Chapter 12: Behavioral and Collective Decisions

Algorithmic Game Theory

Prospect theory · Loss aversion · Quantal response · Social preferences · The commons and Ostrom · Public goods and punishment · Threshold projects · Information cascades · Condorcet jury theorem

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_12_Behavioral_and_Collective_Decisions.html

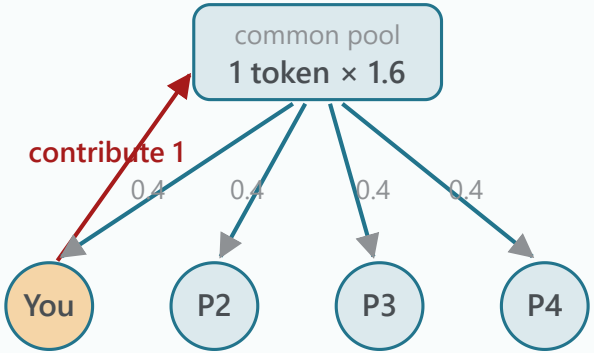
Warm-up: The Public Goods Game

The rules

Each round, four players get 10 tokens each and put any number into a common pool; the pool is multiplied by 1.6 and split equally among all four. Tokens not contributed are kept.

Think about it

- In a group of four, how many tokens do you think are contributed on average in the first round? After ten rounds?
- If anyone contributing fewer than 3 tokens is fined 2 tokens (a punishment variant), how do contributions change?



Net to you −1 + 0.4 = −0.6; +1.2 in total to the other three

Learning Outcomes

Behavioral decisions

Value a gamble with prospect theory; explain loss aversion, the endowment effect and framing; show how quantal response changes the Nash prediction.

Social preferences

Explain deviations in dictator, ultimatum and trust games by inequity aversion and reciprocity; compute the Fehr–Schmidt acceptance threshold.

Commons and public goods

Compare the commons equilibrium with the optimum; interpret Ostrom's principles, punishment and second-order free riding, and equilibria of threshold projects.

Information aggregation

Analyze the wisdom of crowds with the diversity theorem, information cascades with Bayes' rule, and the Condorcet jury theorem with binomial probabilities.

Are People Always Rational? Bounded Rationality

- **Bounded rationality** (Simon): information, time and computing power are limited; people look for "good enough", not the best.
- Deviations are **systematic**, not random: the same mistakes recur, so they can be modeled.
- Causes: missing information, time pressure, emotions, shortcuts (heuristics).

Overconfidence

Trusting one's own estimates too much: investors trade too often.

Framing effect

The decision changes when the same option is presented differently.

Loss aversion

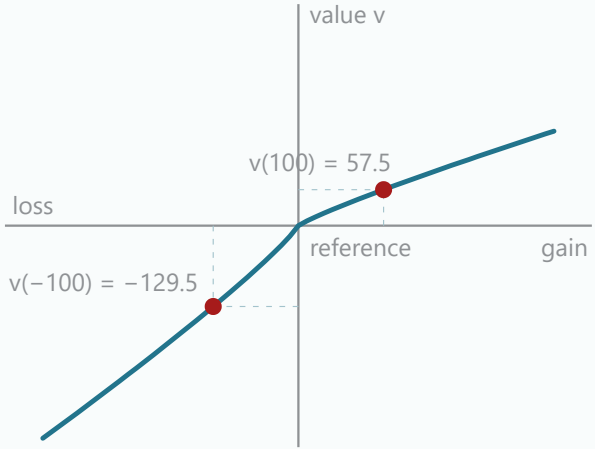
A loss hurts more than an equal gain pleases: panic selling.

Base-rate neglect

Ignoring how common a disease is (the prior) when reading a positive test result.

Prospect Theory

- Kahneman and Tversky (1979): outcomes are judged as gains or losses relative to a **reference point**, not as final wealth.
- The value function is concave for gains, convex for losses, and **steeper** for losses: $v(x) = x^{0.88}$ $(x \geq 0)$; $v(x) = -2.25 \cdot (-x)^{0.88}$ $(x < 0)$
- $x^{0.88}$ grows more slowly than x: $v(200) \approx 105.9$, 1.84 times $v(100)$, not 2. Concave: the curve flattens to the right.
- $\lambda = 2.25$ is the **loss aversion** coefficient (Tversky and Kahneman's 1992 estimate).
- Consequence (ignoring weights): risk aversion for gains, risk seeking for losses.



Reference point example: someone expecting a 10,000 TL bonus who gets 9,000 TL feels a 1,000 TL **loss**; to someone expecting nothing, the same amount is a big gain.

BEHAVIORAL DECISIONS

Loss Aversion: A Coin-Flip Bet

The bet

Heads +100 TL, tails −100 TL. Expected value 0.

Prospect value

$\frac{1}{2}\cdot v(100) + \frac{1}{2}\cdot v(-100) = \frac{1}{2}\cdot 57.5 - \frac{1}{2}\cdot 129.5 = \mathbf{-36.0} < 0 \Rightarrow$ rejected.

Acceptance threshold

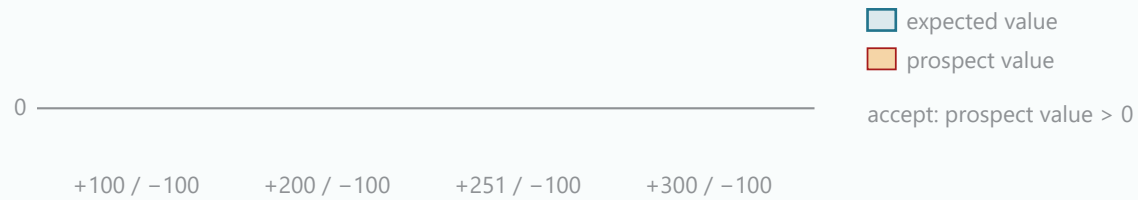
$\frac{1}{2}\cdot G^{0.88} = \frac{1}{2}\cdot 2.25\cdot 100^{0.88} \Rightarrow (G/100)^{0.88} = 2.25 \Rightarrow G = 100\cdot 2.25^{1/0.88} \approx$ **251 TL**.

To risk losing 100 TL, a person needs a chance to win about **251 TL**. In experiments, too, people typically ask for more than twice the loss.

Probability weighting w(p)	p = 0.01	p = 0.50	p = 0.99
Perceived weight	0.055	0.421	0.912

The probability weighting function w(p) shows how a probability feels (Tversky and Kahneman's 1992 estimate): 1% feels like about 5.5%, 50% like about 42%, 99% like about 91%. Small probabilities are overweighted and large ones underweighted: the same person may buy both lottery tickets and insurance.

Code Trace: When Is a Bet Accepted?



Four coin-flip bets: win +G with probability ½, lose 100 TL with probability ½. Expected value (G − 100)/2; prospect value uses v.

