadimitriou; two players: Chen–Deng).
2012, 2020	Nobel: Roth and Shapley (stable matching, market design); Milgrom and Wilson (auction design).

BASIC CONCEPTS

Components of a Game

Players

The people or organizations making decisions.

$N = \{1, \ldots, n\}$

Actions / Strategies

The set of actions each player can choose.

$S_i = \{\text{Rock, Paper, Scissors}\}$

Payoffs

The value of each outcome to a player; it depends on *everyone's* choices.

$u_i(s_1, \ldots, s_n)$

Information

Who knows what, and when? Do you see your opponent's move?

$\text{chess} \neq \text{poker}$

Rules and order

Simultaneous or sequential? The rules linking actions to payoffs.

normal / extensive form

Outcome

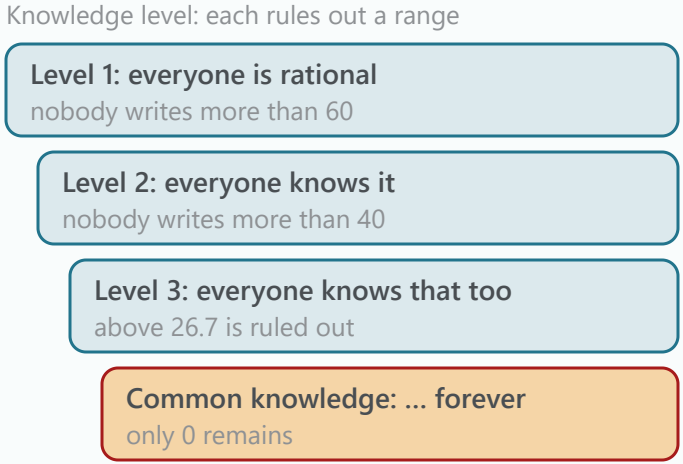
Determined jointly by all players' choices.

$s = (s_1, \ldots, s_n)$

A game: a situation in which players interact and each player's choice affects everyone's payoff.

Players and Rationality

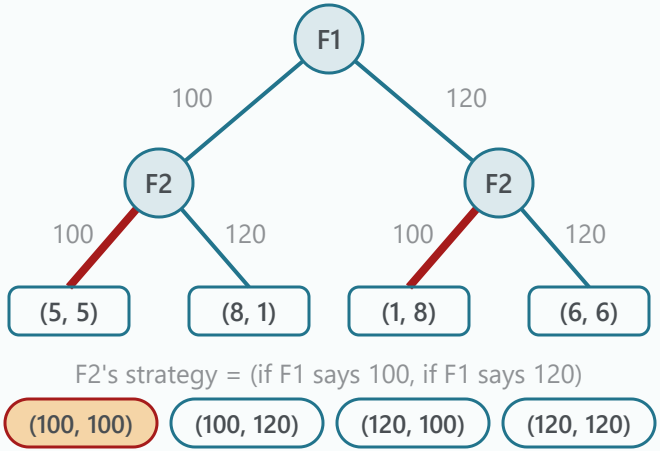
- **Rational**: each player makes the choice that maximizes its own expected utility given its beliefs.
- **Self-interest**: a player follows its own utility function; others' payoffs matter only if they enter that utility.
- **Common knowledge**: everyone is rational, everyone knows this, everyone knows that everyone knows it, ...



Rational ≠ selfish: a utility function may include fairness, altruism or reputation. The outcome results from an **interaction** that no single player controls.

Actions and Strategies

- **Strategy**: the plan of action a player chooses. In sequential games it is a *complete plan* saying what to do in *every* possible situation.
- On the right, F2 sees F1's price and responds. F2 has 2 decision nodes with 2 actions each: $2 \cdot 2 =$ **4 strategies**, though only 2 actions (with 3 nodes, $2 \cdot 2 \cdot 2 = 8$).
- A strategy rests on **expectations** about how others behave.



Pure strategy

Choosing a single plan for sure.

`"always Rock"`

Mixed strategy

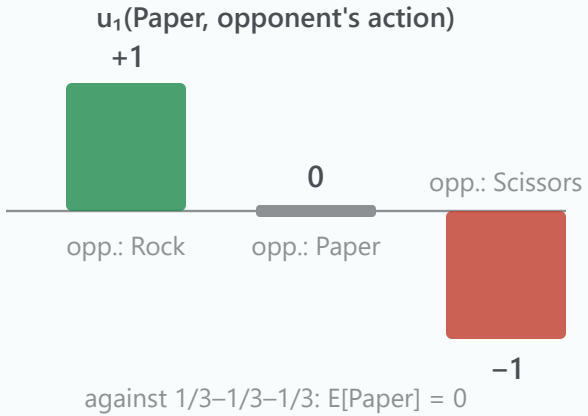
A probability distribution over pure strategies.

`(Rock 1/3, Paper 1/3, Scissors 1/3)`

Strategy profile: one strategy per player, $s = (s_1, \ldots, s_n)$. The profile determines the outcome.

Payoffs and Utility Functions

- The value a player obtains, which depends on the strategy profile chosen by **all players**; measured by the utility function u_i.
- Players choose strategies to maximize their payoffs.
- It may be money, points, years in prison (with a minus sign) or a level of satisfaction.
- With mixed strategies a player computes its **expected** payoff.



The same action (Paper) pays +1, 0 or −1 depending on the opponent's action: a payoff is a function of the **profile**, u_1(Paper, Rock) = +1.

Ordinal and Cardinal Utility

Ordinal utility gives only the order (Opera ≻ Football ≻ Home; ≻: "is preferred to"). Order-preserving changes (30, 20, 0 for 3, 2, 0) leave pure-strategy analysis, dominance and pure equilibria intact. **Cardinal utility** (von Neumann–Morgenstern) also carries the size of differences: lotteries and mixed strategies are compared by **expected utility**, so the numbers matter.

Example

Does Ayşe prefer Football for sure, or the lottery ½ Opera + ½ Home?

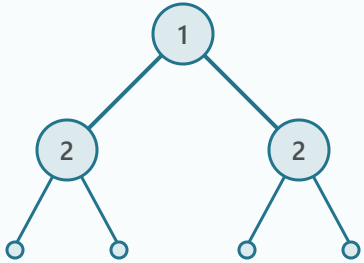
Utility (Opera, Football, Home)	Lottery	Sure Football	Choice
(3, 2, 0)	½·3 + ½·0 = 1.5	2	Football
(3, 1, 0): same order	1.5	1	**Lottery**
2u + 1 = (7, 5, 1)	4	5	Football

Same order, different numbers: the lottery decision can flip; a·u + b (a > 0) preserves it.

Common mistake: saying "4 is twice as good as 2", or using ordinal numbers in mixed-strategy calculations. Expected payoffs assume cardinal (vNM) utilities; with only an order, stick to pure strategies.

Information: Who Knows What?

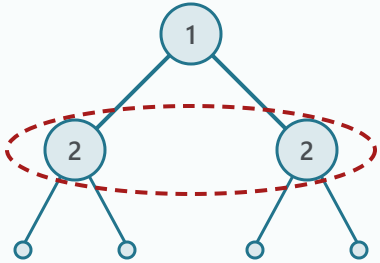
- What each player knows about the game, the payoffs and the moves made so far; it directly affects strategies and outcomes.



2 sees 1's choice

Perfect information

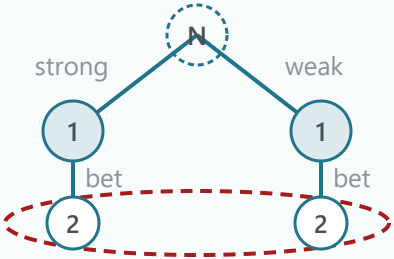
chess, tic-tac-toe, Nim



dashed: information set (2 can't tell)

Simultaneous moves

RPS, price competition



N: Nature deals; 2 can't see it

Imperfect information

poker, auctions

A dashed line is an **information set**: the player cannot tell which of these nodes it is at. **Perfect information** means seeing every past move, not knowing the future.

Example: Poker and Bluffing Strategies

Setup (Texas Hold'em)

- Each player has 2 hidden cards and tries to make the best hand with the open cards in the middle.
- You cannot see your opponents' hidden cards: information is private.
- Even with a strong hand, you try to infer your opponents' hands from their bets.

The game-theoretic view

- An opponent's behavior (betting, checking) carries **information** about them.
- A **bluff** is a misleading signal sent to hide the strength of a hand.
- A player who bets only with strong hands is easy to read.

A bluff is really a **mixed strategy**: betting with some probability on weak hands too keeps opponents from reading you. Poker programs (e.g. Libratus, 2017) beat professionals by computing such equilibria.

Normal Form (Strategic Form)

- Models situations where players decide **simultaneously** (or without knowing each other's choices).
- A set of players **N = {1, …, n}**; each player has a strategy set $\mathbf{S_i}$.
- **Strategy profile**: $s = (s_1, \ldots, s_n)$.
- **Utility function**: $u_i : S_1 \times \ldots \times S_n \to \mathbb{R}$.
- With two players, a **payoff matrix**: row = player 1, column = player 2, cell = (u_1, u_2).

		Player 2	
Player 1		**Left**	**Right**
Top		(4 , 3)	(1 , 2)
Bottom		(3 , 1)	(2 , 4)

S_1 = {Top, Bottom}, S_2 = {Left, Right}

profile s = (Top, Right): $u_1 = 1$, $u_2 = 2$

$G = (N, S_1, S_2, u_1, u_2)$, N = {1, 2}

Example: Rock-Paper-Scissors in Normal Form

- Two players (n = 2) choose Rock, Paper or Scissors **simultaneously**. Rock beats scissors, scissors beats paper, paper beats rock.

		Player 2		
Player 1	Rock	Paper	Scissors	
Rock	(0 , 0)	(−1 , 1)	(1 , −1)	Winner +1
Paper	(1 , −1)	(0 , 0)	(−1 , 1)	Loser −1
Scissors	(−1 , 1)	(1 , −1)	(0 , 0)	Tie 0

Every cell sums to 0 (a zero-sum game)

In no cell are both players happy with their choice: one of them always wants to switch.

Choosing a Strategy in Rock-Paper-Scissors

1. Play at random

Each round, pick each symbol with probability 1/3. Cannot be exploited: whatever the opponent does, the expected payoff is 0.

2. Play by history

If the opponent played Rock last round, play Paper this round.
Wins against a patterned opponent; but the rule itself is a pattern the opponent can exploit.

3. An unbalanced mix

E.g. Paper half the time, Scissors half the time.
If the opponent always plays Scissors, its expected payoff is **+1/2** and yours −1/2: this mix is **exploitable**.

Even in a simple game, strategy matters. Being unpredictable is not enough; you must be unpredictable with the right probabilities.

Zero-Sum Games

- Payoffs sum to zero in every profile: $\mathbf{u_1(s) + u_2(s) = 0}$. One player's gain is the other's loss.
- A constant sum (e.g. one shared point) works the same: a *constant-sum* game.
- Examples: RPS, Matching Pennies, chess (+1 / 0 / −1). Not: the Prisoner's Dilemma, price competition.
- One matrix suffices: the row player maximizes its worst case (**maximin**); the column player minimizes the row player's best case (**minimax**). (**Minimax search** applies this to game trees, from leaves to root.)

Matcher \ Mismatcher	Heads	Tails
Heads	(1 , −1)	(−1 , 1)
Tails	(−1 , 1)	(1 , −1)

since $u_2 = -u_1$, only u_1 is written

	Heads	Tails
Heads	1	−1
Tails	−1	1

A

A zero-sum game has **no common interest**: pure competition. In *non*-zero-sum games everyone may win (or lose); cooperation and competition coexist, and negotiation can change the outcome (trade talks, climate deals).

Example: A Price Competition Game

- Two firms sell similar products in the same market and set prices simultaneously: 100 TL or 120 TL.
- The cheaper one captures the customers; if both are expensive, margins stay high.
- Cells are profit units: (Firm 1, Firm 2).

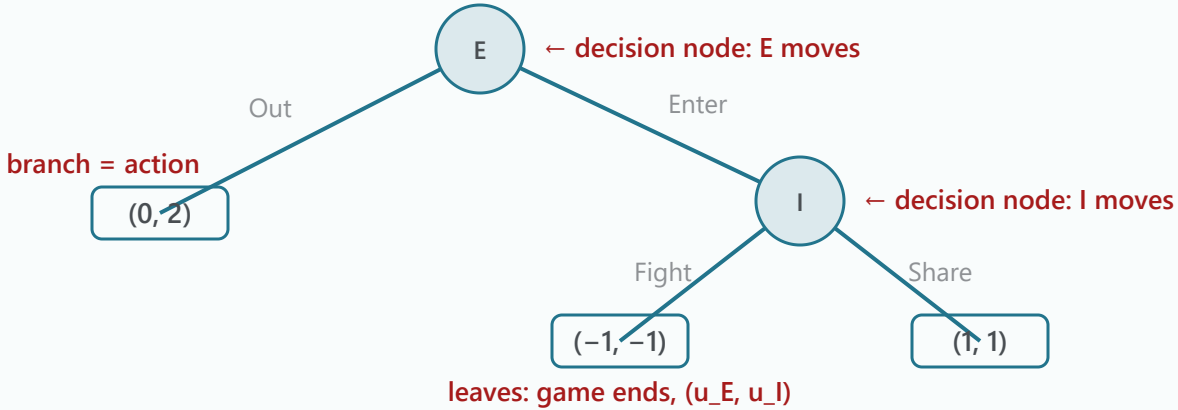
	Firm 2: 100 TL	**Firm 2: 120 TL**
Firm 1: 100 TL	**(5, 5)**	(8, 1)
Firm 1: 120 TL	(1, 8)	(6, 6)

Both at 100 TL earn 5 each; both at 120 TL earn 6 each.

If the other side picks 100, 5 > 1; if 120, 8 > 6: **100 TL is dominant** for each firm. Equilibrium (100, 100) → 5 each, while (120, 120) would give both 6. This is the structure of the Prisoner's Dilemma.

Extensive Form (Game Trees)

- A **game tree** shows the **order** of decisions and who knows what at each point; it models sequential interactions (chess, bargaining, market entry) naturally.



Entry game: the entrant (E) enters the market or stays out; if it enters, the incumbent (I) fights or shares the market. The tree shows the order and who moves at each decision point.

From Normal Form to Game Tree, Step by Step

		Firm 2
F1	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

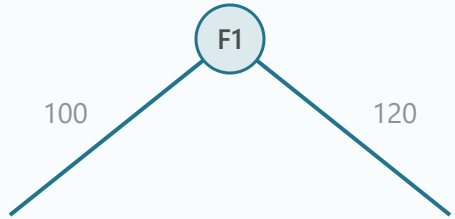
normal form

Start: the normal form of the price competition game. We will draw the same game as a **game tree**.

From Normal Form to Game Tree, Step by Step

	Firm 2	
F1	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

normal form

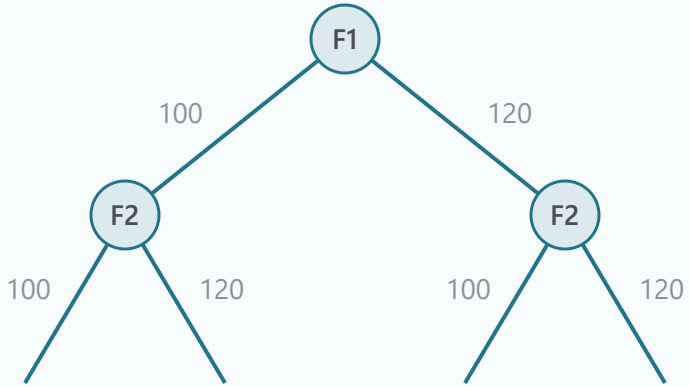


Root: Firm 1's decision. Each branch is an action: 100 TL or 120 TL.

From Normal Form to Game Tree, Step by Step

	Firm 2	
F1	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

normal form



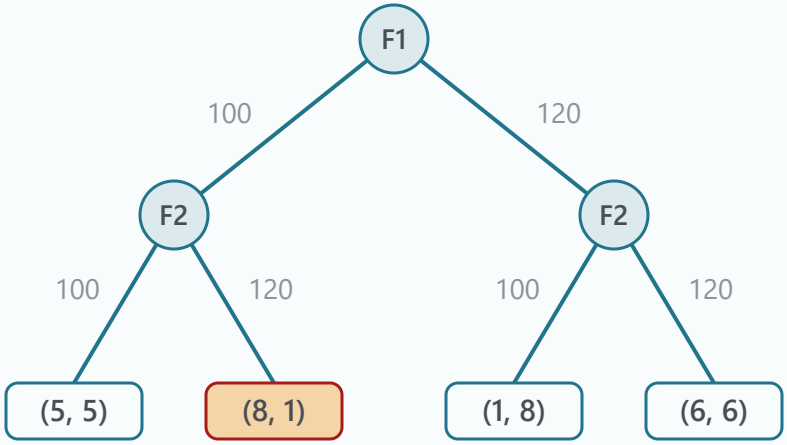
At the end of each branch is a decision node for Firm 2; it also picks 100 or 120.

From Normal Form to Game Tree, Step by Step

Firm 2

F1	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

normal form

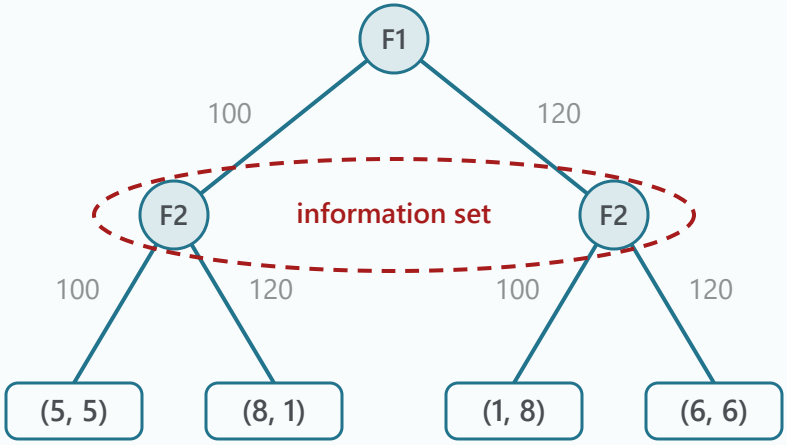


Leaves are outcomes: the four paths match the four cells of the matrix. E.g. path 100–120 is cell (8, 1).