Step 1 / 6

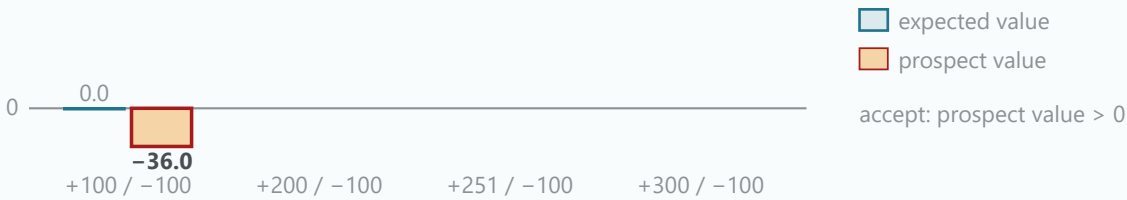
```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):       # ½: +G, ½: −K
5     return 0.5 * v(G) + 0.5 * v(-K)
▶ for G in [100, 200, 251, 300]:
7     print(G, round(prospect(G, 100), 1))
```

VARIABLES

`G = –` `K = 100`

OUTPUT

Code Trace: When Is a Bet Accepted?



G = 100: v(100) = 57.5 and v(−100) = −129.5 ⇒ ½·57.5 − ½·129.5 = **−36.0**. A bet with expected value 0 is rejected.

Step 2 / 6

```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):      # ½: +G, ½: -K
▶     return 0.5 * v(G) + 0.5 * v(-K)
▶ for G in [100, 200, 251, 300]:
▶     print(G, round(prospect(G, 100), 1))
```

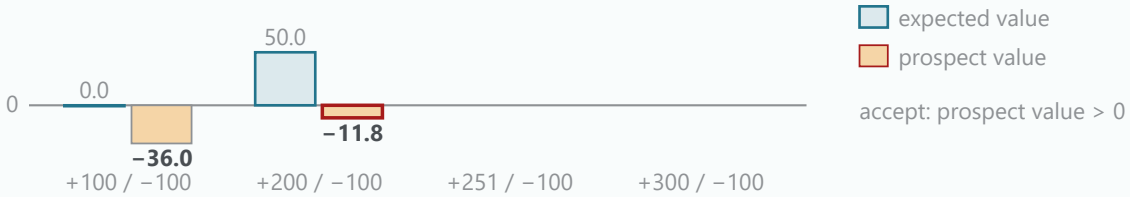
VARIABLES

`G = 100` `v(G) = 57.5` `v(−K) = −129.5` `value = −36.0`

OUTPUT

```
100 -36.0
```

Code Trace: When Is a Bet Accepted?



G = 200: v(200) = 105.9 ⇒ ½·105.9 − ½·129.5 = **−11.8**. Still rejected, although the expected value is +50.

Step 3 / 6

```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):      # ½: +G, ½: −K
▶     return 0.5 * v(G) + 0.5 * v(-K)
▶ for G in [100, 200, 251, 300]:
▶     print(G, round(prospect(G, 100), 1))
```

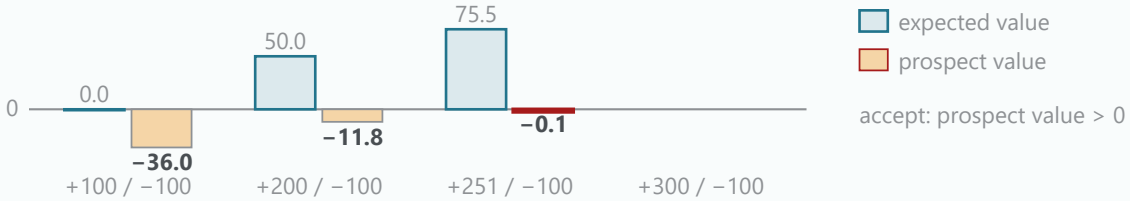
VARIABLES

`G = 200` `v(G) = 105.9` `v(−K) = −129.5` `value = −11.8`

OUTPUT

```
100 -36.0
200 -11.8
```

Code Trace: When Is a Bet Accepted?



G = 251: v(251) = 129.3 ⇒ ½·129.3 − ½·129.5 ≈ **−0.1**: right at the acceptance threshold (≈ 251 TL).

Step 4 / 6

```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):      # ½: +G, ½: −K
▶     return 0.5 * v(G) + 0.5 * v(-K)
▶ for G in [100, 200, 251, 300]:
▶     print(G, round(prospect(G, 100), 1))
```

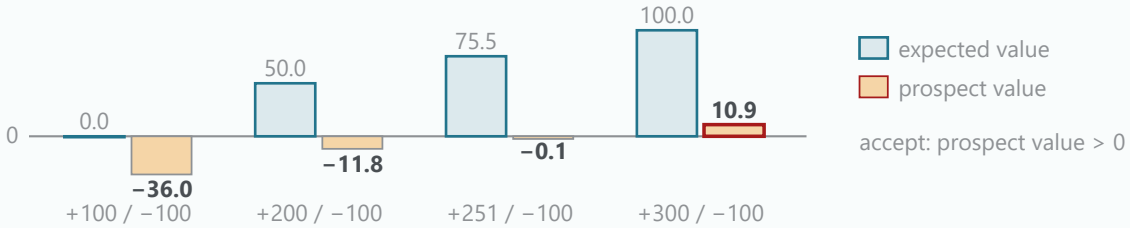
VARIABLES

`G = 251` `v(G) = 129.3` `v(−K) = −129.5` `value = −0.1`

OUTPUT

```
100 -36.0
200 -11.8
251 -0.1
```

Code Trace: When Is a Bet Accepted?



G = 300: v(300) = 151.3 ⇒ ½·151.3 − ½·129.5 = **+10.9** > 0: now the bet is accepted.

Step 5 / 6

```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):      # ½: +G, ½: −K
      return 0.5 * v(G) + 0.5 * v(-K)
for G in [100, 200, 251, 300]:
      print(G, round(prospect(G, 100), 1))
```

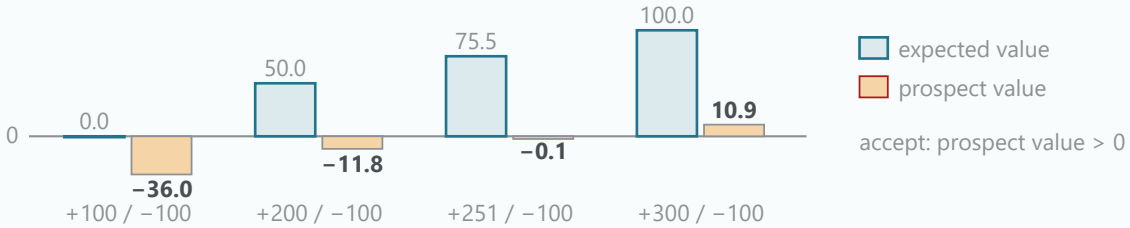
VARIABLES

`G = 300` `v(G) = 151.3` `v(−K) = −129.5` `value = 10.9`

OUTPUT

```
100 -36.0
200 -11.8
251 -0.1
300 10.9
```

Code Trace: When Is a Bet Accepted?



A risk-neutral person takes the +200 / −100 bet (expected value +50) at once; with λ = 2.25 the loss weighs so much that the bet is rejected until the gain exceeds 251 TL.

Step 6 / 6

```
1 def v(x):
2     if x >= 0: return x ** 0.88
3     return -2.25 * (-x) ** 0.88
4 def prospect(G, K):      # ½: +G, ½: −K
5     return 0.5 * v(G) + 0.5 * v(-K)
6 for G in [100, 200, 251, 300]:
▶     print(G, round(prospect(G, 100), 1))
```

VARIABLES

`G = 300`

OUTPUT

```
100 -36.0
200 -11.8
251 -0.1
300 10.9
```

A Trace of Loss Aversion: The Endowment Effect

- **Experiment** (Kahneman, Knetsch and Thaler, 1990): half the students were randomly given a mug. Owners' minimum selling price was about **twice** the maximum price buyers were willing to pay, and far fewer trades took place than expected.
- **Explanation:** giving up the mug is a **loss** from the reference point, getting it is a **gain**; the loss weighs λ times as much.
- **Status quo bias:** leaving the default also feels like a loss, which is why automatic enrollment raises pension plan participation.

Example

The mug's use value is 20 TL, the value function is linear, $\lambda = 2.25$.

- Owner: giving up the mug feels like $-2.25 \cdot 20 = -45 \Rightarrow$ asks at least **45 TL**.
- Buyer: getting the mug is $+20 \Rightarrow$ pays at most **20 TL**.
- Ratio $45/20 = 2.25 = \lambda$: no trade at prices in between.

Common mistake

"The owner likes the mug more." No: the object is the same; only the reference point changes.

Framing: Same Option, Different Choice

Tversky and Kahneman (1981): two programs for an outbreak expected to affect 600 people.

Gain frame

- \+ A: 200 people are **saved**.
- \+ B: with probability 1/3, 600 are saved; with probability 2/3, no one is.
- \+ **72%** of participants chose A (the sure thing).

Loss frame

- – C: 400 people **die**.
- – D: with probability 1/3, no one dies; with probability 2/3, 600 die.
- – **78%** of participants chose D (the gamble).

A ≡ C and B ≡ D (200 expected survivors in each). Expected utility theory says the choice should not change; prospect theory explains it by risk aversion for gains and risk seeking for losses.

Common Mistakes: Prospect Theory

"Loss aversion is risk aversion."

No. Risk aversion comes from the curvature of utility, loss aversion from the kink at the reference point (λ). The ±100 TL bet is rejected because of λ even if v is nearly linear for gains.

"w(p) is a probability."

No, it is a decision weight: $w(0.01) + w(0.99) = 0.055 + 0.912 = 0.967 \neq 1$, and $w(0.5) = 0.421$.

"The reference point is always zero."

It can be an expectation, the status quo or a goal. To someone expecting a 10,000 TL bonus, 9,000 TL is a loss.

"In the loss frame everyone gambles."

78% is a majority tendency. The theory explains average behavior; it does not predict individuals.

Noisy but Smart: Quantal Response

- People choose the better option more often, but not always; the bigger the difference, the fewer the errors.
- **Logit choice:** $P(a) = e^{\lambda \cdot u(a)} / \Sigma_b\, e^{\lambda \cdot u(b)}$ ($e \approx 2.718$; e^x grows fast in x). At $\lambda = 0$ choice is random; as $\lambda \to \infty$ it becomes a best response.
- This λ has nothing to do with the loss-aversion $\lambda = 2.25$; it measures how "sharp" the choice is.
- With two options the formula simplifies: $P(\text{better}) = 1 / (1 + e^{-\lambda \cdot \text{diff}})$. With diff 2 and $\lambda = 1$, $P \approx 0.88$.
- **Quantal response equilibrium** (QRE; McKelvey and Palfrey, 1995): every player gives a logit response to the others' noisy choices; as $\lambda \to \infty$ it approaches a Nash equilibrium.

λ	P(payoff 10)
0	0.500
0.5	0.731
1	0.881
2	0.982
∞	1

Two options, payoffs 10 and 8 (diff 2): $P = 1/(1 + e^{-2\lambda})$

A different model, level-k thinking, limits depth of reasoning rather than adding noise: a level-k player treats the others as level k−1 (Nagel, 1995).

Quantal Response: Asymmetric Matching Pennies

	Left	Right
Top	320, 40	40, 80
Bottom	40, 80	80, 40

- p that makes Column indifferent: $40p + 80(1 - p) = 80p + 40(1 - p) \Rightarrow$ **p = 1/2**.
- q that makes Row indifferent: $320q + 40(1 - q) = 40q + 80(1 - q) \Rightarrow$ **q = 1/8**.
- Nash: whatever Row's own 320 is, the probability of Top stays 1/2.

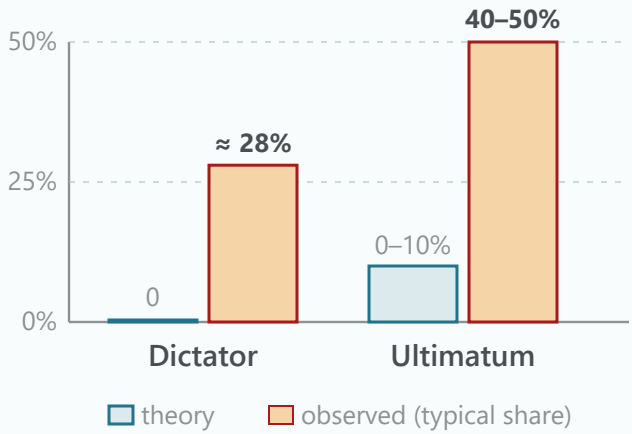
λ	p = P(Top)	q = P(Left)
0	0.50	0.50
0.02	0.83	0.37
0.1	0.71	0.15
1	0.52	0.125
∞ (Nash)	0.50	0.125

The values come from solving the two logit equations jointly (by computer). With small λ, Row is drawn to the 320 and plays Top often (0.83). As λ grows, Row accounts for Column's shift to Right and returns to 1/2.

In experiments, Row players clearly shifted toward the big payoff (320) (Goeree and Holt, 2001). QRE captures this **own-payoff effect** at moderate λ. Common mistake: in a mixed Nash equilibrium a player's probabilities are pinned down by the *opponent's* payoffs; real people also react to their own.

Social Preferences: Is It Only About Money?

- **Dictator game:** one person splits 10 tokens between herself and a recipient, who can do nothing. Someone who cares only about money gives 0.
- In a meta-analysis of over a hundred studies the average share given is **about 28%**; a sizable fraction gives nothing (Engel, 2011).
- **Ultimatum game:** once the recipient can reject, offers rise to 40–50% and low offers are rejected.



In the dictator game the share given comes from fairness or altruism; the extra in the ultimatum game comes from fear of rejection (strategic). **Social preference** models build these motives into the utility function.