From Normal Form to Game Tree, Step by Step

F1 \ Firm 2	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

normal form

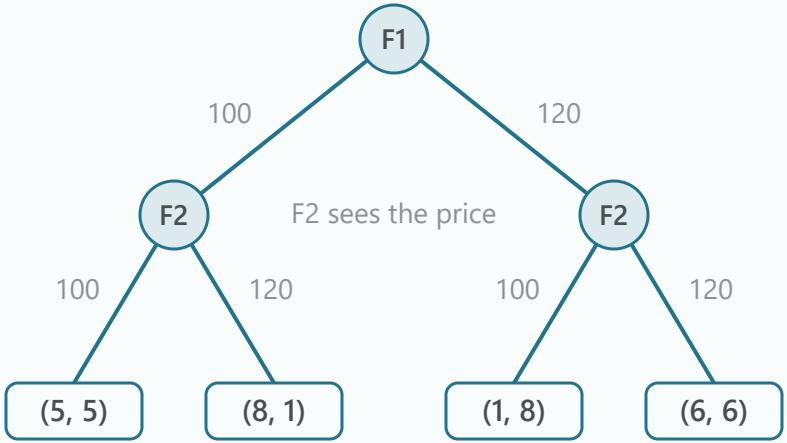


But the firms set prices **simultaneously**: F2 does not see F1's choice. The dashed **information set** shows this; the tree is the same game as the matrix.

From Normal Form to Game Tree, Step by Step

Firm 2		
F1	100 TL	120 TL
100	(5, 5)	(8, 1)
120	(1, 8)	(6, 6)

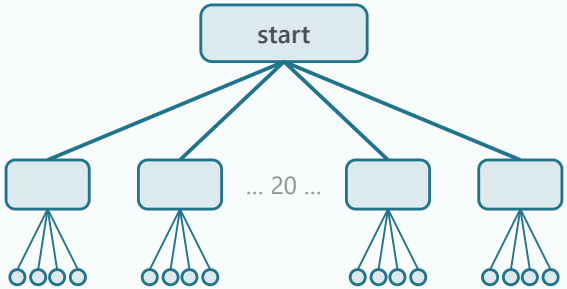
normal form



Remove the information set and F2 sees F1's price first: now the game is **sequential**, solved by backward induction.

Example: Chess Is an Extensive-Form Game

- Two players move in turn; each move is a branch, each position a node.
- **Perfect information**: both players see the board and all past moves.
- The result is a win, a loss **or a draw** (stalemate, threefold repetition, the 50-move rule).
- Zermelo (1913): in such finite games either one side can force a win or both can force a draw; but the tree is so large that we do not know which.



White: 20 moves · Black: 20 replies

$20 \times 20 = 400$ positions; tree $\approx 10^{120}$ games

Chess programs search only part of the tree (minimax + pruning). The trees of small games like tic-tac-toe, Nim and "Don't Say 21" can be solved completely.

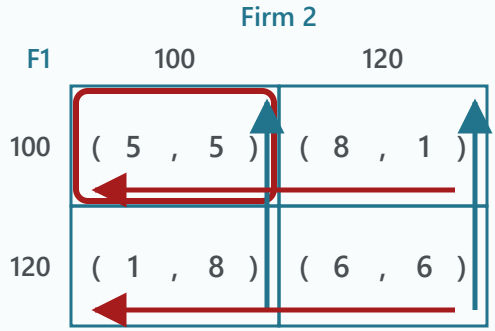
Classifying Games

Dimension	One end	Other end	Question	Example
Timing	Simultaneous	Sequential	Do I see the other's move first?	sealed bid / entry
Agreements	Cooperative	Non-cooperative	Binding agreements possible?	joint venture / Prisoner's Dilemma
Total payoff	Zero-sum	Non-zero-sum	One's gain, the other's loss?	penalty / climate deal
Symmetry	Symmetric	Asymmetric	Same game if players swap?	Stag Hunt / Battle of the Sexes
Information	Perfect	Imperfect	Do I see all past moves?	Connect Four / poker
Horizon	Finite	Infinite	Is there a definite end?	Centipede / price war

- **Complete information** is separate: payoff functions known to all; else **incomplete information** (Bayesian games). **Symmetry:** $u_1(a, b) = u_2(b, a)$. **Horizon:** future payoffs weighted by $\delta \in (0, 1)$: $\Sigma\, \delta^t\, u_t$.
- **Agreements:** cooperative games analyze coalitions (Shapley value, core), non-cooperative games individual players (Nash equilibrium). Chess: sequential, non-cooperative, zero-sum, perfect information, finite.

Equilibrium and Nash Equilibrium

- **Best response**: the strategy giving player i the highest payoff when the other players' strategies (s_{-i} for short) are fixed.
- **Nash equilibrium**: a profile (written s*) in which every player's strategy is a best response to the others; no one wants to deviate *unilaterally* (Nash, 1950).



blue ↑: Firm 1 gains by deviating · red ←: Firm 2 deviates
all arrows flow into (100, 100): nobody wants to leave

What it says

With the others fixed, deviating does not pay; nobody has regrets. Deviation arrows lead into the equilibrium, never out of it.

What it does not say

That the equilibrium is good for everyone: (120, 120) would give both 6. The Prisoner's Dilemma has the same structure.

The Prisoner's Dilemma

- Two suspects (Sam and John) are questioned in separate rooms and cannot talk to each other.
- Each has two options: **stay silent**, or betray the partner and **confess**.

Situation	Sam	John
Both silent	1 year	1 year
Sam confesses, John silent	free	3 years
Sam silent, John confesses	3 years	free
Both confess	2 years	2 years

Terminology: **Cooperate** = cooperate with the partner = **stay silent**. **Defect** = confess (cooperate with the police). In short: "Silent" and "Defect".

The Prisoner's Dilemma

Payoff Matrix of Sam and John

- Payoff = −(years in prison): less time, higher payoff.
- In each cell the first number is **Sam's** (row) payoff, the second **John's** (column).
- Both silent: both −1.
- Sam silent, John defects: Sam −3, John 0.

	John: Silent	**John: Defect**
Sam: Silent	a (−1, −1)	b (−3, 0)
Sam: Defect	c (0, −3)	d (−2, −2)

The cells are labeled a, b, c, d.

Question: what happens if each prisoner minimizes his own sentence?

Why Is Defection Dominant? Case Analysis

Case 1: John stays silent

Sam silent: −1 (cell a) · Sam defects: 0 (cell c) ⇒ **defecting is better** (0 > −1).

Why Is Defection Dominant? Case Analysis

Case 1: John stays silent

Sam silent: −1 (cell a) · Sam defects: 0 (cell c) ⇒ **defecting is better** (0 > −1).

Case 2: John defects

Sam silent: −3 (cell b) · Sam defects: −2 (cell d) ⇒ **defecting is better** (−2 > −3).

Why Is Defection Dominant? Case Analysis

Case 1: John stays silent

Sam silent: −1 (cell a) · Sam defects: 0 (cell c) ⇒ **defecting is better** (0 > −1).

Case 2: John defects

Sam silent: −3 (cell b) · Sam defects: −2 (cell d) ⇒ **defecting is better** (−2 > −3).

Whatever John does, defecting brings a lighter sentence: defection is a **dominant strategy**. No need to know John's probabilities; the argument "it lowers the average sentence" is wrong.

Why Is Defection Dominant? Case Analysis

Case 1: John stays silent

Sam silent: −1 (cell a) · Sam defects: 0 (cell c) ⇒ **defecting is better** (0 > −1).

Case 2: John defects

Sam silent: −3 (cell b) · Sam defects: −2 (cell d) ⇒ **defecting is better** (−2 > −3).

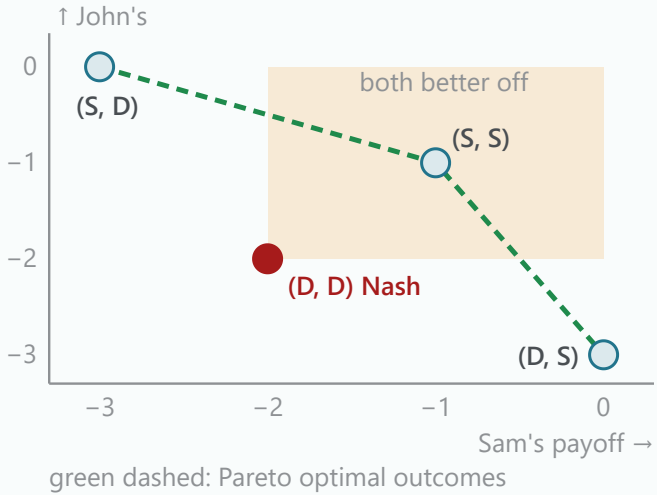
Whatever John does, defecting brings a lighter sentence: defection is a **dominant strategy**. No need to know John's probabilities; the argument "it lowers the average sentence" is wrong.

John faces the same dilemma; defection is dominant for him too. The outcome is cell **d**: **(−2, −2)**, 2 years each. Had both stayed silent (cell a) they would get **(−1, −1)**: confessing is **not** "best for both"; individually optimal choices make both worse off.

The Lesson: Individual Rationality, Collective Harm

- The game's unique Nash equilibrium is (Defect, Defect) = (−2, −2).
- (Silent, Silent) = (−1, −1) is better for both: the equilibrium is **not Pareto optimal**.
- Even if they make a deal, it is not binding (a non-cooperative game): each has an incentive to break it.

Lesson: everyone acting rationally in their own interest does not guarantee a good outcome for society.



green dashed: Pareto optimal outcomes

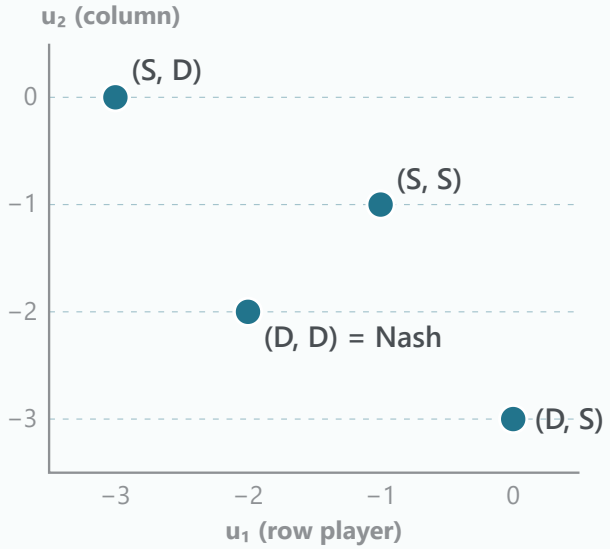
Where does it appear?

Price wars, arms races, climate agreements, doping in sports, overuse of common resources.

Ways out

Repeated play and reputation (Tit-for-Tat), contracts and incentive design.

Pareto Optimality in Payoff Space



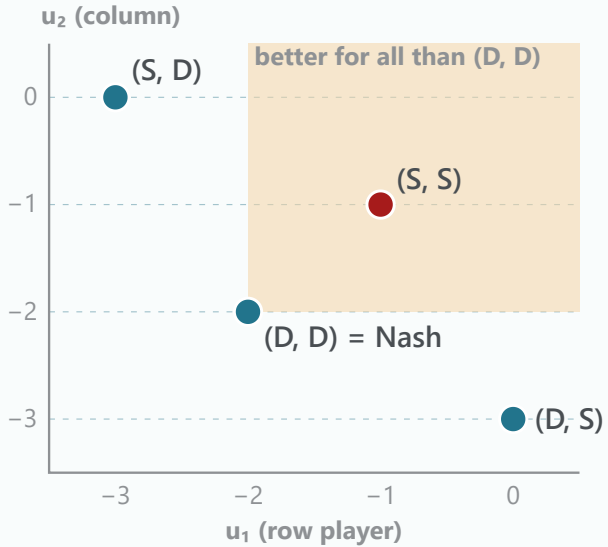
Each outcome is a point (u_1, u_2). The four outcomes of the Prisoner's Dilemma (S: Silent, D: Defect), payoff = −(years in prison).

Step 1 / 3

- **Pareto dominance:** x dominates y ⇔ u(x) ≥ u(y) for everyone, with > for at least one.
- **Pareto optimal:** no outcome dominates it; nobody can be made better off without making someone worse off.
- In the plot: a point is Pareto optimal if no other outcome lies in its upper-right quadrant.
- Pareto optimality **is not fairness**: (S, D) is Pareto optimal too, yet one prisoner serves 3 years.

Equilibrium (stability) and efficiency are separate questions; the loss between them is measured by the Price of Anarchy.

Pareto Optimality in Payoff Space

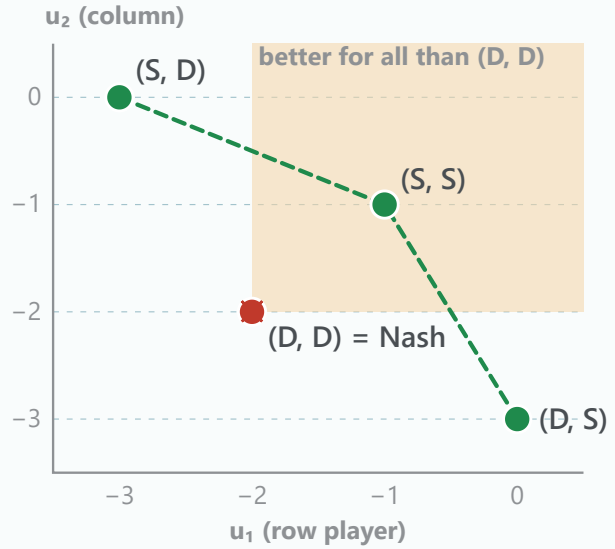


Nash equilibrium (D, D) = (−2, −2). In the orange region there is an outcome better for both: **(S, S)**.

- **Pareto dominance:** x dominates y ⇔ u(x) ≥ u(y) for everyone, with > for at least one.
- **Pareto optimal:** no outcome dominates it; nobody can be made better off without making someone worse off.
- In the plot: a point is Pareto optimal if no other outcome lies in its upper-right quadrant.
- Pareto optimality **is not fairness**: (S, D) is Pareto optimal too, yet one prisoner serves 3 years.

Equilibrium (stability) and efficiency are separate questions; the loss between them is measured by the Price of Anarchy.

Pareto Optimality in Payoff Space



Pareto optimal: (S, S), (S, D), (D, S) (green). The unique Nash equilibrium (D, D) is Pareto dominated.

- **Pareto dominance:** x dominates y ⇔ u(x) ≥ u(y) for everyone, with > for at least one.
- **Pareto optimal:** no outcome dominates it; nobody can be made better off without making someone worse off.
- In the plot: a point is Pareto optimal if no other outcome lies in its upper-right quadrant.
- Pareto optimality **is not fairness**: (S, D) is Pareto optimal too, yet one prisoner serves 3 years.

Equilibrium (stability) and efficiency are separate questions; the loss between them is measured by the Price of Anarchy.

CLASSIC 2×2 GAMES

Six Classic 2×2 Games and Their Pure Equilibria

Game	Actions (1 / 2)	(1, 1)	(1, 2)	(2, 1)	(2, 2)	Pure Nash equilibria
Matching Pennies	Heads / Tails	(1, −1)	(−1, 1)	(−1, 1)	(1, −1)	none
Chicken	Swerve / Straight	(0, 0)	(−1, 1)	(1, −1)	(−10, −10)	(Swerve, Straight), (Straight, Swerve)
Battle of the Sexes	Opera / Football	(3, 2)	(0, 0)	(0, 0)	(2, 3)	(Opera, Opera), (Football, Football)
Stag Hunt	Stag / Hare	(4, 4)	(0, 3)	(3, 0)	(3, 3)	(Stag, Stag), (Hare, Hare)
Pure Coordination	Party / Home	(10, 10)	(0, 0)	(0, 0)	(5, 5)	(Party, Party), (Home, Home)
Free Money	Heads / Tails	(1, 1)	(0, 0)	(0, 0)	(0, 0)	(Heads, Heads) strict; (Tails, Tails) weak

(1, 2): the row player plays action 1, the column player action 2; a cell holds (row player's, column player's) payoffs.

Finding pure equilibria: in each column mark the row player's **best response**, in each row the column player's; a cell with both marks is a pure Nash equilibrium.

CLASSIC 2×2 GAMES

Six Games: Conflict or Coordination?

Matching Pennies

Interests are fully opposed; in every cell the loser deviates. Any predictable tendency is exploited: the only way out is to play at random (a mixed strategy).

Chicken

Anti-coordination: players want different actions. Each of the two equilibria favors one driver; whoever credibly shows they cannot back down (commitment) wins.

Battle of the Sexes

Meeting is a common interest; where to meet is the conflict. No dominant strategy; the fix is communication, convention or taking turns.

Stag Hunt

A trust dilemma: (Stag, Stag) is Pareto optimal and an equilibrium, but Hare guarantees 3 in every case. Unlike the Prisoner's Dilemma, cooperation is an equilibrium.

Pure Coordination

No conflict of interest; yet the bad equilibrium (Home, Home) also sustains itself. Players gravitate toward the salient choice, the focal point (Schelling).

Free Money

(Heads, Heads) is a strict equilibrium: a deviator loses. (Tails, Tails) is a weak equilibrium: deviating neither gains nor loses; Heads is weakly dominant.

Level-k Reasoning: 45 → 30 → 20 ...

Level of reasoning	Assumption	Guess
Level 0	Everyone writes at random, average 45	—
Level 1	Others are level 0	2/3·45 = 30
Level 2	Others are level 1	2/3·30 = 20
Level 3	Others are level 2	≈ 13.3
Level ∞	Everyone fully rational	**0**

- "The average will be 45, so 2/3 of it is 30" is only **level 1** reasoning.
- Someone who expects others to say 30 writes lower (e.g. 20); someone who expects everyone to go too low may shift up a bit.
- The best guess depends on your belief about the levels at which others reason.
- A level is how many steps a player thinks ahead; do not confuse it with the level of knowledge, which counts how many times "everyone knows" is nested.

Code Trace: Level-k Reasoning and the Best Guess



Level 0 writes at random (average 45). Level k thinks everyone is level k−1 and writes 2/3 of their guess.

Step 1 / 10

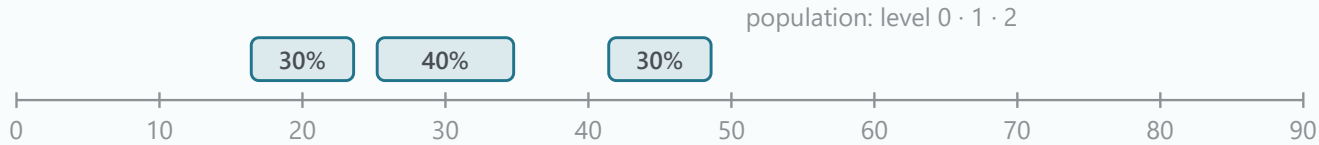
```
def guess(k, L0=45):     # level-k
    g = L0
    for _ in range(k):
        g = g * 2 / 3
    return g
share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
avg = sum(p * guess(k) for k, p in share.items())
target = avg * 2 / 3
for k in range(5):
    print(k, round(guess(k), 1),
          round(abs(guess(k) - target), 1))
```

VARIABLES

`L0 = 45`

OUTPUT

Code Trace: Level-k Reasoning and the Best Guess



Belief: 30% of the class reasons at level 0, 40% at level 1, 30% at level 2.

Step 2 / 10

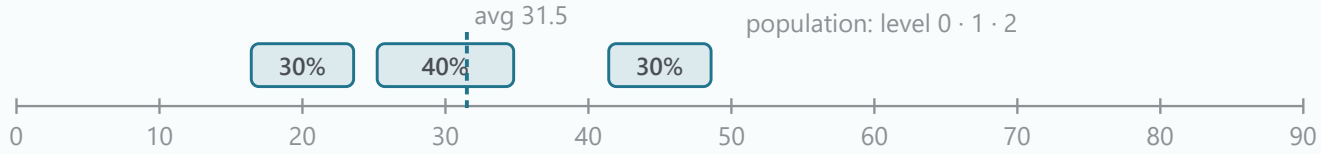
```
1  def guess(k, L0=45):     # level-k
2      g = L0
3      for _ in range(k):
4          g = g * 2 / 3
5      return g
▶ share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7  avg = sum(p * guess(k) for k, p in share.items())
8  target = avg * 2 / 3
9  for k in range(5):
10     print(k, round(guess(k), 1),
11           round(abs(guess(k) - target), 1))
```

VARIABLES

`share = {0: 0.3, 1: 0.4, 2: 0.3}`

OUTPUT

Code Trace: Level-k Reasoning and the Best Guess



Expected average: 0.3·45 + 0.4·30 + 0.3·20 = **31.5**.

Step 3 / 10

```
1  def guess(k, L0=45):      # level-k
2      g = L0
3      for _ in range(k):
4          g = g * 2 / 3
5      return g
6  share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
▶  avg = sum(p * guess(k) for k, p in share.items())
8  target = avg * 2 / 3
9  for k in range(5):
10     print(k, round(guess(k), 1),
11           round(abs(guess(k) - target), 1))
```

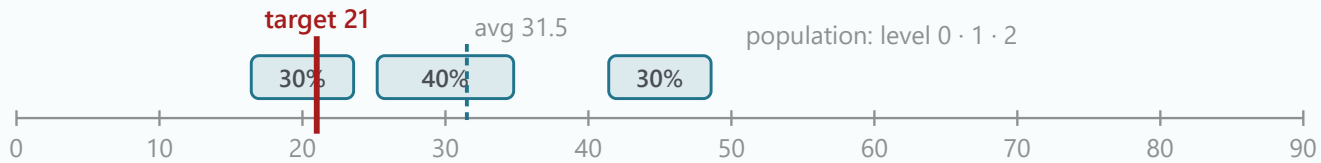
VARIABLES

`share = {0: 0.3, 1: 0.4, 2: 0.3}`

`avg = 31.5`

OUTPUT

Code Trace: Level-k Reasoning and the Best Guess



The winning number is 2/3 of the average: **target = 21**.

Step 4 / 10

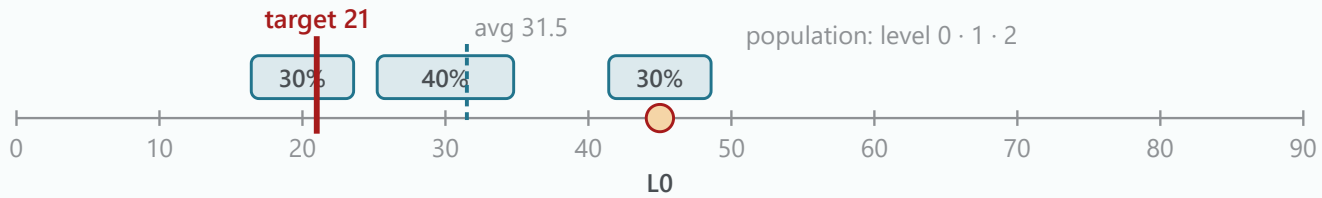
```
1  def guess(k, L0=45):     # level-k
2      g = L0
3      for _ in range(k):
4          g = g * 2 / 3
5      return g
6  share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7  avg = sum(p * guess(k) for k, p in share.items())
▶  target = avg * 2 / 3
9  for k in range(5):
10     print(k, round(guess(k), 1),
11           round(abs(guess(k) - target), 1))
```

VARIABLES

`avg = 31.5` `target = 21.0`

OUTPUT

Code Trace: Level-k Reasoning and the Best Guess



Level 0: guess 45, distance to target 24. Writing the average is far too high.

Step 5 / 10

```
1  def guess(k, L0=45):     # level-k
2      g = L0
3      for _ in range(k):
4          g = g * 2 / 3
5      return g
6  share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7  avg = sum(p * guess(k) for k, p in share.items())
8  target = avg * 2 / 3
▶  for k in range(5):
▶      print(k, round(guess(k), 1),
▶            round(abs(guess(k) - target), 1))
```

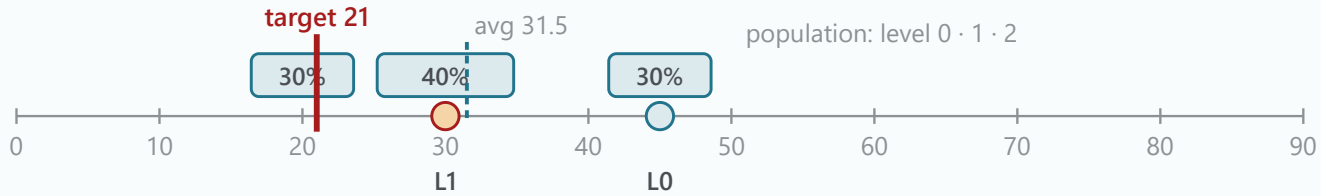
VARIABLES

`k = 0` `guess = 45.0` `dist = 24.0`

OUTPUT

```
0 45 24.0
```

Code Trace: Level-k Reasoning and the Best Guess



Level 1: guess 30, distance 9. The classic "average 45 → 30" answer: still far off.

Step 6 / 10

```
1 def guess(k, L0=45):     # level-k
2     g = L0
3     for _ in range(k):
4         g = g * 2 / 3
5     return g
6 share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7 avg = sum(p * guess(k) for k, p in share.items())
8 target = avg * 2 / 3
▶ for k in range(5):
▶     print(k, round(guess(k), 1),
▶           round(abs(guess(k) - target), 1))
```

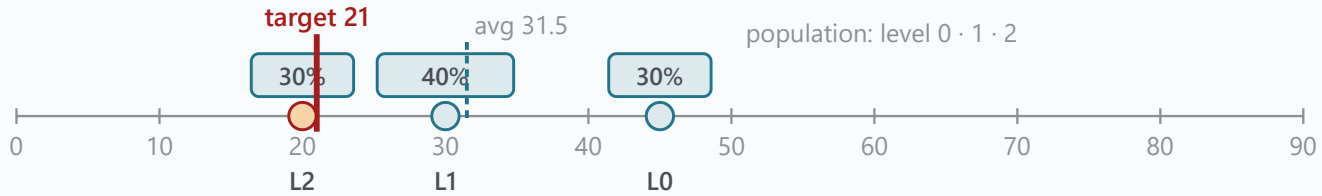
VARIABLES

`k = 1` `guess = 30.0` `dist = 9.0`

OUTPUT

```
0 45 24.0
1 30.0 9.0
```

Code Trace: Level-k Reasoning and the Best Guess



Level 2: guess 20, distance 1. The **closest** of the levels examined; in the large-population approximation the best response is the target itself: about 21.

Step 7 / 10

```
1 def guess(k, L0=45):     # level-k
2     g = L0
3     for _ in range(k):
4         g = g * 2 / 3
5     return g
6 share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7 avg = sum(p * guess(k) for k, p in share.items())
8 target = avg * 2 / 3
▶ for k in range(5):
▶     print(k, round(guess(k), 1),
▶           round(abs(guess(k) - target), 1))
```

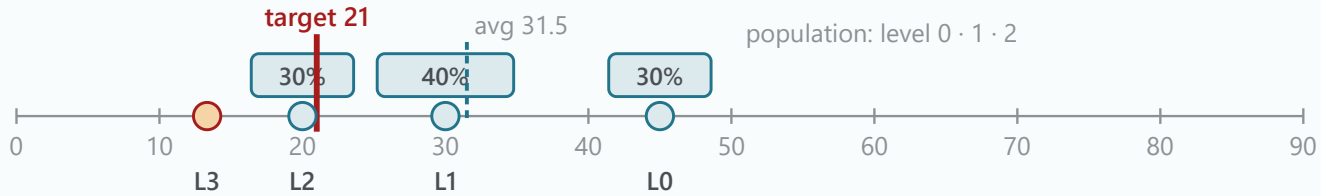
VARIABLES

`k = 2` `guess = 20.0` `dist = 1.0`

OUTPUT

```
0 45 24.0
1 30.0 9.0
2 20.0 1.0
```

Code Trace: Level-k Reasoning and the Best Guess



Level 3: guess 13.3, distance 7.7. Thinking one level too far loses as well.

Step 8 / 10

```
1 def guess(k, L0=45):     # level-k
2     g = L0
3     for _ in range(k):
4         g = g * 2 / 3
5     return g
6 share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7 avg = sum(p * guess(k) for k, p in share.items())
8 target = avg * 2 / 3
▶ for k in range(5):
▶     print(k, round(guess(k), 1),
▶           round(abs(guess(k) - target), 1))
```

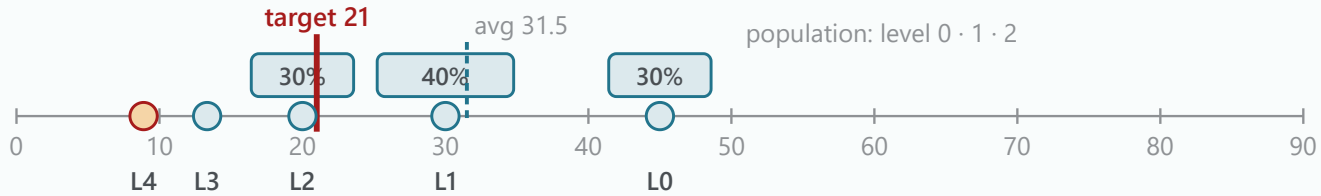
VARIABLES

`k = 3` `guess = 13.33` `dist = 7.67`

OUTPUT

```
0 45 24.0
1 30.0 9.0
2 20.0 1.0
3 13.3 7.7
```

Code Trace: Level-k Reasoning and the Best Guess



Level 4: guess 8.9, distance 12.1. The distance grows as we approach the fully rational answer (0).

Step 9 / 10

```
1 def guess(k, L0=45):     # level-k
2     g = L0
3     for _ in range(k):
4         g = g * 2 / 3
5     return g
6 share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7 avg = sum(p * guess(k) for k, p in share.items())
8 target = avg * 2 / 3
▶ for k in range(5):
▶     print(k, round(guess(k), 1),
▶           round(abs(guess(k) - target), 1))
```

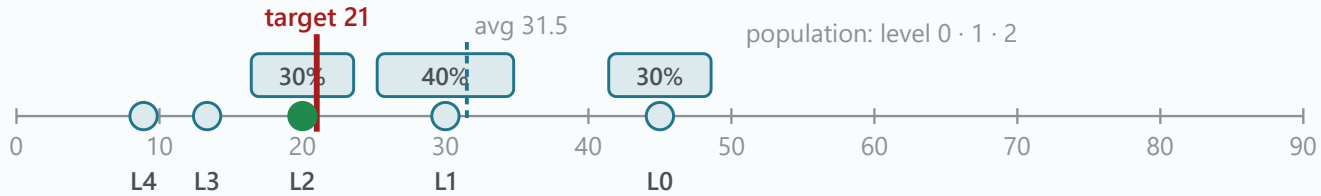
VARIABLES

`k = 4` `guess = 8.89` `dist = 12.11`

OUTPUT

```
0 45 24.0
1 30.0 9.0
2 20.0 1.0
3 13.3 7.7
4 8.9 12.1
```

Code Trace: Level-k Reasoning and the Best Guess



Result: against this population, 20 wins; the Nash equilibrium answer 0 loses. The best response depends on **beliefs**.

Step 10 / 10

```
1  def guess(k, L0=45):      # level-k
2      g = L0
3      for _ in range(k):
4          g = g * 2 / 3
5      return g
6  share = {0: 0.3, 1: 0.4, 2: 0.3}   # belief
7  avg = sum(p * guess(k) for k, p in share.items())
8  target = avg * 2 / 3
▶ for k in range(5):
10     print(k, round(guess(k), 1),
11           round(abs(guess(k) - target), 1))
```

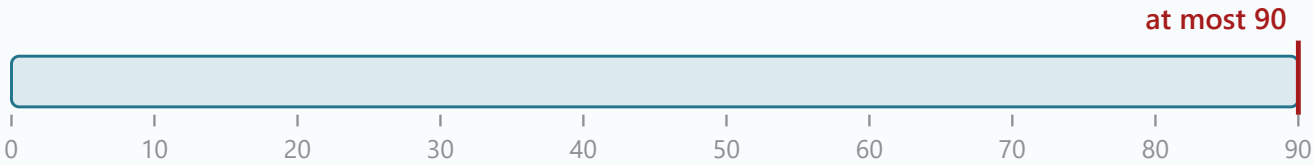
VARIABLES

`best = level 2` `guess = 20.0`

OUTPUT

```
0 45 24.0
1 30.0 9.0
2 20.0 1.0
3 13.3 7.7
4 8.9 12.1
```

Iterated Elimination: 90 → 60 → 40 → … → 0



Start: every guess lies between 0 and 90. Nothing eliminated yet.

Step 1 / 8

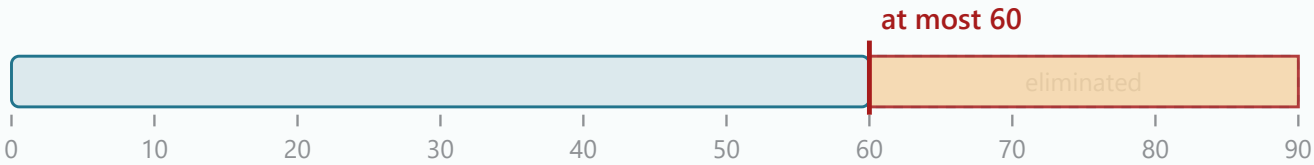
```
hi = 90                  # largest possible guess
for t in range(1, 7):
    # avg ≤ hi  ⇒  target ≤ (2/3)·hi
    hi = hi * 2 / 3
    print(t, round(hi, 1))
# hi = 90·(2/3)^k → 0 : unique equilibrium, all 0
```

VARIABLES

`hi = 90`

OUTPUT

Iterated Elimination: 90 → 60 → 40 → … → 0



The average is at most 90, so the target (2/3 × average) is at most 60. A guess above 60 is never better than 60 → eliminated.

Step 2 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

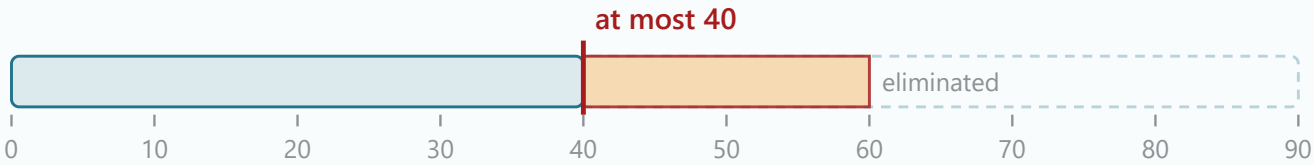
VARIABLES

`t = 1` `hi = 60.0`

OUTPUT

```
1 60.0
```

Iterated Elimination: 90 → 60 → 40 → … → 0



Everyone is rational and everyone knows it: nobody exceeds 60. Average ≤ 60 ⇒ target ≤ 40. Above 40 is eliminated.

Step 3 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

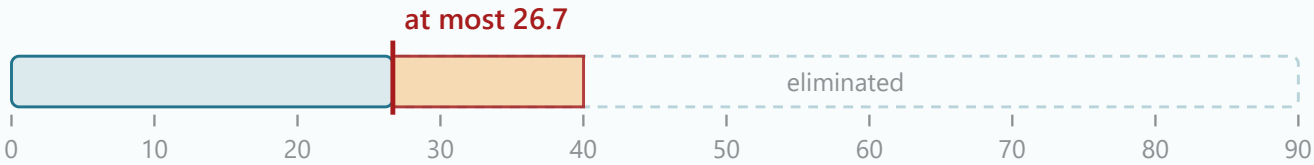
VARIABLES

`t = 2` `hi = 40.0`

OUTPUT

```
1 60.0
2 40.0
```

Iterated Elimination: 90 → 60 → 40 → … → 0



One more round: average ≤ 40 ⇒ target ≤ 26.7. This needs "everyone knows that everyone knows everyone is rational".