Inequity Aversion: The Fehr–Schmidt Model

For two people: $u_i = x_i - \alpha \cdot \max(x_j - x_i, 0) - \beta \cdot \max(x_i - x_j, 0)$, $0 \le \beta \le \alpha$, $\beta < 1$ (Fehr and Schmidt, 1999)

- **α**: the pain of being behind (envy); **β**: the discomfort of being ahead (guilt). People mind being behind more: $\alpha \ge \beta$.
- $\alpha = \beta = 0$ gives the classic player who cares only about money.

Ultimatum

A responder offered s out of 10 accepts if $s - \alpha(10 - 2s) \ge 0 \Leftrightarrow$ **$s \ge 10\alpha/(1 + 2\alpha)$**. $\alpha = 1 \Rightarrow s \ge 3.33$.

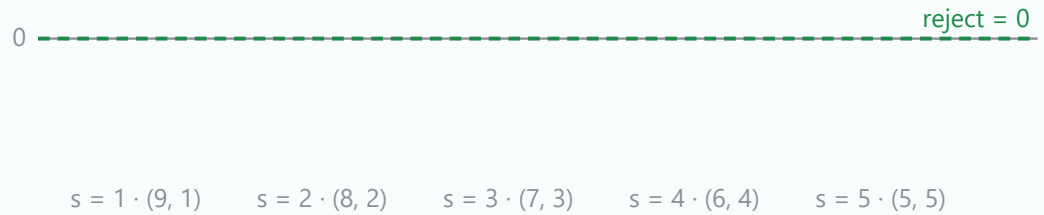
Dictator

Giving $s \le 5$ keeps her ahead, so only the β term matters: $u = (10 - s) - \beta \cdot (10 - 2s) = 10 - 10\beta + s(2\beta - 1)$. If **$\beta > 1/2$** she splits equally; if $\beta < 1/2$ she gives nothing; at $\beta = 1/2$ every share from 0 to 5 gives the same utility.

Public goods

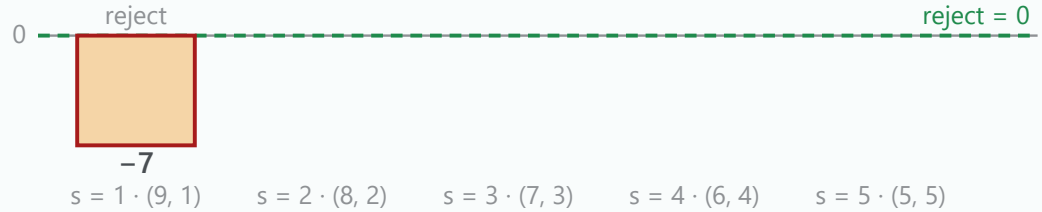
A sufficiently inequity-averse player contributes if the others do and not otherwise: a conditional cooperator.

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



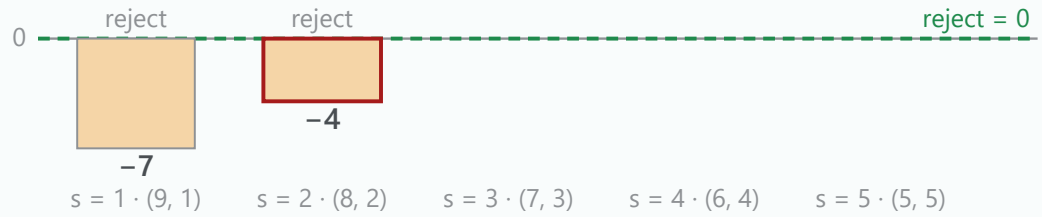
With α = 1 the responder gets u = s − 1·(10 − 2s) by accepting and 0 by rejecting. The proposer offers (10 − s, s).

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



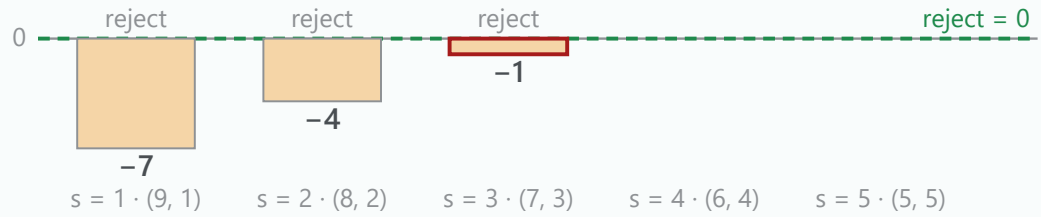
s = 1: u = 1 − 8 = **−7** < 0 ⇒ reject. One token does not make up for the pain of an 8-token gap.

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



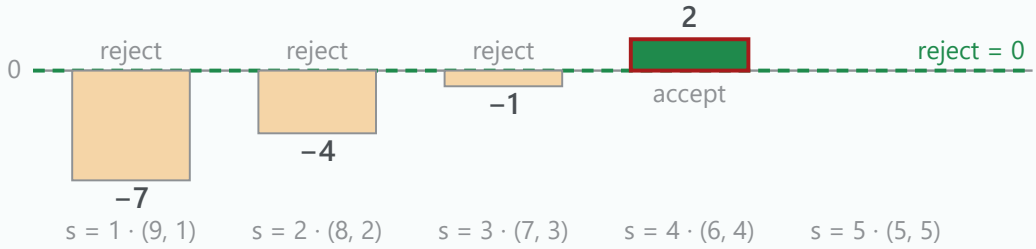
s = 2: u = 2 − 6 = **−4** ⇒ reject.

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



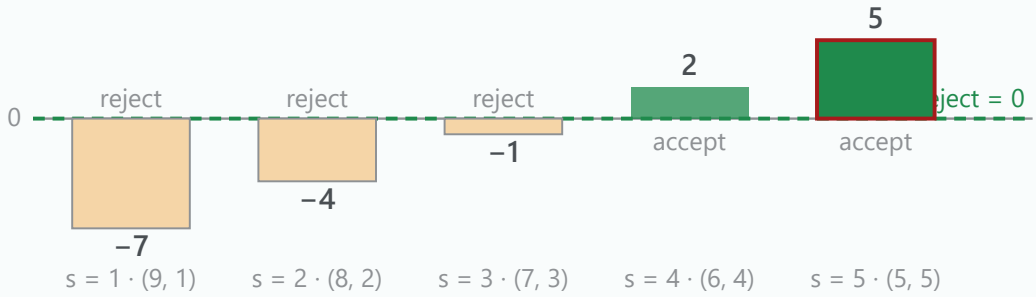
s = 3: u = 3 − 4 = **−1** ⇒ still reject (barely).

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



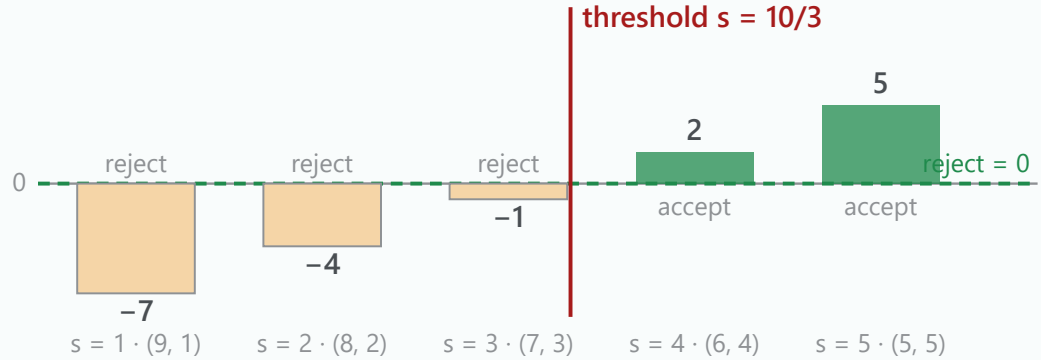
s = 4: u = 4 − 2 = **+2** > 0 ⇒ accept.

Step by Step: The Ultimatum Acceptance Threshold (α = 1)



s = 5: u = **5**, no gap ⇒ accept.

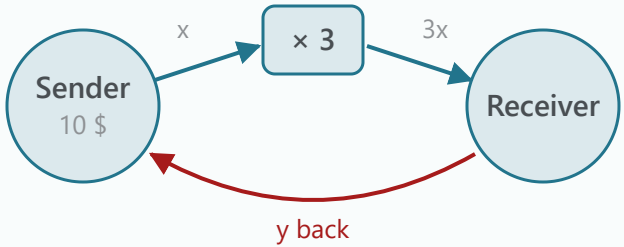
Step by Step: The Ultimatum Acceptance Threshold (α = 1)



Threshold 10α/(1 + 2α) = 10/3 ≈ 3.33: the smallest acceptable whole-token offer is 4. A proposer who knows this offers 4 and keeps 6: rejecting low offers reflects a sense of fairness.

Reciprocity: The Trust Game

- **Trust game** (Berg, Dickhaut and McCabe, 1995): the sender sends any part of $10, the amount is tripled, and the receiver returns any part of it. Played one-shot and fully anonymously.
- **Theory** (money only): the receiver returns nothing; knowing this, the sender sends 0.
- **In fact** senders sent $5.16 on average, and $4.66 came back.
- **Reciprocity:** responding to intentions: kindness for kindness, punishment for harm (Rabin, 1993). Inequity aversion looks at the distribution of outcomes, reciprocity at intentions.



average: x = 5.16 · 3x = 15.48 · y = 4.66

Common mistake: "Trusting paid off." On average the amount returned (4.66) was less than the amount sent (5.16); still, most people sent something.

Collective Decision Making: Individual vs Group Interest

- In group decisions, **social influence**, conformity pressure and shared interests all play a role.
- Communities decide by majority, by consensus or by voting rules.
- Individual interest maximizes one's own utility, collective interest the group's; the two often **conflict**.

Traffic

Everyone takes their own shortest route → total delay is larger than necessary (selfish routing; Pigou's example, the Price of Anarchy).

Work teams

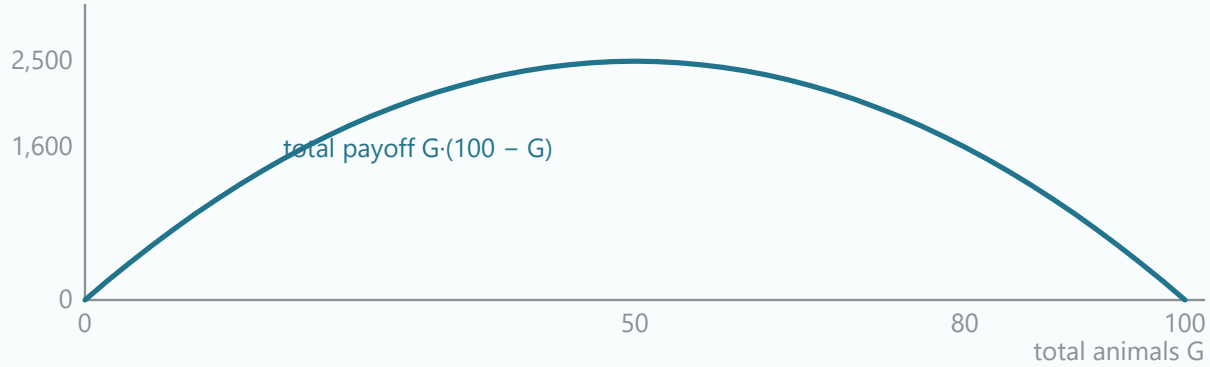
Everyone focuses on their own tasks → the joint project suffers (a public good).

Common resources

Fisheries, pastures, groundwater, the atmosphere → overuse.

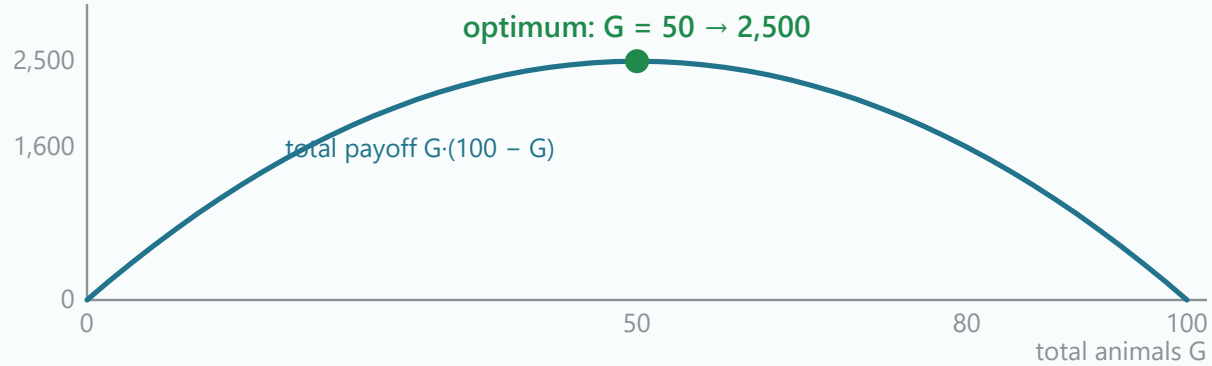
The classic name for the last case: the **tragedy of the commons** (Hardin 1968).