Step 4 / 8

```
1 hi = 90                # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

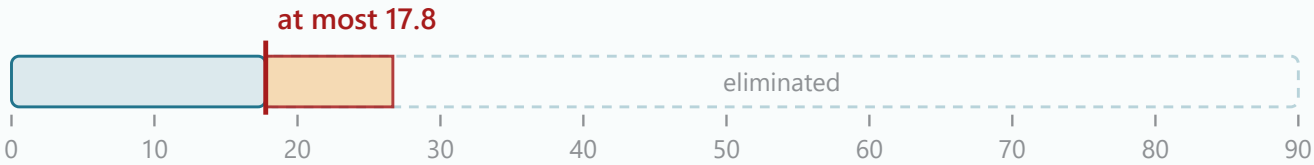
VARIABLES

`t = 3` `hi = 26.67`

OUTPUT

```
1 60.0
2 40.0
3 26.7
```

Iterated Elimination: 90 → 60 → 40 → … → 0



Average ≤ 26.7 ⇒ target ≤ 17.8. Each round multiplies the upper bound by 2/3.

Step 5 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

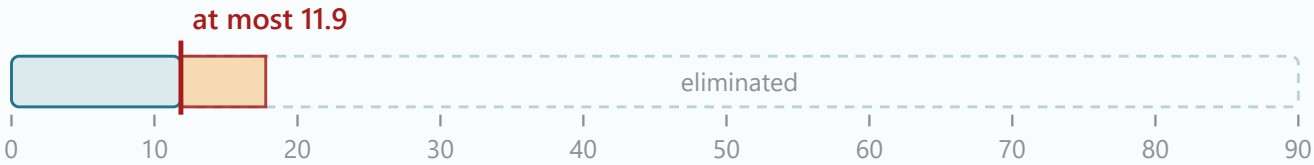
VARIABLES

`t = 4` `hi = 17.78`

OUTPUT

```
1 60.0
2 40.0
3 26.7
4 17.8
```

Iterated Elimination: 90 → 60 → 40 → … → 0



Average ≤ 17.8 ⇒ target ≤ 11.9. The range shrinks geometrically.

Step 6 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

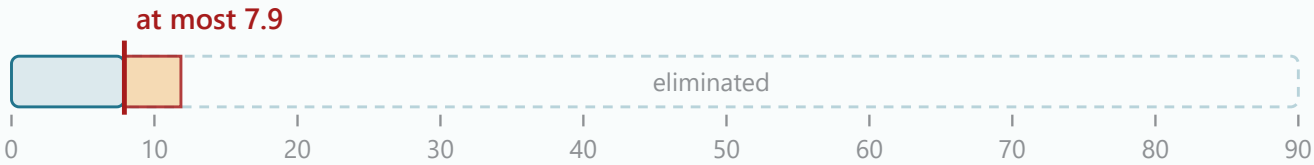
VARIABLES

`t = 5` `hi = 11.85`

OUTPUT

```
1 60.0
2 40.0
3 26.7
4 17.8
5 11.9
```

Iterated Elimination: 90 → 60 → 40 → … → 0



Average ≤ 11.9 ⇒ target ≤ 7.9. After k rounds the upper bound is $90 \cdot (2/3)^k$.

Step 7 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
▶     hi = hi * 2 / 3
▶     print(t, round(hi, 1))
6 # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

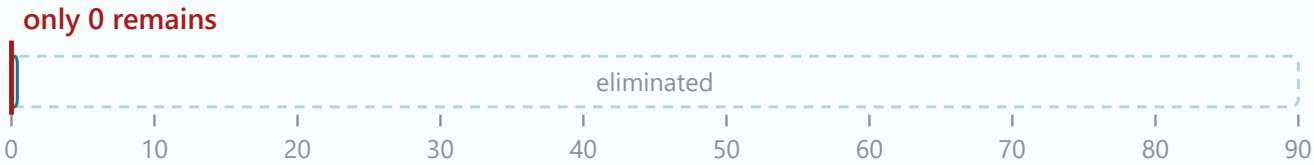
VARIABLES

`t = 6` `hi = 7.9`

OUTPUT

```
1 60.0
2 40.0
3 26.7
4 17.8
5 11.9
6 7.9
```

Iterated Elimination: 90 → 60 → 40 → … → 0



The process goes on forever and the upper bound converges to 0: the only profile surviving iterated elimination is **everyone saying 0**.

Step 8 / 8

```
1 hi = 90                  # largest possible guess
2 for t in range(1, 7):
3     # avg ≤ hi  ⇒  target ≤ (2/3)·hi
4     hi = hi * 2 / 3
5     print(t, round(hi, 1))
▶ # hi = 90·(2/3)^k → 0 : unique equilibrium, all
  0
```

VARIABLES

`t = …` `hi = 0 (limit)`

OUTPUT

```
1 60.0
2 40.0
3 26.7
4 17.8
5 11.9
6 7.9
```

The Unique Nash Equilibrium: Everyone Says 0

Check that it is an equilibrium

- If everyone says 0, the target is 0: all win (prize shared).
- A deviation to $x > 0$ makes the target $2x/(3n)$; the others' 0 is closer → the deviator loses.
- No other equilibrium: a loser can move to the target and win a share; if all tie, one can deviate and win alone. It is **unique**. (With integers only, "everyone 1" is also one for $n \geq 4$.)

And in practice?

- In experiments the winner is not 0: among Financial Times readers (1997, 0–100) it was **13**.
- Full rationality as **common knowledge** does not quite hold in reality.

Note: the equilibrium is easy to find (unique, pure: everyone 0). The hard part is the best response to *real* players. With repetition, guesses drift toward 0.

Back to the Game: Guess 2/3 of the Average

Theory's answer: 0

Step	Largest sensible guess
start	90
1	2/3 × 90 = 60
2	2/3 × 60 = 40
3	26.7
4	17.8
…	→ 0

Each step adds one more layer of "everyone is rational"; the end of infinitely many steps is the unique Nash equilibrium: everyone 0.

In practice: a few steps

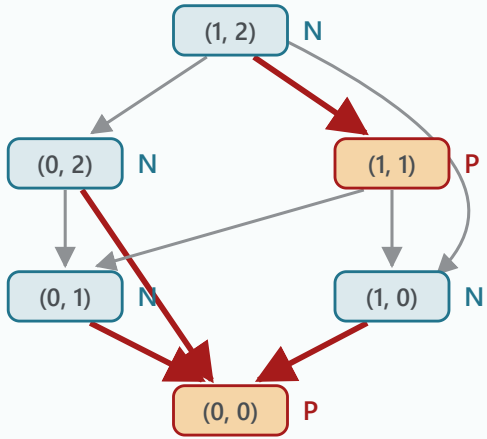
Level	Assumption	Guess
0	random, average 45	45
1	others level-0	30
2	others level-1	20
3	others level-2	13.3

In Nagel's (1995) experiments, first-round guesses clustered around 1–2 steps of reasoning; as the game was repeated, guesses fell toward 0.

Winning takes knowing not 0 but how far the class reasons: if the class plays level-1 and the average is 30, the target is 20, so level-2 wins.

Combinatorial Games and Zermelo's Theorem

- Two players move **in turn**; perfect information, no chance, and the game ends in finitely many moves.
- **Normal play**: whoever makes the last move wins (Nim; in Don't Say 21, whoever says 21 loses). Tic-tac-toe and chess also allow draws.
- **Zermelo (1913)**: in such a game either one side can force a win or both sides can force a draw.
- **Strategy stealing** (Hex): there are no draws; if the second player had a winning strategy, the first player could play it with an extra stone, so the first player wins.
- In games without draws (like Nim) every position is **N** (next player wins) or **P** (previous player wins): every move from P leads to N, and at least one move from N leads to P.

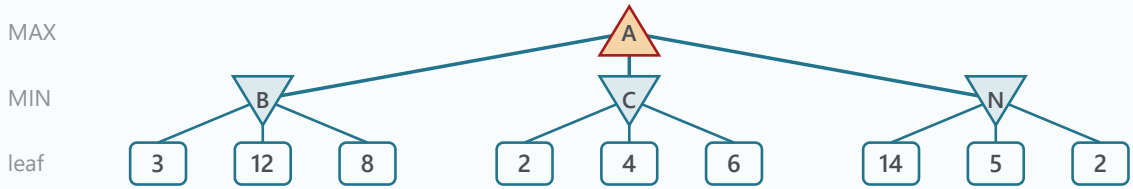


P (red): player to move loses · bold arrow: winning move

Two-heap Nim (1 and 2 chips): the player to move takes at least one chip from a single heap; whoever takes the last chip wins.

Labels are computed from the leaves up: (0, 0) has no moves → P. This is a special case of minimax.

Code Trace: Minimax on a Game Tree



At root A, MAX moves (wants large values); at B, C, D, MIN moves (wants small ones). Leaves are final payoffs.

Step 1 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
6    best = float("-inf") if maxi else float("inf")
7    for c in T[n]:
8      v = minimax(c, not maxi)
9      best = max(best, v) if maxi else min(best, v)
10   return best
▶ print(minimax("A", True))
```

VARIABLES

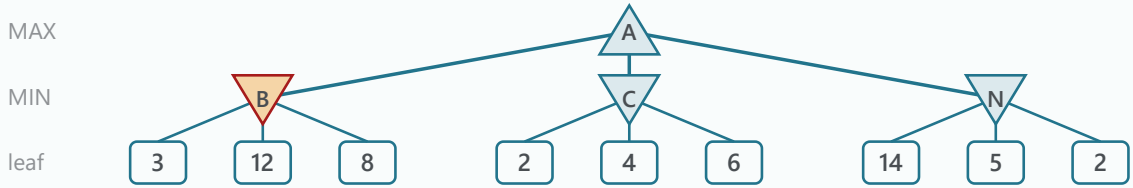
`n = "A"` `maxi = True`

CALL STACK

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



B is a MIN node: start with best = +∞ and evaluate the children in turn.

Step 2 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
▶    best = float("-inf") if maxi else float("inf")
▶    for c in T[n]:
8      v = minimax(c, not maxi)
9      best = max(best, v) if maxi else min(best, v)
10   return best
11 print(minimax("A", True))
```

VARIABLES

`n = "B"` `maxi = False` `best = inf`

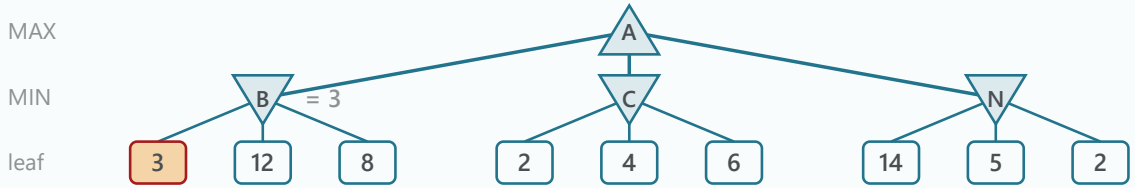
CALL STACK

`minimax("B", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 3 (int → returns at once). best = min(+∞, 3) = **3**.

Step 3 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "B"` `c = 3` `v = 3` `best = 3`

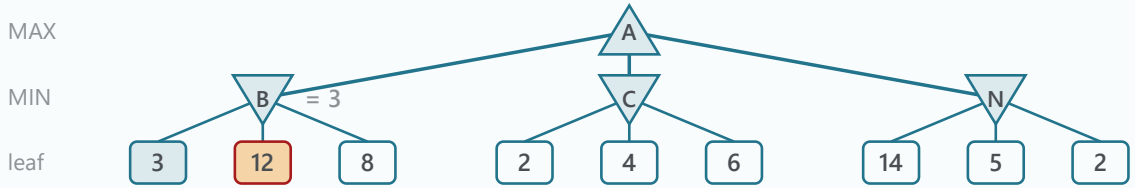
CALL STACK

`minimax("B", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 12: min(3, 12) = **3**.

Step 4 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "B"` `c = 12` `v = 12` `best = 3`

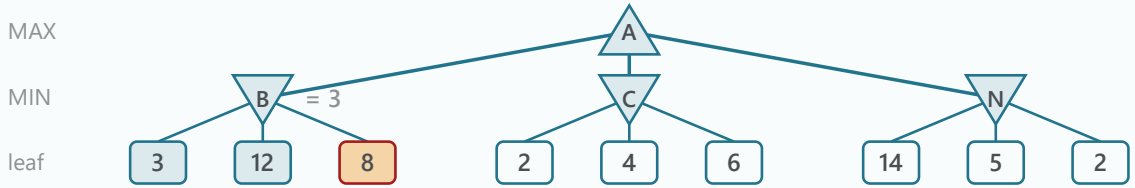
CALL STACK

`minimax("B", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 8: min(3, 8) = **3**.

Step 5 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "B"` `c = 8` `v = 8` `best = 3`

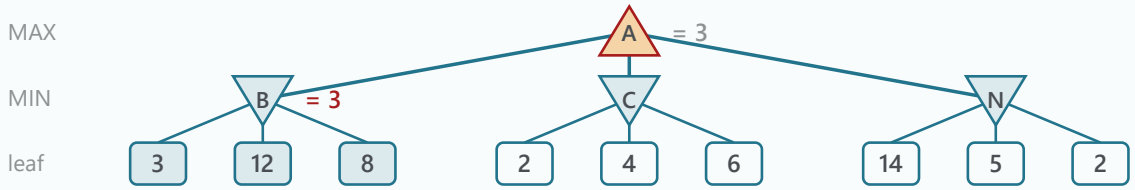
CALL STACK

`minimax("B", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



B returns 3. At A: max(−∞, 3) = **3**.

Step 6 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
6    best = float("-inf") if maxi else float("inf")
7    for c in T[n]:
8      v = minimax(c, not maxi)
▶      best = max(best, v) if maxi else min(best, v)
▶    return best
11 print(minimax("A", True))
```

VARIABLES

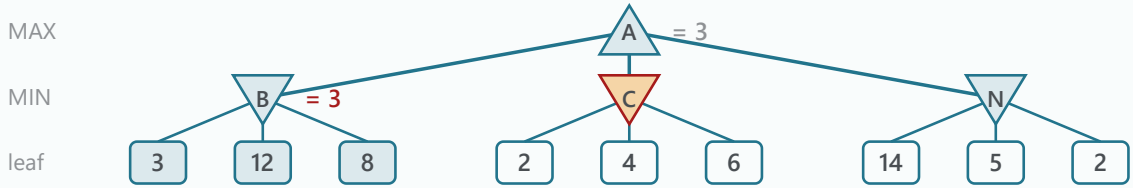
`n = "A"` `maxi = True` `c = "B"` `v = 3` `best = 3`

CALL STACK

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



A's next child is C (MIN). Again best = +∞.

Step 7 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
▶    best = float("-inf") if maxi else float("inf")
▶    for c in T[n]:
8      v = minimax(c, not maxi)
9      best = max(best, v) if maxi else min(best, v)
10   return best
11 print(minimax("A", True))
```

VARIABLES

`n = "C"` `maxi = False` `best = inf`

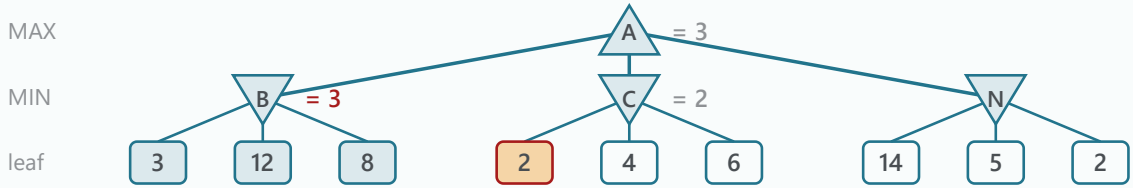
CALL STACK

`minimax("C", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 2 (int → returns at once). best = min(+∞, 2) = **2**.

Step 8 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "C"` `c = 2` `v = 2` `best = 2`

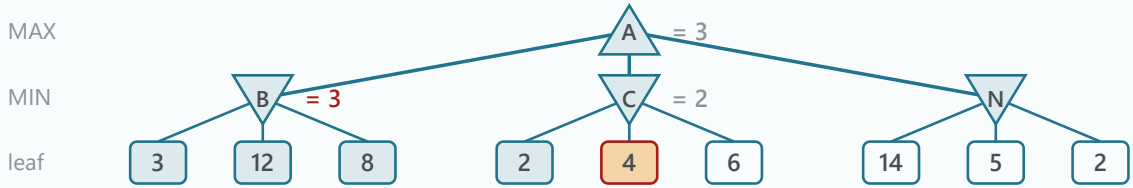
CALL STACK

`minimax("C", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 4: min(2, 4) = **2**.

Step 9 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "C"` `c = 4` `v = 4` `best = 2`

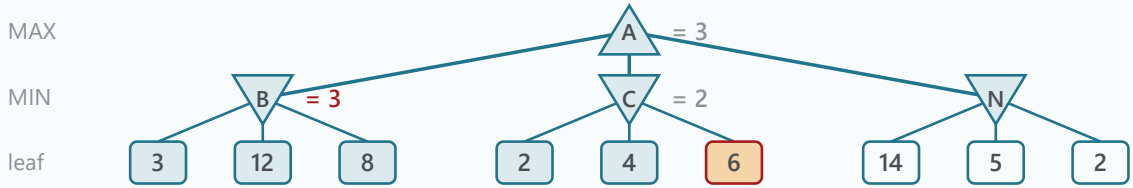
CALL STACK

`minimax("C", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 6: min(2, 6) = **2**.

Step 10 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "C"` `c = 6` `v = 6` `best = 2`

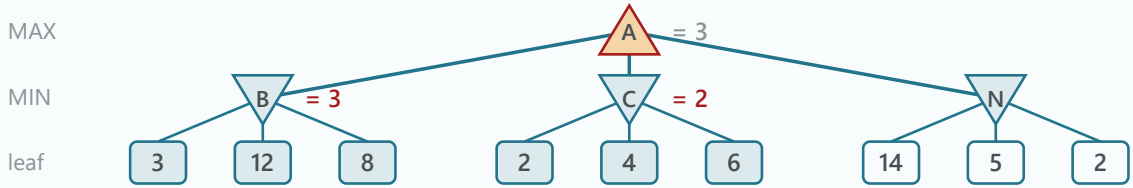
CALL STACK

`minimax("C", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



C returns 2. At A: max(3, 2) = **3**. For A, C is worse than B.

Step 11 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
6    best = float("-inf") if maxi else float("inf")
7    for c in T[n]:
8      v = minimax(c, not maxi)
▶      best = max(best, v) if maxi else min(best, v)
▶    return best
11 print(minimax("A", True))
```

VARIABLES

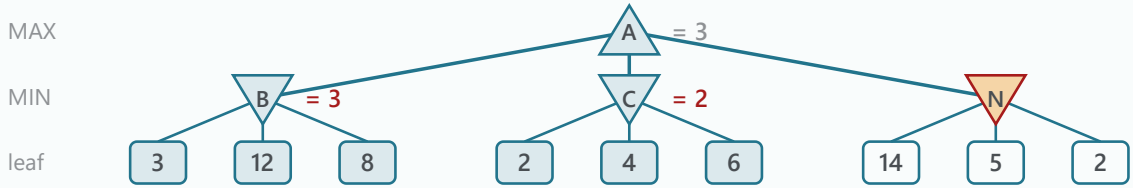
`n = "A"` `maxi = True` `c = "C"` `v = 2` `best = 3`

CALL STACK

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



A's next child is D (MIN). Again best = +∞.