The Tragedy of the Commons: A Numerical Model



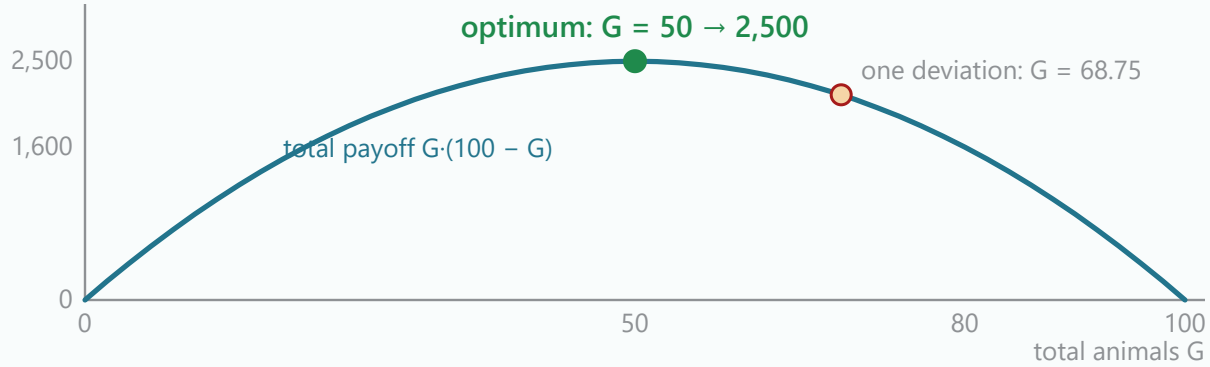
4 herders graze animals on a common pasture. Each animal is worth 100 − G (G = total animals): the more crowded the pasture, the less each animal eats. Herder i's payoff is $u_i = g_i(100 - G)$.

The Tragedy of the Commons: A Numerical Model



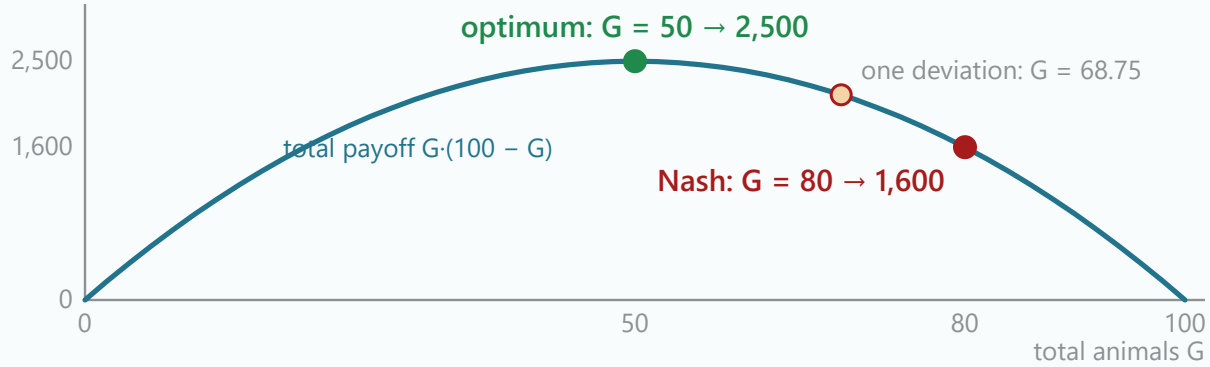
Social optimum: maximize $G(100 - G) \Rightarrow G = 50$, total 2,500. Shared equally, each herder has 12.5 animals and earns 625.

The Tragedy of the Commons: A Numerical Model



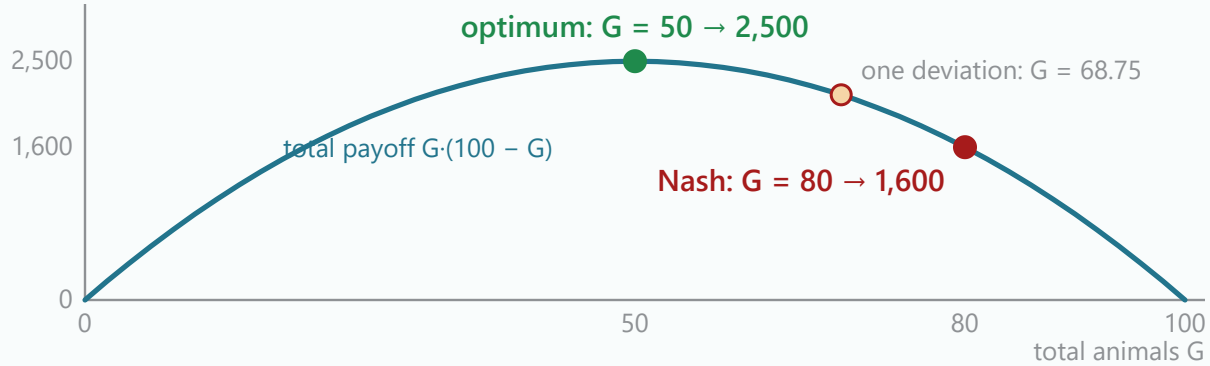
But while the other three graze 37.5 animals in total (12.5 each), herder 1 earns g·(100 − 37.5 − g). Setting the derivative 62.5 − 2g = 0 gives g = 31.25: a payoff of 976.6 > 625. The others' payoff falls to 390.6.

The Tragedy of the Commons: A Numerical Model



When everyone best-responds: g = (100 − 3g)/2 ⇒ **g* = 20**, G = 80. Check: with the others at 60, the best response is (100 − 60)/2 = 20. 400 per herder, 1,600 in total. Herders taking turns best-responding also converge to this equilibrium.

The Tragedy of the Commons: A Numerical Model



The tragedy: equilibrium total is 1,600, 64% of the optimum. Each herder feels only 1/4 of the damage his own animal does to the pasture (an externality).

More Herders, and Remedies

n	g* (per herder)	G*	total	share of optimum
1	50	50	2,500	100%
2	33.33	66.7	2,222	88.9%
4	20	80	1,600	64%
10	9.09	90.9	826	33.1%
100	0.99	99	98	3.9%

In equilibrium $g^* = 100/(n + 1)$; as $n \to \infty$, $G^* \to 100$ and the total $\to 0$

Pigouvian tax

With a tax, a herder earns $g \cdot (100 - G - t)$; the same computation gives the symmetric equilibrium $g = (100 - t)/5$. $t = 37.5 \Rightarrow g = 12.5$, $G = 50$.

Quotas / permits

Rights to 50 animals in total are allocated and can be traded.

Ostrom (Nobel 2009)

Local communities can sustain the resource through monitoring, graduated sanctions and shared rules.

COLLECTIVE DECISIONS

State or Market? Ostrom's Third Way

Hardin's remedy: central control or privatization. Ostrom (1990) found shared principles in local arrangements lasting centuries (Nobel 2009).

1. Clear boundaries
Who may use which resource is well defined.

2. Rules fit local conditions
Rules are tailored to the resource.

3. Collective choice
Those affected by the rules can change them.

4. Monitoring
Monitors accountable to the users.

5. Graduated sanctions
Light first penalties, heavier for repeats.

6. Cheap conflict resolution
Disputes are settled quickly and locally.

7. Right to organize
External authorities recognize local rules.

8. Nested governance
Layered institutions for large resources.

Ostrom's example, Alanya's inshore fishers: spots are drawn by lottery each season, and boats rotate as the fish migrate.