Step 12 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
▶    best = float("-inf") if maxi else float("inf")
▶    for c in T[n]:
8      v = minimax(c, not maxi)
9      best = max(best, v) if maxi else min(best, v)
10   return best
11 print(minimax("A", True))
```

VARIABLES

`n = "D"` `maxi = False` `best = inf`

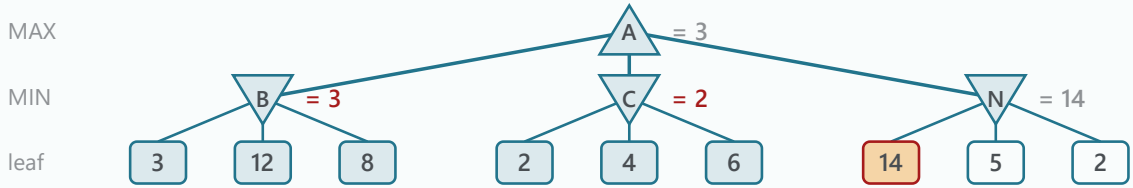
CALL STACK

`minimax("D", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 14 (int → returns at once). best = min(+∞, 14) = **14**.

Step 13 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
6    best = float("-inf") if maxi else float("inf")
7    for c in T[n]:
▶      v = minimax(c, not maxi)
▶      best = max(best, v) if maxi else min(best, v)
10   return best
11 print(minimax("A", True))
```

VARIABLES

`n = "D"` `c = 14` `v = 14` `best = 14`

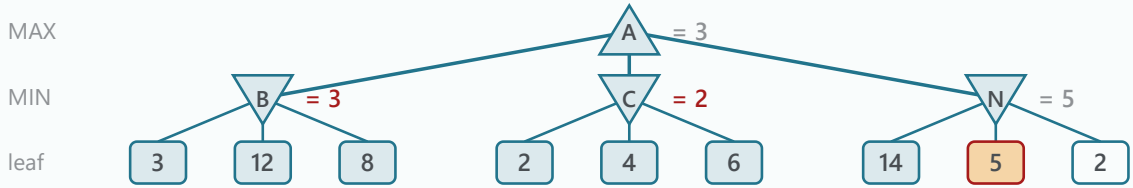
CALL STACK

`minimax("D", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 5: min(14, 5) = **5**.

Step 14 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "D"` `c = 5` `v = 5` `best = 5`

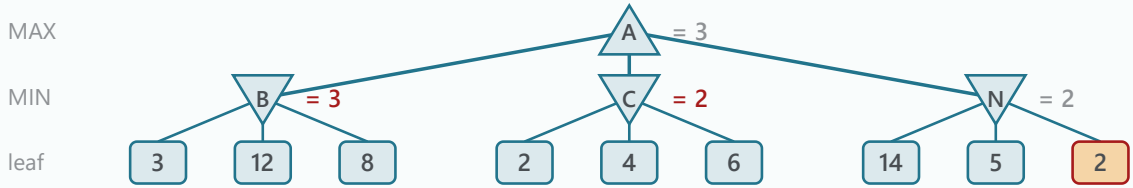
CALL STACK

`minimax("D", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



Leaf 2: min(5, 2) = **2**.

Step 15 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
▶     v = minimax(c, not maxi)
▶     best = max(best, v) if maxi else min(best, v)
10  return best
11 print(minimax("A", True))
```

VARIABLES

`n = "D"` `c = 2` `v = 2` `best = 2`

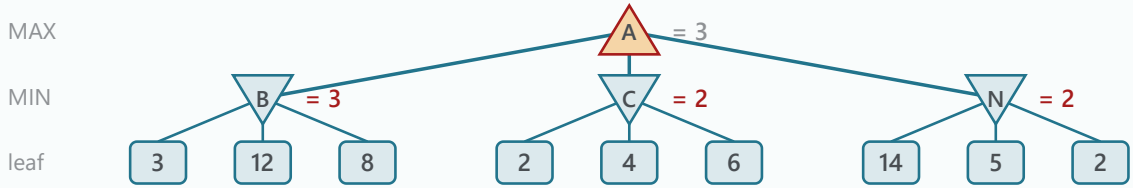
CALL STACK

`minimax("D", False)`

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



D returns 2. At A: max(3, 2) = **3**. The 14 under D looks tempting, but MIN picks 2.

Step 16 / 17

```
1  T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2       "C": [2, 4, 6], "D": [14, 5, 2]}
3  def minimax(n, maxi):
4    if isinstance(n, int):          # leaf
5      return n
6    best = float("-inf") if maxi else float("inf")
7    for c in T[n]:
8      v = minimax(c, not maxi)
▶      best = max(best, v) if maxi else min(best, v)
▶    return best
11 print(minimax("A", True))
```

VARIABLES

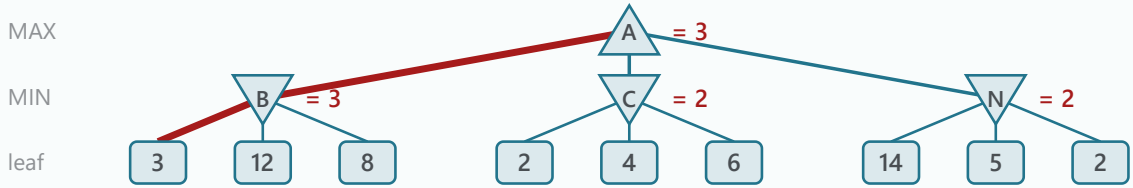
`n = "A"` `maxi = True` `c = "D"` `v = 2` `best = 3`

CALL STACK

`minimax("A", True)`

OUTPUT

Code Trace: Minimax on a Game Tree



The game value is 3: MAX picks B and is guaranteed at least 3 whatever MIN does. All 9 leaves were examined.

Step 17 / 17

```
1 T = {"A": ["B", "C", "D"], "B": [3, 12, 8],
2      "C": [2, 4, 6], "D": [14, 5, 2]}
3 def minimax(n, maxi):
4   if isinstance(n, int):          # leaf
5     return n
6   best = float("-inf") if maxi else float("inf")
7   for c in T[n]:
8     v = minimax(c, not maxi)
9     best = max(best, v) if maxi else min(best, v)
▶   return best
▶ print(minimax("A", True))
```

VARIABLES

`n = "A"` `best = 3`

CALL STACK

empty

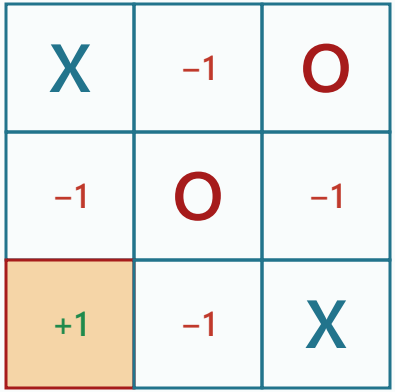
OUTPUT

```
3
```

COMBINATORIAL GAMES

Tic-Tac-Toe: The Whole Tree Can Be Searched, and It Is a Draw

- Tic-tac-toe's tree is small: a program enumerating all games finds **255,168** distinct games and **5,478** reachable positions.
- The minimax value of the empty board is **0**: with perfect play by both sides the game is a **draw**.
- In the position on the right, X wins only with the bottom-left square: it blocks O's diagonal and creates two threats at once (+1). Every other square loses to O (−1).
- In chess a full search is impossible ($\approx 10^{120}$ games): depth limits, evaluation functions and alpha-beta pruning.



X to move

+1: X wins

−1: O wins

bottom left: blocks O, sets two threats

Empty squares: minimax value after that move

Tic-tac-toe can be played perfectly by computing these values: against such a player you can at best draw. Alpha-beta pruning and depth-limited search: Minimax and Alpha-Beta Pruning.

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

Rule: starting from 1, players take turns saying 1, 2 or 3 consecutive numbers; whoever says 21 loses.
Base case: if the last number said is 20, the next player must say 21 → **loses** (L).

Step 1 / 16

```
TARGET = 21
wins = {TARGET - 1: False}   # c=20: lost
for c in range(TARGET - 2, -1, -1):
    wins[c] = any(not wins[c + k]
        for k in (1, 2, 3) if c + k < TARGET)
keys = [c for c in wins if not wins[c]]
print(sorted(keys))
```

VARIABLES

`TARGET = 21` `wins[20] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 19: saying 20 pushes the opponent into an L position → c = 19 is **W**.

Step 2 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 19` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 18: saying up to 20 pushes the opponent into an L position → c = 18 is **W**.

Step 3 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 18` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 17: saying up to 20 pushes the opponent into an L position → c = 17 is **W**.

Step 4 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 17` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 16: 17, 18, 19 are all W (the opponent wins) → c = 16 is **L**. A new key number!

Step 5 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 16` `wins[c] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 15: saying 16 pushes the opponent into an L position → c = 15 is **W**.

Step 6 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 15` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



c = 14: saying up to 16 pushes the opponent into an L position → c = 14 is **W**.

Step 7 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 14` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 13: saying up to 16 pushes the opponent into an L position → c = 13 is **W**.

Step 8 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 13` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 12: 13, 14, 15 are all W (the opponent wins) → c = 12 is **L**. A new key number!

Step 9 / 16

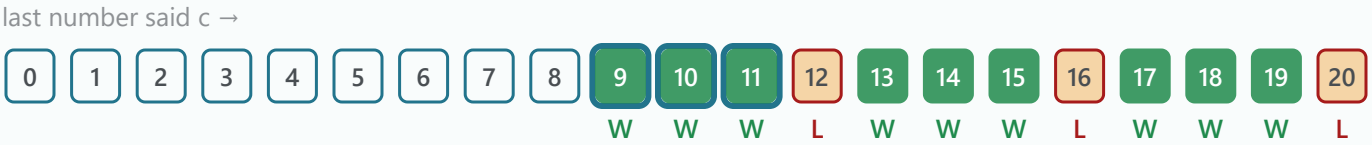
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 12` `wins[c] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 11…9: same pattern; all can reach the next L (12) → W.

Step 10 / 16

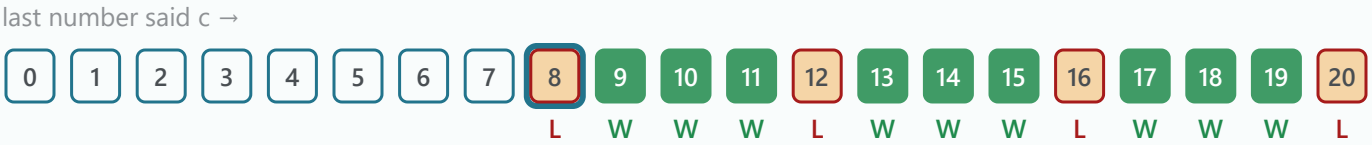
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 9` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 8: 9, 10, 11 are all W (the opponent wins) → c = 8 is **L**. A new key number!

Step 11 / 16

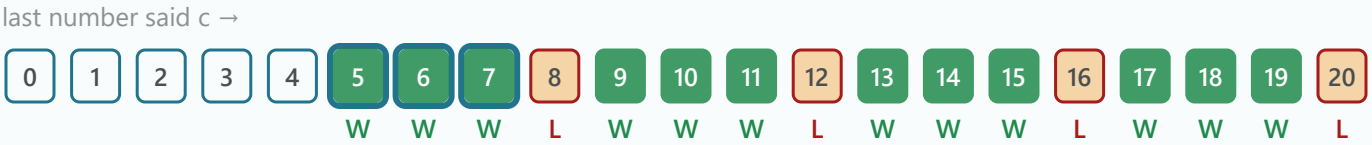
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 8` `wins[c] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 7…5: same pattern; all can reach the next L (8) → W.

Step 12 / 16

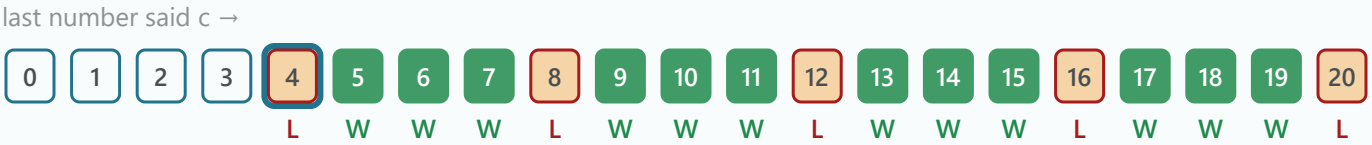
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 5` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 4: 5, 6, 7 are all W (the opponent wins) → c = 4 is **L**. A new key number!

Step 13 / 16

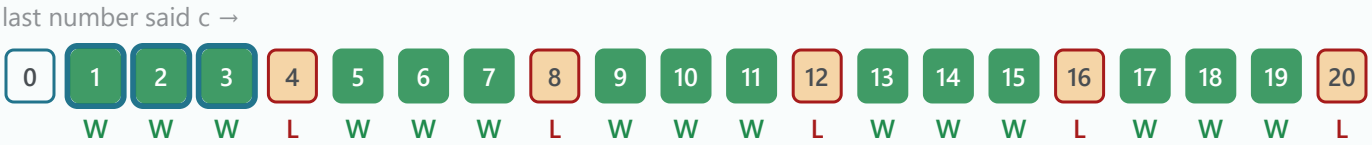
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 4` `wins[c] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



c = 3…1: same pattern; all can reach the next L (4) → W.

Step 14 / 16

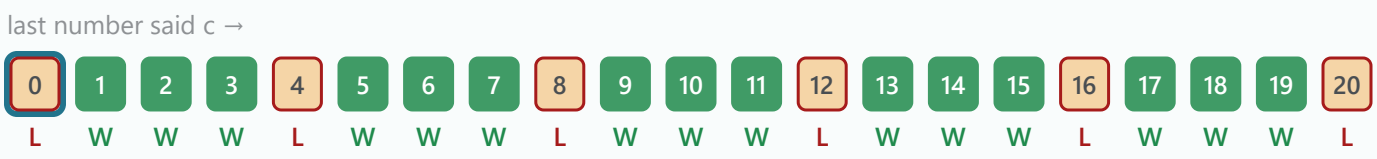
```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

VARIABLES

`c = 1` `wins[c] = True`

OUTPUT

Code Trace: Backward Induction in Don't Say 21



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

c = 0 (start of the game) is L: the first player loses. The second player always completes to 4, 8, 12, 16, 20.

Step 15 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
▶ for c in range(TARGET - 2, -1, -1):
▶     wins[c] = any(not wins[c + k]
▶         for k in (1, 2, 3) if c + k < TARGET)
6 keys = [c for c in wins if not wins[c]]
7 print(sorted(keys))
```

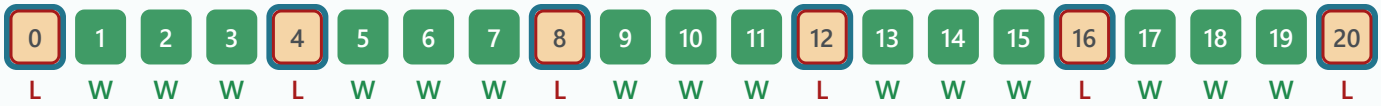
VARIABLES

`c = 0` `wins[c] = False`

OUTPUT

Code Trace: Backward Induction in Don't Say 21

last number said c →



W: player to move wins · L (red): player to move loses; a key number to leave the opponent

L positions are multiples of 4: always leave the opponent at 4k. If the opponent says x numbers, you say 4 − x.

Step 16 / 16

```
1 TARGET = 21
2 wins = {TARGET - 1: False}   # c=20: lost
3 for c in range(TARGET - 2, -1, -1):
4     wins[c] = any(not wins[c + k]
5         for k in (1, 2, 3) if c + k < TARGET)
▶ keys = [c for c in wins if not wins[c]]
▶ print(sorted(keys))
```

VARIABLES

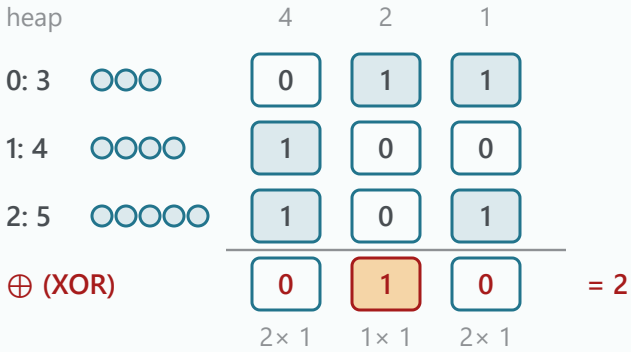
`keys = [0, 4, 8, 12, 16, 20]`

OUTPUT

```
[0, 4, 8, 12, 16, 20]
```

Nim: A Game Solved Completely by XOR

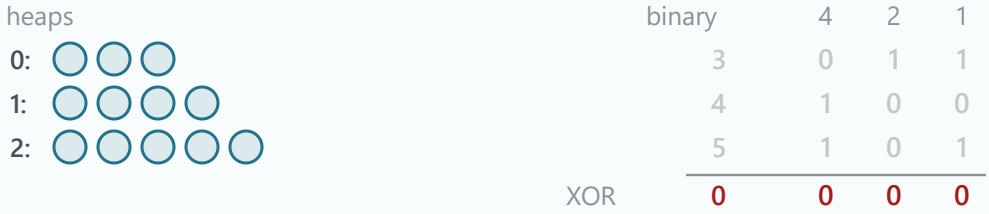
- **Rules**: several heaps of chips; the player to move takes at least one chip from a *single* heap. Whoever takes the last chip wins.
- **Nim-sum**: the bitwise XOR of the heap sizes (binary addition without carry): $3 \oplus 4 \oplus 5 = 2$.
- **Bouton (1901)**: a position is losing (P) $\Leftrightarrow$ its nim-sum is 0.
- Why? When the XOR is 0, every move changes at least one bit of one heap $\rightarrow$ XOR $\neq 0$. When XOR $= x \neq 0$, reducing a heap h that has x's leftmost 1 bit to $h \oplus x$ makes the XOR 0.



column with an odd number of 1s → 1 in XOR

Many other sequential games can be solved by reducing them to Nim.

Code Trace: The Winning Move in Nim (XOR)



Position [3, 4, 5]. Our move: compute the nim-sum (bitwise XOR).

Step 1 / 7

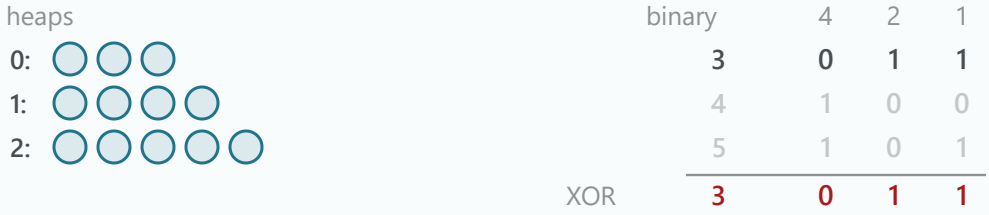
```
heaps = [3, 4, 5]
x = 0
for h in heaps:
    x ^= h              # nim-sum (XOR)
for i, h in enumerate(heaps):
    if h ^ x < h:       # can it shrink?
        heaps[i] = h ^ x
        print("heap", i, ":", h, "->", h ^ x)
        break
print(heaps)
```

VARIABLES

`heaps = [3, 4, 5]` `x = 0`

OUTPUT

Code Trace: The Winning Move in Nim (XOR)



x = 0 XOR 3 = **3** (binary 000 ⊕ 011 = 011).

Step 2 / 7

```
1  heaps = [3, 4, 5]
2  x = 0
▶  for h in heaps:
▶      x ^= h              # nim-sum (XOR)
5  for i, h in enumerate(heaps):
6      if h ^ x < h:       # can it shrink?
7          heaps[i] = h ^ x
8          print("heap", i, ":", h, "->", h ^ x)
9          break
10 print(heaps)
```

VARIABLES

`heaps = [3, 4, 5]` `h = 3` `x = 3`

OUTPUT

Code Trace: The Winning Move in Nim (XOR)

heaps

0: ◯◯◯

1: ◯◯◯◯

2: ◯◯◯◯◯

	binary	4	2	1
	3	0	1	1
	4	1	0	0
	5	1	0	1
XOR	7	1	1	1

if each column has an even count of 1s: XOR=0

x = 3 XOR 4 = **7** (binary 011 ⊕ 100 = 111).

Step 3 / 7

```
1  heaps = [3, 4, 5]
2  x = 0
▶  for h in heaps:
▶      x ^= h              # nim-sum (XOR)
5  for i, h in enumerate(heaps):
6      if h ^ x < h:       # can it shrink?
7          heaps[i] = h ^ x
8          print("heap", i, ":", h, "->", h ^ x)
9          break
10 print(heaps)
```

VARIABLES

`heaps = [3, 4, 5]` `h = 4` `x = 7`

OUTPUT

Code Trace: The Winning Move in Nim (XOR)

heaps

0: ◯◯◯
1: ◯◯◯◯
2: ◯◯◯◯◯

	binary	4	2	1
	3	0	1	1
	4	1	0	0
	5	1	0	1
XOR	2	0	1	0

if each column has an even count of 1s: XOR=0

x = 7 XOR 5 = **2** (binary 111 ⊕ 101 = 010).

Step 4 / 7

```
1  heaps = [3, 4, 5]
2  x = 0
▶  for h in heaps:
▶      x ^= h              # nim-sum (XOR)
5  for i, h in enumerate(heaps):
6      if h ^ x < h:       # can it shrink?
7          heaps[i] = h ^ x
8          print("heap", i, ":", h, "->", h ^ x)
9          break
10 print(heaps)
```

VARIABLES

`heaps = [3, 4, 5]` `h = 5` `x = 2`

OUTPUT

Code Trace: The Winning Move in Nim (XOR)

heaps

0: ○○○

1: ○○○○

2: ○○○○○

	binary	4	2	1
	3	0	1	1
	4	1	0	0
	5	1	0	1
XOR	2	0	1	0

if each column has an even count of 1s: XOR=0

Heap 0: 3 XOR 2 = 1 < 3 → take 2 chips from this heap. (The XOR's leftmost 1 bit is also set in 3.)

Step 5 / 7

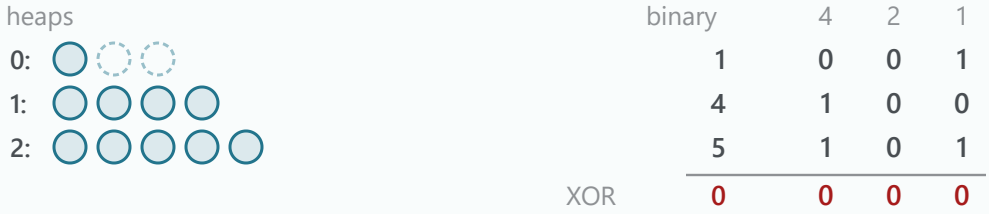
```
1  heaps = [3, 4, 5]
2  x = 0
3  for h in heaps:
4      x ^= h              # nim-sum (XOR)
▶  for i, h in enumerate(heaps):
▶      if h ^ x < h:       # can it shrink?
7          heaps[i] = h ^ x
8          print("heap", i, ":", h, "->", h ^ x)
9          break
10 print(heaps)
```

VARIABLES

heaps = [3, 4, 5] | x = 2 | i = 0 | h = 3 | h^x = 1

OUTPUT

Code Trace: The Winning Move in Nim (XOR)



Heap 0: 3 → 1. New position [1, 4, 5]: the nim-sum is now 0.


```
1  heaps = [3, 4, 5]
2  x = 0
3  for h in heaps:
4      x ^= h              # nim-sum (XOR)
5  for i, h in enumerate(heaps):
6      if h ^ x < h:       # can it shrink?
           heaps[i] = h ^ x
           print("heap", i, ":", h, "->", h ^ x)
           break
10 print(heaps)
```

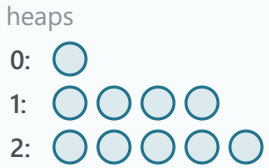
VARIABLES

`heaps = [1, 4, 5]` `x = 2` `i = 0` `h = 3`

OUTPUT

```
heap 0 : 3 -> 1
```

Code Trace: The Winning Move in Nim (XOR)



	binary	4	2	1
	1	0	0	1
	4	1	0	0
	5	1	0	1
XOR	0	0	0	0

if each column has an even count of 1s: XOR=0

The opponent faces XOR = 0: every move breaks it, and we restore 0 each time → we take the last chip.

Step 7 / 7

```
1 heaps = [3, 4, 5]
2 x = 0
3 for h in heaps:
4     x ^= h              # nim-sum (XOR)
5 for i, h in enumerate(heaps):
6     if h ^ x < h:       # can it shrink?
7         heaps[i] = h ^ x
8         print("heap", i, ":", h, "->", h ^ x)
9         break
▶ print(heaps)
```

VARIABLES

`heaps = [1, 4, 5]` `x = 2`

OUTPUT

```
heap 0 : 3 -> 1
[1, 4, 5]
```

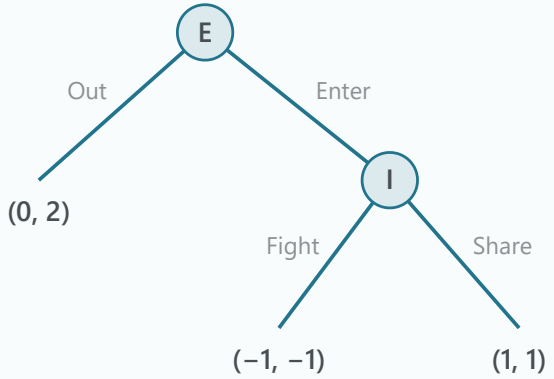
Subgames and Subgame-Perfect Equilibrium

- **Subgame**: a part of the game that starts at a single decision node and contains everything below it (without splitting information sets).
- **Subgame-perfect equilibrium** (SPE): a strategy profile that is a Nash equilibrium **in every subgame** (Selten, 1965).
- An ordinary Nash equilibrium may rely on **non-credible threats** at nodes that are never reached; SPE rules them out.
- In finite games of perfect information, SPE is found by **backward induction** (Kuhn/Zermelo).

Backward induction

1. Find the last decision nodes, those followed only by leaves.
2. There, pick the best branch for the player who decides.
3. Replace the node with the payoff of the chosen leaf.
4. Repeat until you reach the root.

Example: A Non-Credible Threat



E (the entrant) moves first; I (the incumbent monopolist) decides only if E enters. The normal form has two pure Nash equilibria (table on the right).

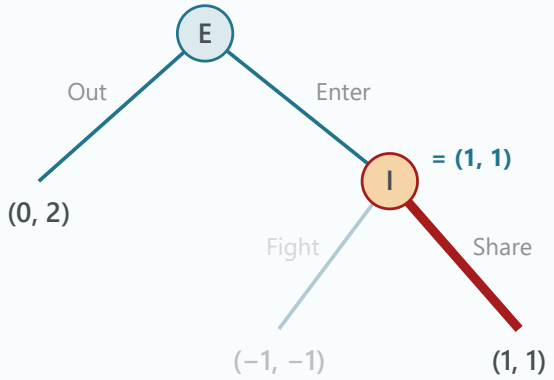
Step 1 / 3

E \ I	Fight	Share
Enter	(−1, −1)	(1, 1)
Out	(0, 2)	(0, 2)

Row: E, column: I. Orange = best response; framed cell = Nash equilibrium.

- (Enter, Share): the SPE, found by backward induction.
- (Out, Fight): a Nash equilibrium, but I's threat "if you enter, I fight" is **not credible**: once E enters, fighting gives I −1 and sharing gives 1.

Example: A Non-Credible Threat



Last node, I: Fight → −1, Share → 1 ⇒ **Share**. I's node is worth (1, 1).

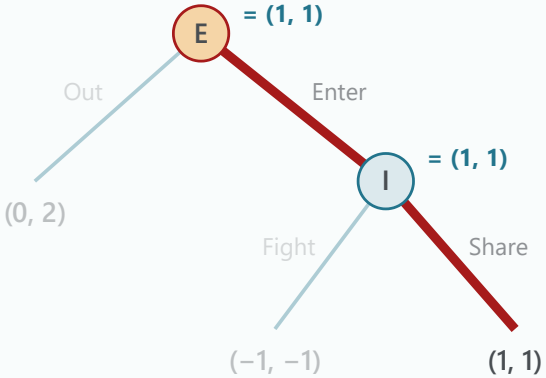
Step 2 / 3

E \ I	Fight	Share
Enter	(−1, −1)	(1, 1)
Out	(0, 2)	(0, 2)

Row: E, column: I. Orange = best response; framed cell = Nash equilibrium.

- (Enter, Share): the SPE, found by backward induction.
- (Out, Fight): a Nash equilibrium, but I's threat "if you enter, I fight" is **not credible**: once E enters, fighting gives I −1 and sharing gives 1.

Example: A Non-Credible Threat



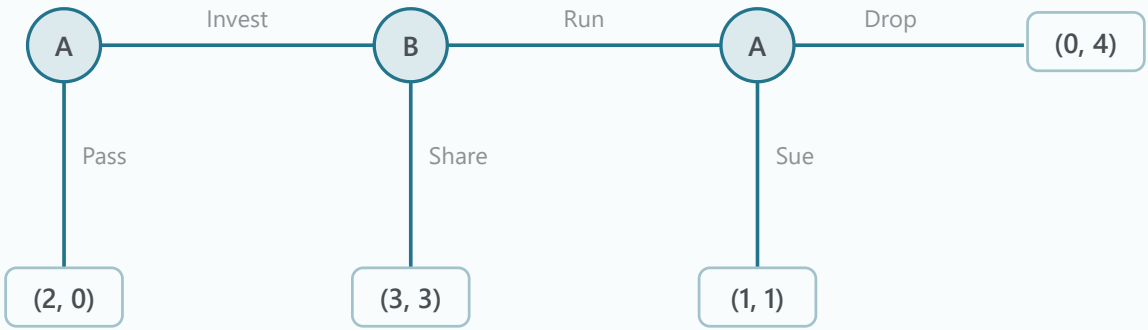
Root, E: Out → 0, Enter → 1 (knowing I will share) ⇒ **Enter**. SPE: (Enter, Share), outcome (1, 1).

E \ I	Fight	Share
Enter	(−1, −1)	(1, 1)
Out	(0, 2)	(0, 2)

Row: E, column: I. Orange = best response; framed cell = Nash equilibrium.

- (Enter, Share): the SPE, found by backward induction.
- (Out, Fight): a Nash equilibrium, but I's threat "if you enter, I fight" is **not credible**: once E enters, fighting gives I −1 and sharing gives 1.

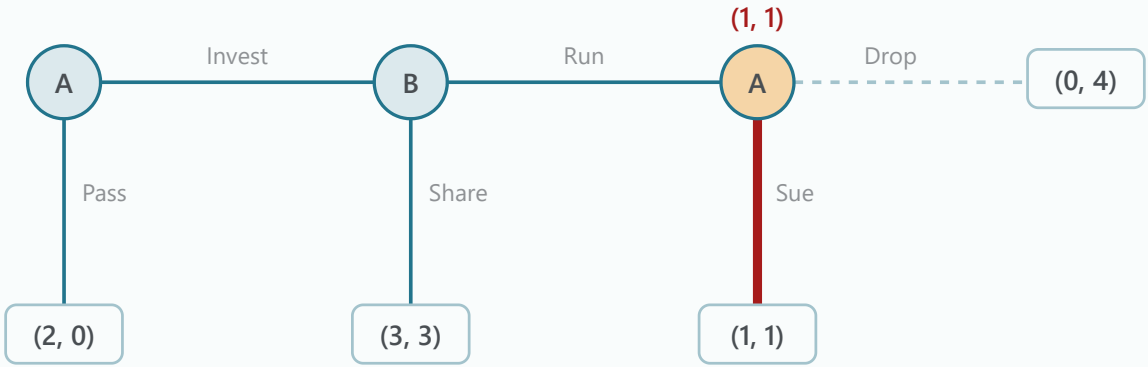
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Ayşe invests or not. If she invests, Bora shares the profit or runs off with the money. If he runs, Ayşe sues or drops it. The solution starts at the **end**.

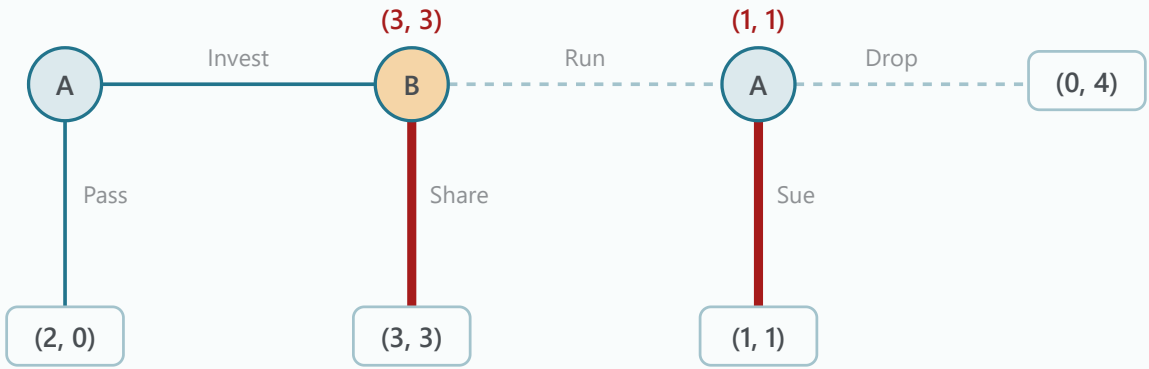
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Last decision node (Ayşe, after Bora runs): Sue 1 > Drop 0 → **Sue**. The node is worth (1, 1).

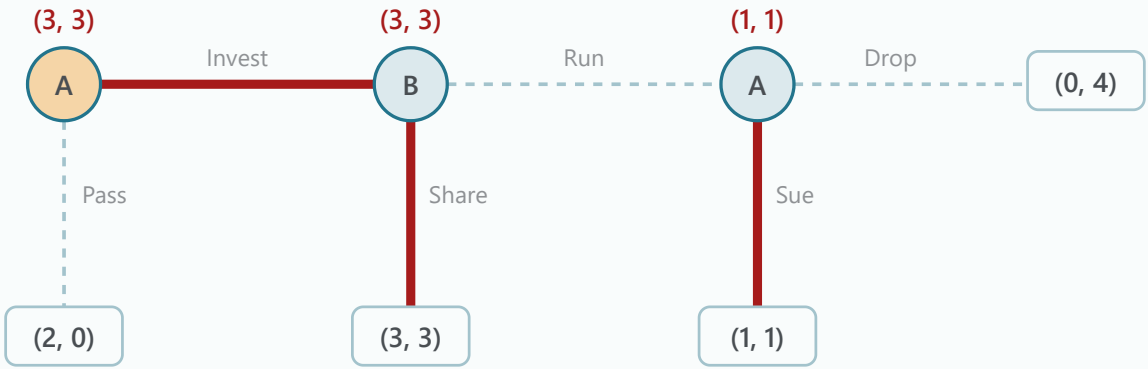
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Bora knows he will be sued if he runs: Share 3 > Run 1 → **Share**. Bora's node is worth (3, 3).

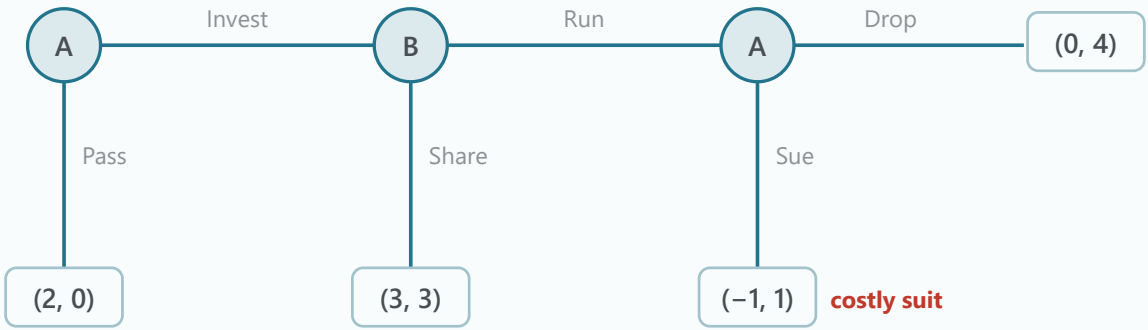
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Root: Invest 3 > Pass 2 → **Invest**. Subgame-perfect equilibrium: Ayşe (Invest, Sue), Bora Share; outcome **(3, 3)**.

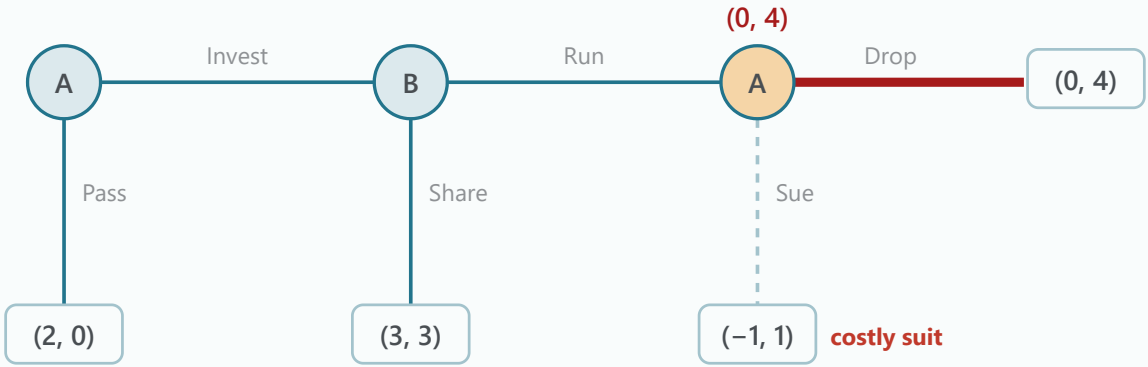
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Now suing is costly: Ayşe gets −1 if she sues. Same tree, one leaf changed. Start again from the end.

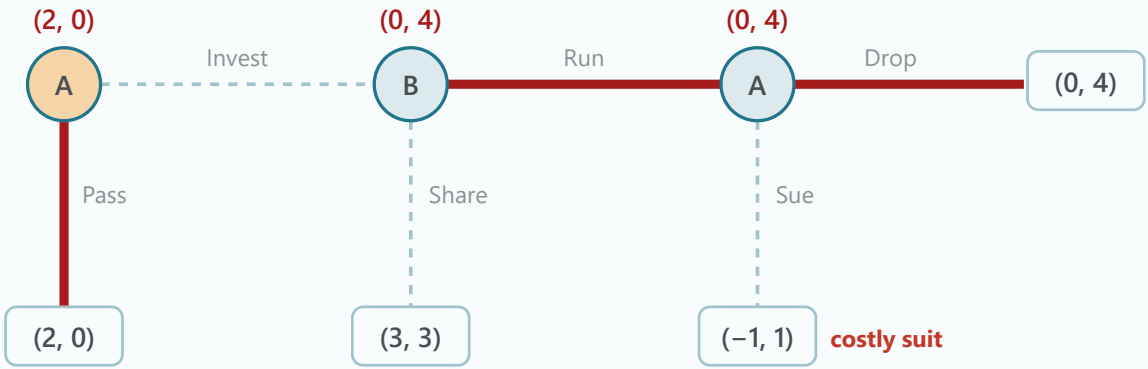
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Last node: Drop 0 > Sue −1 → **Drop**. The node is worth (0, 4).

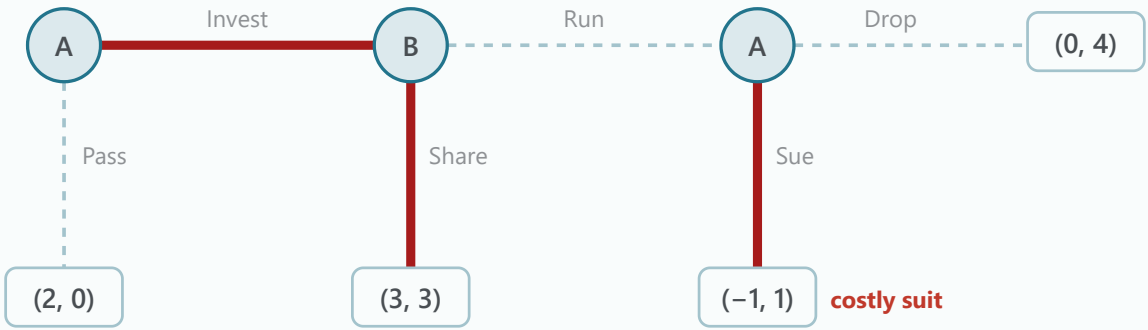
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

Bora: Run 4 > Share 3 → **Run**. Root: Pass 2 > 0 → **Pass**. The outcome (2, 0) is worse for both than (3, 3).

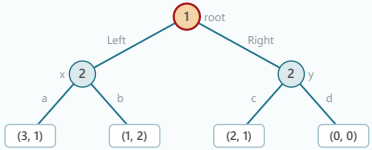
Step by Step: Backward Induction (Investment Game)



A: Ayşe (investor), B: Bora (entrepreneur); leaves show (Ayşe, Bora)

With the plan "if you run, I sue", (Invest, Share) is still a **Nash equilibrium**: nobody deviates alone. But the threat is not credible ($-1 < 0$): it is not a subgame-perfect equilibrium.

Code Trace: `solve(d)`



Player 1 (index 0) chooses at the root, player 2 (index 1) at x and y. solve(root) is called.

Step 1 / 10

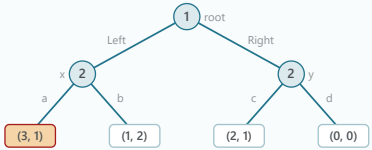
```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
▶      name, player, branches = d
▶      best = None
6      for move, child in branches:
7          v = solve(child)
8          if best is None or v[player] > best[player]:
9              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
▶  print(solve(T))
```

VARIABLES

`name = root` `player = 0` `best = None`

OUTPUT

Code Trace: `solve(d)`



Left branch: solve(x) → first branch a; leaf (3, 1) is returned as is.

Step 2 / 10

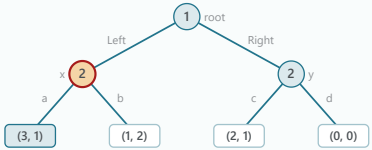
```
1  def solve(d):
▶      if isinstance(d, tuple):          # leaf: payoffs
▶          return d
4      name, player, branches = d
5      best = None
▶      for move, child in branches:
▶          v = solve(child)
8          if best is None or v[player] > best[player]:
9              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