Evaluating Collective Choices: The Veil of Ignorance

Three welfare criteria

The utilitarian ($SW = \Sigma u_i$), egalitarian ($\min u_i$) and Nash social welfare (Πu_i) criteria can rank the same options differently. The commons optimum ($G = 50$, total 2,500) is the utilitarian best.

Veil of ignorance: a policy is chosen without knowing who will be in which position. Give each position equal probability and maximize expected utility (the Laplace rule) and you get the **utilitarian** criterion (Harsanyi, 1955); apply maximin and you get the **egalitarian** one (Rawls, 1971).

Example

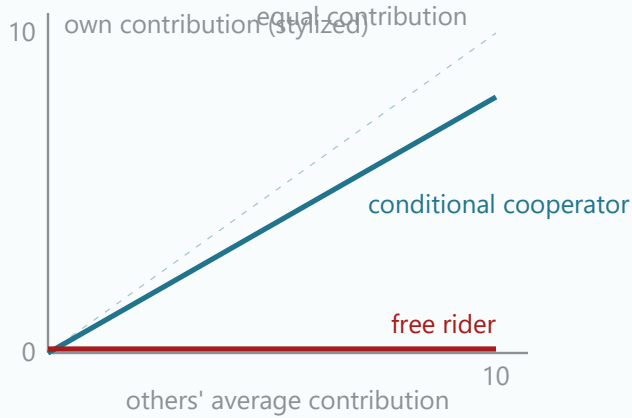
Three positions: $P = (10, 6, 2)$, $R = (5, 5, 5)$. Laplace: P's average $6 > 5 \Rightarrow$ P. Maximin: worst case $2 < 5 \Rightarrow$ R.

Decision under uncertainty

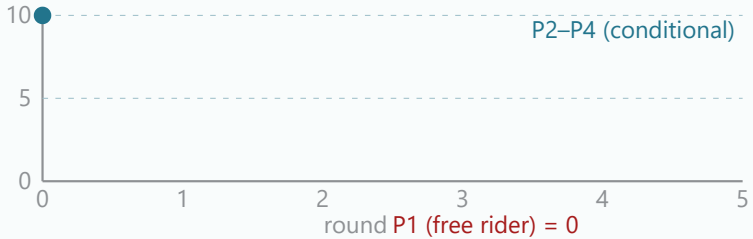
The veil turns social choice into a decision theory problem: the same table recommends different policies depending on the uncertainty criterion.

Public Goods in the Lab: What Do We See?

- Each token contributed returns 0.4 to the contributor and 1.6 to the group; not contributing is dominant, and theory predicts zero contributions.
- **Early rounds:** about 40–60% of the endowment is contributed (Ledyard, 1995).
- **Decay with repetition:** contributions fall round by round and approach free-riding levels in the last rounds.
- **Who?** About half the subjects are conditional cooperators (they want to give as much as others do); about a third are free riders (Fischbacher, Gächter and Fehr, 2001).
- **The mechanism of decay:** conditional cooperators give slightly below the average, free riders pull the average down: contributions erode a little each round.



Code Trace: `conditional(c, rounds)`



round 0: contributions

[0, 10, 10, 10]

payoff u = 10 − k + 0.4·Σk

P1: 22 P2: 12

Model: P1 always gives 0; P2–P4 are **conditional cooperators**: each round they give the others' average from the previous round.

Step 1 / 12

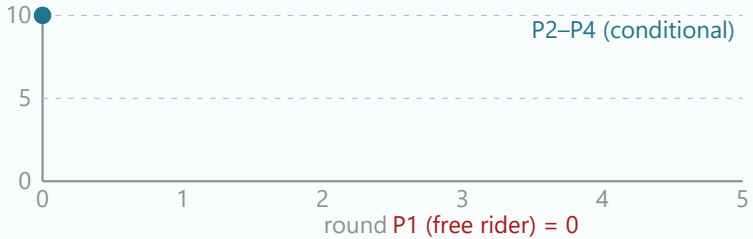
```
1  def conditional(c, rounds):
▶    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`c = [0, 10, 10, 10]`

`history = [c]`

Code Trace: `conditional(c, rounds)`



round 0: contributions

[0, 10, 10, 10]

payoff u = 10 − k + 0.4·Σk

P1: 22 P2: 12

t = 0: for P2 the others' total is 20 ⇒ average 20 / 3 = **6.7** (same for P3, P4).

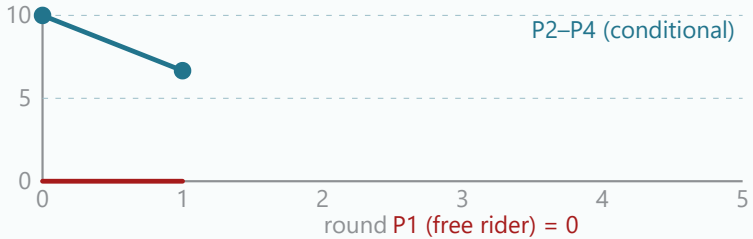
Step 2 / 12

```
1  def conditional(c, rounds):
2    history = [c]
▶    for t in range(rounds):
▶      new = [0]                    # P1 always free-rides
▶      for i in range(1, len(c)):
▶        others = sum(c) - c[i]
▶        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`t = 0` `c = [0, 10, 10, 10]` `others = 20` `new = [0, 6.7, 6.7, 6.7]`

Code Trace: `conditional(c, rounds)`



round 1: contributions
[0, 6.7, 6.7, 6.7]

payoff u = 10 − k + 0.4·Σk
P1: 18 P2: 11.3

Round 1: contributions [0, 6.7, 6.7, 6.7]. The free rider P1 earns more than anyone.

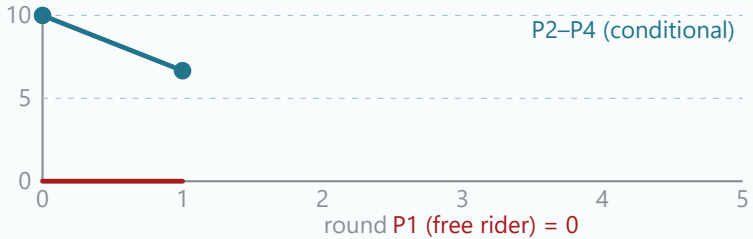
Step 3 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
▶      c = new
▶      history.append(c)
10   return history
```

VARIABLES

`t = 0` `c = [0, 6.7, 6.7, 6.7]` `history = 2 rounds`

Code Trace: `conditional(c, rounds)`



round 1: contributions

[0, 6.7, 6.7, 6.7]

payoff u = 10 − k + 0.4·Σk

P1: 18 P2: 11.3

t = 1: for P2 the others' total is 13.3 ⇒ average 13.3 / 3 = **4.4** (same for P3, P4).

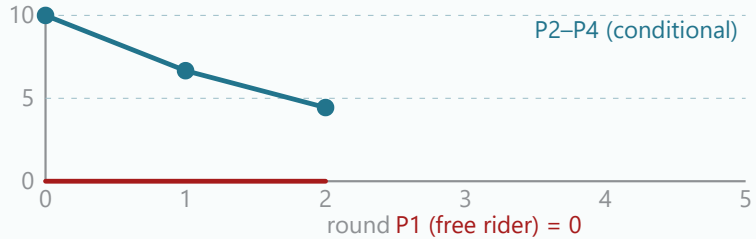
Step 4 / 12

```
1  def conditional(c, rounds):
2    history = [c]
▶    for t in range(rounds):
▶      new = [0]                    # P1 always free-rides
▶      for i in range(1, len(c)):
▶        others = sum(c) - c[i]
▶        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`t = 1` `c = [0, 6.7, 6.7, 6.7]` `others = 13.3` `new = [0, 4.4, 4.4, 4.4]`

Code Trace: `conditional(c, rounds)`



round 2: contributions
[0, 4.4, 4.4, 4.4]

payoff u = 10 − k + 0.4·Σk
P1: 15.3 P2: 10.9

Round 2: contributions [0, 4.4, 4.4, 4.4]. Each round contributions drop to two thirds of the previous value (a one-third decline).