VARIABLES

`name = x` `player = 1`
`move = a` `v = (3, 1)`

OUTPUT

Code Trace: `solve(d)`



No best at x yet: best = (3, 1), choice = a.

Step 3 / 10

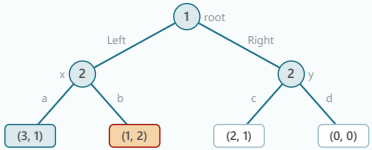
```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
6      for move, child in branches:
7          v = solve(child)
▶          if best is None or v[player] > best[player]:
▶              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

VARIABLES

`name = x` `player = 1` `v = (3, 1)` `best = (3, 1)` `choice = a`

OUTPUT

Code Trace: `solve(d)`



Second branch b: (1, 2). Player 2 looks at its own payoff: v[1] = 2 > best[1] = 1.

Step 4 / 10

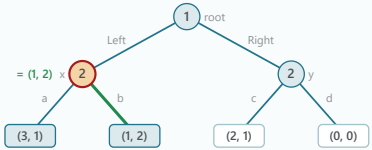
```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
▶      for move, child in branches:
▶          v = solve(child)
▶          if best is None or v[player] > best[player]:
9              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

VARIABLES

`name = x` `player = 1` `move = b` `v = (1, 2)` `best = (3, 1)`

OUTPUT

Code Trace: `solve(d)`



choice = b; "x b" is printed and x is worth (1, 2). Player 1's favorite, (3, 1), is now out of reach.

Step 5 / 10

```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
6      for move, child in branches:
7          v = solve(child)
8          if best is None or v[player] > best[player]:
               best, choice = v, move
      print(name, choice)
      return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                  ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

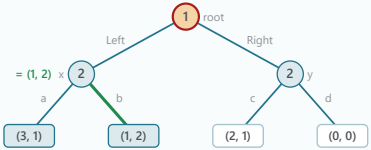
VARIABLES

`name = x` `best = (1, 2)` `choice = b`

OUTPUT

```
x b
```

Code Trace: `solve(d)`



At the root, Left is worth (1, 2): best = (1, 2), choice = Left.

Step 6 / 10

```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
6      for move, child in branches:
▶          v = solve(child)
▶          if best is None or v[player] > best[player]:
▶              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

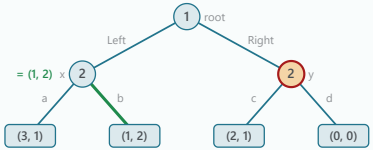
VARIABLES

`name = root` `player = 0` `move = Left` `v = (1, 2)` `best = (1, 2)` `choice = Left`

OUTPUT

```
x b
```

Code Trace: `solve(d)`



Right branch: solve(y). c → (2, 1), d → (0, 0); for player 2, 1 > 0, so choice = c.

Step 7 / 10

```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
▶      for move, child in branches:
▶          v = solve(child)
▶          if best is None or v[player] > best[player]:
▶              best, choice = v, move
10     print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                 ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

VARIABLES

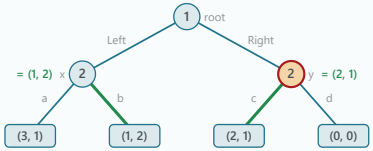
`name = y` `player = 1`

`best = (2, 1)` `choice = c`

OUTPUT

```
x b
```

Code Trace: `solve(d)`



"y c" is printed; y returns (2, 1).

Step 8 / 10

```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
6      for move, child in branches:
7          v = solve(child)
8          if best is None or v[player] > best[player]:
9              best, choice = v, move
▶      print(name, choice)
▶      return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                  ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

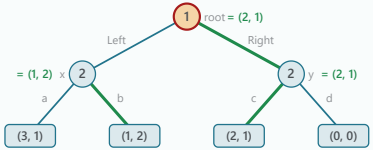
VARIABLES

`name = y` `best = (2, 1)` `choice = c`

OUTPUT

```
x b
y c
```

Code Trace: `solve(d)`



At the root player 1 compares: Right 2 > Left 1 → choice = Right. "root Right" is printed.

Step 9 / 10

```
1  def solve(d):
2      if isinstance(d, tuple):          # leaf: payoffs
3          return d
4      name, player, branches = d
5      best = None
6      for move, child in branches:
7          v = solve(child)
▶          if best is None or v[player] > best[player]:
▶              best, choice = v, move
▶      print(name, choice)
11     return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                  ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
14 print(solve(T))
```

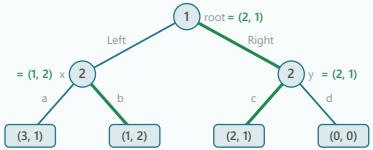
VARIABLES

`name = root` `player = 0` `v = (2, 1)` `best = (2, 1)` `choice = Right`

OUTPUT

```
x b
y c
root Right
```

Code Trace: `solve(d)`



Result (2, 1). SPE: (Right; b at x, c at y). The unreached choice b at x is part of the strategy too.

Step 10 / 10

```
 1 def solve(d):
 2     if isinstance(d, tuple):          # leaf: payoffs
 3         return d
 4     name, player, branches = d
 5     best = None
 6     for move, child in branches:
 7         v = solve(child)
 8         if best is None or v[player] > best[player]:
 9             best, choice = v, move
10     print(name, choice)
▶       return best
12 T = ['root', 0, [('Left', ['x', 1, [('a', (3, 1)), ('b', (1, 2))]]),
13                  ('Right', ['y', 1, [('c', (2, 1)), ('d', (0, 0))]])]]
▶ print(solve(T))
```

VARIABLES

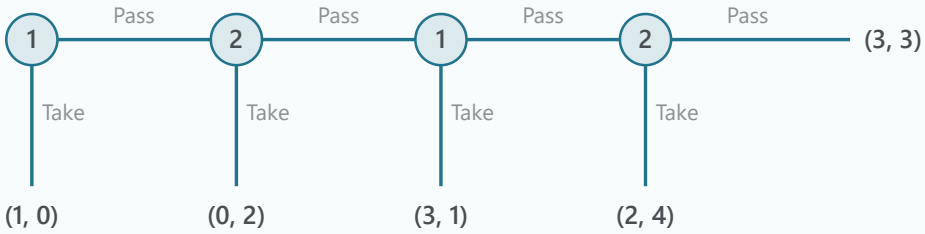
`result = (2, 1)`

OUTPUT

```
x b
y c
root Right
(2, 1)
```

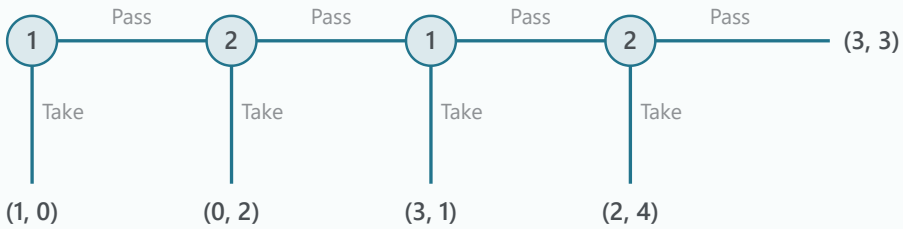
The Centipede Game

- Two players decide in turn (Rosenthal, 1981). At each node a player either says **Take**, and the current split is realized, or **Pass**, and the pot grows.
- The total payoff grows (1 → 2 → 4 → 6; also 6 at the end), but a player who passes gets less if the opponent takes at the next step.



The number in a node is the player who decides. Leaves show (Player 1, Player 2) payoffs; at the far right, if everyone passes, (3, 3).

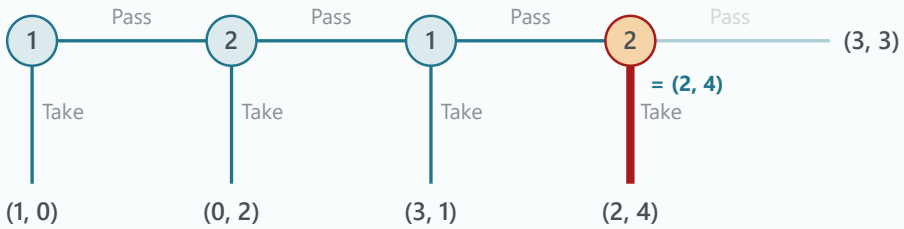
Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	?	?	?
3	Player 1	(3, 1)	?	?	?
2	Player 2	(0, 2)	?	?	?
1	Player 1	(1, 0)	?	?	?

Player 1 starts; at each node the player says **Take** (down) or **Pass** (right). We start at the last decision node (node 4, Player 2) and work backward.

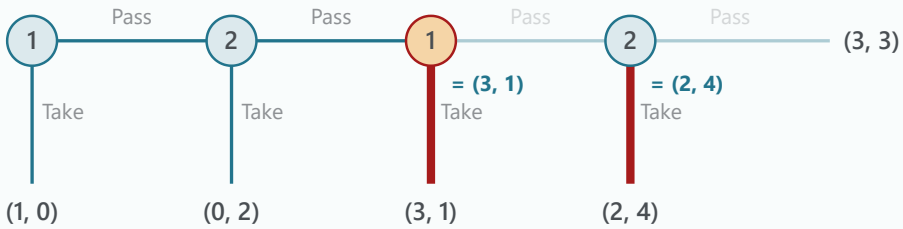
Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	(3, 3)	**Take**	(2, 4)
3	Player 1	(3, 1)	?	?	?
2	Player 2	(0, 2)	?	?	?
1	Player 1	(1, 0)	?	?	?

Node 4, Player 2: Take → 4, Pass → 3 (passing is worth (3, 3)). 4 > 3 ⇒ **Take**; the node is worth (2, 4).

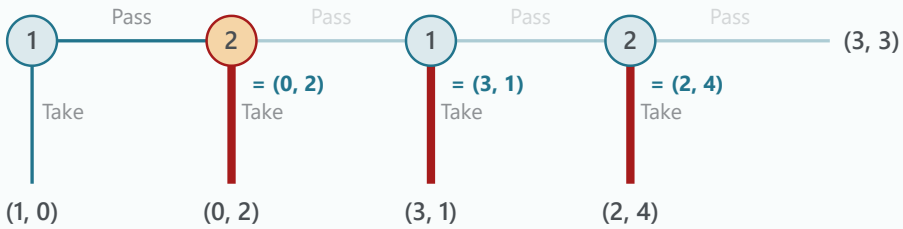
Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	(3, 3)	**Take**	(2, 4)
3	Player 1	(3, 1)	(2, 4)	**Take**	(3, 1)
2	Player 2	(0, 2)	?	?	?
1	Player 1	(1, 0)	?	?	?

Node 3, Player 1: Take → 3, Pass → 2 (passing is worth (2, 4)). 3 > 2 ⇒ **Take**; the node is worth (3, 1).

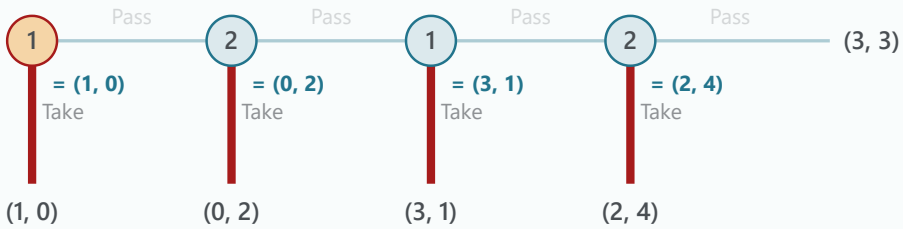
Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	(3, 3)	**Take**	(2, 4)
3	Player 1	(3, 1)	(2, 4)	**Take**	(3, 1)
2	Player 2	(0, 2)	(3, 1)	**Take**	(0, 2)
1	Player 1	(1, 0)	?	?	?

Node 2, Player 2: Take → 2, Pass → 1 (passing is worth (3, 1)). 2 > 1 ⇒ **Take**; the node is worth (0, 2).

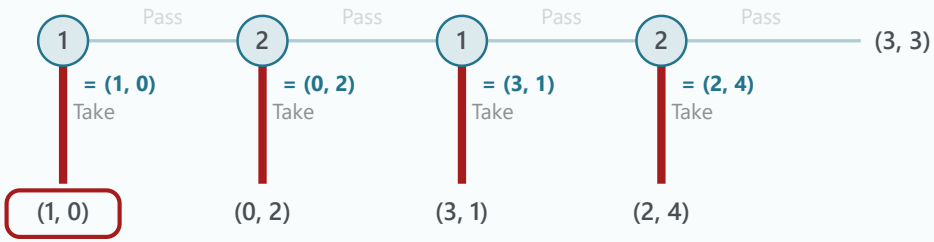
Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	(3, 3)	**Take**	(2, 4)
3	Player 1	(3, 1)	(2, 4)	**Take**	(3, 1)
2	Player 2	(0, 2)	(3, 1)	**Take**	(0, 2)
1	Player 1	(1, 0)	(0, 2)	**Take**	(1, 0)

Node 1, Player 1: Take → 1, Pass → 0 (passing is worth (0, 2)). 1 > 0 ⇒ **Take**; the node is worth (1, 0).

Centipede: Backward Induction Step by Step



Node	Decides	Take	Value of passing	Decision	Node value
4	Player 2	(2, 4)	(3, 3)	**Take**	(2, 4)
3	Player 1	(3, 1)	(2, 4)	**Take**	(3, 1)
2	Player 2	(0, 2)	(3, 1)	**Take**	(0, 2)
1	Player 1	(1, 0)	(0, 2)	**Take**	(1, 0)

Subgame-perfect equilibrium: every player **takes at every node**. The game ends at the first move with (1, 0), although (3, 3) was better for both.

The Centipede Game: Theory and Experiment

Theory (SPE)

Every player takes at every node; the game ends at the first move with (1, 0). Every Nash equilibrium of the game has the same outcome.

Inefficiency

(3, 3) is better than (1, 0) for both players: the SPE outcome is **not Pareto optimal**. The loss grows with the number of nodes.

Experiment

In the lab most players do not take at the first node; they pass for a few rounds (McKelvey and Palfrey, 1992).

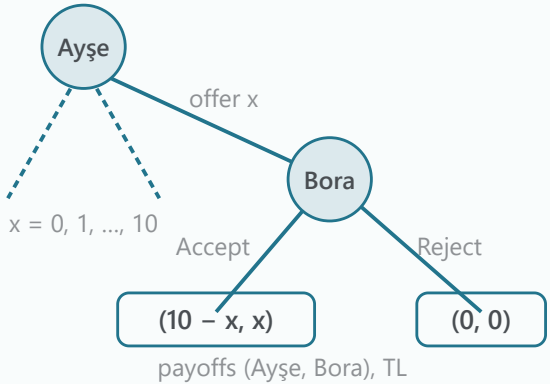
Why?

Backward induction requires rationality to be **common knowledge** and a long chain of reasoning; even a small chance that the opponent is "nice" makes passing profitable.

Centipede shares its logic with the finitely repeated Prisoner's Dilemma: looking back from the last round, cooperation unravels. With an infinite horizon, cooperation can survive in equilibrium.

The Ultimatum Game: Backward Induction in Bargaining

- **Rule:** Ayşe offers Bora x of 10 TL (whole TL). If Bora **accepts**: (10 − x, x); if he **rejects**: both 0 (Güth, Schmittberger and Schwarze, 1982).
- **Backward induction:** for x > 0 Bora prefers accepting (x) to rejecting (0), so Ayşe offers the smallest unit: **SPE outcome (9, 1)**. (If Bora also accepts 0, (10, 0) is an SPE too.)
- **Non-credible threat:** "I reject anything under 4 TL" supports a Nash equilibrium, but facing 1 TL, rejecting gives 0 < 1.



Offer (share of the total)	Theory (money only)	In experiments
Most common offer	smallest unit	40–50%
Below 20%	accepted	about half rejected

Common mistake: reading this as "people cannot do backward induction". Rejections show fairness enters utility; proposers anticipate it (Camerer, 2003).

The Infinitely Repeated Prisoner's Dilemma: Trigger Strategies

The **grim trigger strategy** (GT): start Silent; if the opponent ever defects, Defect forever. When the same one-shot game is repeated, what decides the outcome is **whether players know when the game ends**.

Finite, known horizon

In the last round there is no future and no punishment: Defect is dominant there. Backward induction carries this to the previous round and on to the first; Defect in every round (as in the centipede).

Infinite horizon (each round continues with probability δ)

Today's gain from defecting is weighed against the lasting loss from future punishment. If δ is large enough (patient players), deviating does not pay and cooperation is sustained in equilibrium.

The punishment is credible because (Defect, Defect) is an equilibrium of the one-shot game: two grim trigger strategies together form a **subgame-perfect equilibrium**.

Common Mistakes: Sequential Games

Starting at the root

The first move depends on the responses to later moves; the solution always starts at the last decision nodes.

The wrong player's payoff

At each node compare only the numbers of the player *deciding there* (the second number at Bora's node).

Confusing a strategy with a path

A strategy specifies a choice at every node, including unreached ones: Ayşe's strategy is (Invest, Sue).

Nash = subgame perfect

Every subgame-perfect equilibrium is a Nash equilibrium, not conversely: some Nash equilibria rest on non-credible threats.

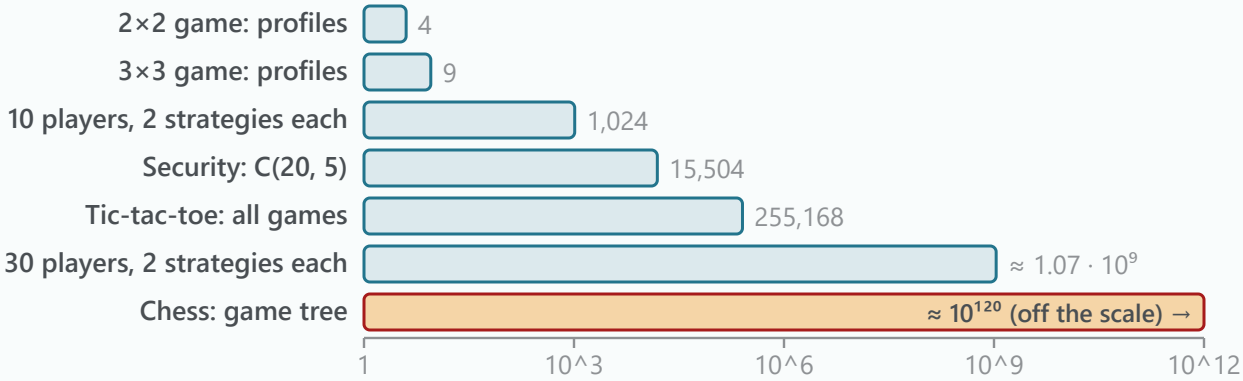
Solving simultaneous games sequentially

If moves are not observed (Prisoner's Dilemma), the tree has an information set; backward induction does not apply directly, so solve the matrix.

Words = commitment

Saying "I will sue" is not enough; carrying out the threat must pay at that moment, or be locked in beforehand (contract, deposit).

Why Is Algorithmic Game Theory "Algorithmic"?



Size

Even writing the game down grows exponentially: n players × 2 strategies → 2^n profiles. Brute-force search does not scale.

Complexity

A Nash equilibrium always exists, but finding one is computationally hard even in two-player games.

Structure and design

Special structure brings efficiency: linear programming for zero-sum games, XOR for Nim. Rules are algorithms too.

THE ALGORITHMIC VIEW

The Four Questions of Algorithmic Game Theory

1. Computation: can equilibria be found?

Zero-sum games are solved quickly by linear programming. No fast method is known for Nash equilibria in general games; the problem is **PPAD-complete**. Solving Hex on large boards is very hard (**PSPACE-complete**).

2. Efficiency: what does selfishness cost?

Price of Anarchy = cost of the worst equilibrium / cost of the best outcome. In the Prisoner's Dilemma, counting prison years as cost, the equilibrium totals 4 years and (Silent, Silent) 2: ratio 4/2 = 2.

3. Design: which rules?

Mechanism design seeks rules under which players report private information truthfully (incentive compatible): auctions, matching, voting.

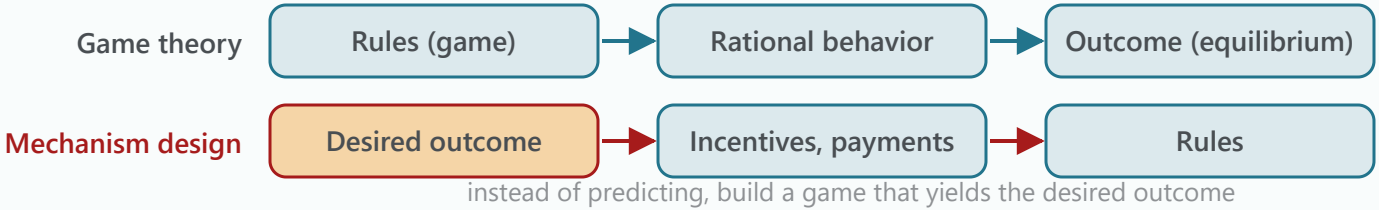
4. Dynamics: how do players reach equilibrium?

In repeated and online games information arrives over time; players learn from the past and adapt their strategies.

The common language of the four questions: classifying games, equilibria of classic games and backward induction in sequential games.

Mechanism Design: Game Theory in Reverse

- Game theory predicts the outcome of a given game; **mechanism design** designs the rules and incentives that produce a desired outcome.
- The difficulty: agents' private information (valuations, locations, preferences) is hidden from the designer, and agents may lie.



With money

The designer can charge payments; payments reward telling the truth.

`auctions, ad placement`

Without money

Incentives come from the rules alone; usually the objective is partly compromised.

`elections, room swaps, facility location`

Real-World Applications

Search ads
Ad slots are allocated on every search by a **generalized second-price** (GSP) auction. With several slots, truthful bidding is not dominant in GSP.

Spectrum auctions
Governments sell frequency licenses through designed auctions (in the US since 1994).

School choice
Under student-proposing Gale–Shapley, reporting true preferences is best for each student; the result is a stable matching.

Room and shift swaps
Everyone brings what they hold; trading cycles leave nobody worse off than before, and no money is used.

Traffic and routing
Each driver picks their own fastest route; the cost of selfish choices to total delay is measured by the Price of Anarchy.

Platforms and incentives
Surge pricing draws drivers to peak hours; Bitcoin miners validate transactions for rewards.

The common idea: design the rules so that the system works well even when everyone acts **in their own interest**.

Summary: Concept Map

Concept	Short definition	Example
Normal / extensive form	Matrix (simultaneous) or tree (order, information)	RPS, price competition; chess
Classification	Timing, agreements, sum, symmetry, information, horizon	Chess: sequential, zero-sum, perfect info
Dominant strategy, Nash equilibrium	Best whatever others do; no profitable unilateral deviation	Defect; (Defect, Defect)
Pareto optimality	No one better off without hurting someone	Equilibrium (−2, −2) is not
Classic 2×2 games	Conflict or coordination; best responses	Matching Pennies: none; Stag Hunt: two
Iterated elimination, level-k	Delete dominated strategies; best guess given beliefs	2/3 game: 90 → 60 → 40 → … → 0
Minimax, Nim	Leaves to root; P/N positions	Tree value 3; tic-tac-toe draws; Nim XOR
Subgame-perfect equilibrium	Nash in every subgame; found by backward induction	Entry game; centipede (1, 0)
Grim trigger	Permanent punishment deters defection in infinite play	Cooperation if patient (large δ)

SUMMARY

Practice Problems (1/2)

1. What are the dominant strategies and the equilibrium of the price competition game? Is the equilibrium Pareto optimal?
2. If the game used 0–100 and the target were "half the average", how would iterated elimination proceed, and what is the equilibrium?
3. In the 2/3 game (0–90), if 20% of the class reasons at level 0 and 80% at level 1, which level wins?
4. (A, A) = (4, 1), (B, B) = (1, 4), other cells (0, 0). What are the pure equilibria? Which classic game does it resemble?
5. Which of the six classic 2×2 games are zero-sum, and which are symmetric?

▶ **Show answers**

SUMMARY

Practice Problems (2/2)

6. Nim position [2, 5, 6], your move: is it a winning position? Which move wins?
7. A MAX root has three MIN children: B = [7, 2, 9], C = [4, 6], D = [1, 8]. What is the game value, and what does MAX choose?
8. "Don't Say 16": players say 1–3 consecutive numbers in turn; whoever says 16 loses. What are the key numbers? Who wins?
9. If the far-right payoff in the centipede were (5, 5), what would the subgame-perfect equilibrium be?
10. Why is (Out, Fight) not subgame perfect in the entry game?

▶ **Show answers**

References (1/2)

- Nisan, Roughgarden, Tardos and Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007 (Chapter 1).
- Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, Cambridge University Press, 2016.
- Osborne, *An Introduction to Game Theory*, Oxford University Press, 2004 (Chapters 2–5) · Shoham and Leyton-Brown, *Multiagent Systems*, Cambridge University Press, 2009 (Chapters 3 and 5).
- Nash, “Equilibrium points in n-person games”, *PNAS* 36(1), 1950 · von Neumann, “Zur Theorie der Gesellschaftsspiele”, *Mathematische Annalen* 100, 1928.
- Bouton, “Nim, a game with a complete mathematical theory”, *Annals of Mathematics* 3, 1901–1902.
- Nagel, “Unraveling in guessing games: An experimental study”, *American Economic Review* 85(5), 1995.
- R. W. Rosenthal, "Games of perfect information, predatory pricing and the chain-store paradox", *J. Economic Theory* 25, 1981 · R. D. McKelvey, T. R. Palfrey, "An experimental study of the centipede game", *Econometrica* 60(4), 1992.
- W. Güth, R. Schmittberger, B. Schwarze, "An experimental analysis of ultimatum bargaining", *J. Economic Behavior & Organization* 3(4), 1982 · C. F. Camerer, *Behavioral Game Theory*, Princeton University Press, 2003.
- B. Skyrms, *The Stag Hunt and the Evolution of Social Structure*, Cambridge University Press, 2004.
- S. Reisch, "Hex ist PSPACE-vollständig", *Acta Informatica* 15, 1981 · J. Schaeffer et al., "Checkers is solved", *Science* 317(5844), 2007.

References (2/2): Further Books, Course Pages, Tools

Further books

- Karlin and Peres, *Game Theory, Alive*, American Mathematical Society, 2017. Combinatorial games, Nim, 2×2 games.
- Easley and Kleinberg, *Networks, Crowds, and Markets*, Cambridge University Press, 2010. Games, networks and markets.
- Hearn and Demaine, *Games, Puzzles, and Computation*, A K Peters, 2009. Computational complexity of games.

Course pages

- Roughgarden, CS364A (Stanford): the course behind Twenty Lectures.
- Shah, CSC304 (Toronto) · Procaccia, 15-896 (CMU)
- Dinitz, AGT (Johns Hopkins) · Hoefer, AGT (Frankfurt)

Tools and video

- Gambit: open-source tools for computing equilibria of game trees and matrix games.
- Nashpy: a Python library for two-player games.
- Game Theory 101 (YouTube playlist).

The End

Chapter 2: Games and Representations

Previous: Chapter 1: Decision Theory

Next: Chapter 2: Minimax and Alpha-Beta Pruning · Chapter 3: Nash Equilibrium