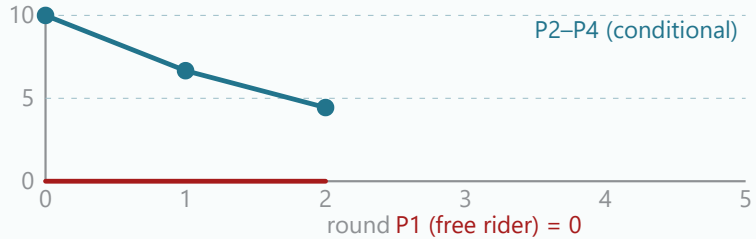
Step 5 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
▶      c = new
▶      history.append(c)
10   return history
```

VARIABLES

`t = 1` `c = [0, 4.4, 4.4, 4.4]` `history = 3 rounds`

Code Trace: `conditional(c, rounds)`



round 2: contributions

[0, 4.4, 4.4, 4.4]

payoff u = 10 − k + 0.4·Σk

P1: 15.3 P2: 10.9

t = 2: for P2 the others' total is 8.9 ⇒ average 8.9 / 3 = **3.0** (same for P3, P4).

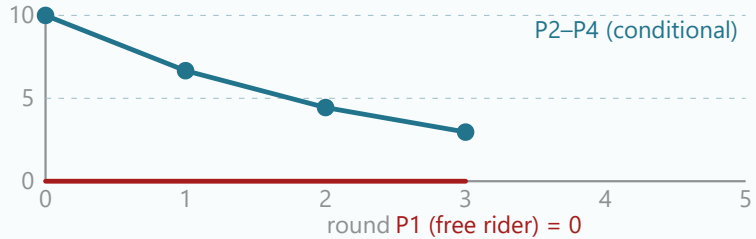
Step 6 / 12

```
1  def conditional(c, rounds):
2    history = [c]
▶    for t in range(rounds):
▶      new = [0]                    # P1 always free-rides
▶      for i in range(1, len(c)):
▶        others = sum(c) - c[i]
▶        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`t = 2` `c = [0, 4.4, 4.4, 4.4]` `others = 8.9` `new = [0, 3.0, 3.0, 3.0]`

Code Trace: `conditional(c, rounds)`



round 3: contributions
[0, 3.0, 3.0, 3.0]

payoff u = 10 − k + 0.4·Σk
P1: 13.6 P2: 10.6

Round 3: contributions [0, 3.0, 3.0, 3.0].

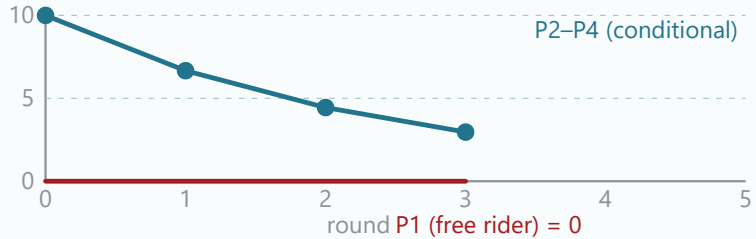
Step 7 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
▶      c = new
▶      history.append(c)
10   return history
```

VARIABLES

`t = 2` `c = [0, 3.0, 3.0, 3.0]` `history = 4 rounds`

Code Trace: `conditional(c, rounds)`



round 3: contributions

[0, 3.0, 3.0, 3.0]

payoff u = 10 − k + 0.4·Σk

P1: 13.6 P2: 10.6

t = 3: for P2 the others' total is 5.9 ⇒ average 5.9 / 3 = **2.0** (same for P3, P4).

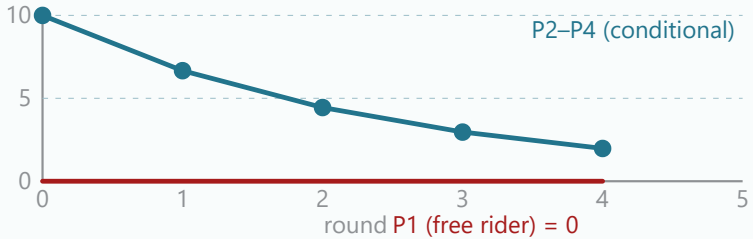
Step 8 / 12

```
1  def conditional(c, rounds):
2    history = [c]
▶    for t in range(rounds):
▶      new = [0]                    # P1 always free-rides
▶      for i in range(1, len(c)):
▶        others = sum(c) - c[i]
▶        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`t = 3` `c = [0, 3.0, 3.0, 3.0]` `others = 5.9` `new = [0, 2.0, 2.0, 2.0]`

Code Trace: `conditional(c, rounds)`



round 4: contributions
[0, 2.0, 2.0, 2.0]

payoff u = 10 − k + 0.4·Σk
P1: 12.4 P2: 10.4

Round 4: contributions [0, 2.0, 2.0, 2.0].

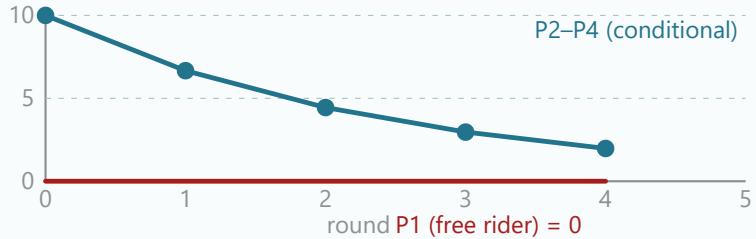
Step 9 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
▶      c = new
▶      history.append(c)
10   return history
```

VARIABLES

`t = 3` `c = [0, 2.0, 2.0, 2.0]` `history = 5 rounds`

Code Trace: `conditional(c, rounds)`



round 4: contributions

[0, 2.0, 2.0, 2.0]

payoff u = 10 − k + 0.4·Σk

P1: 12.4 P2: 10.4

t = 4: for P2 the others' total is 4.0 ⇒ average 4.0 / 3 = **1.3** (same for P3, P4).

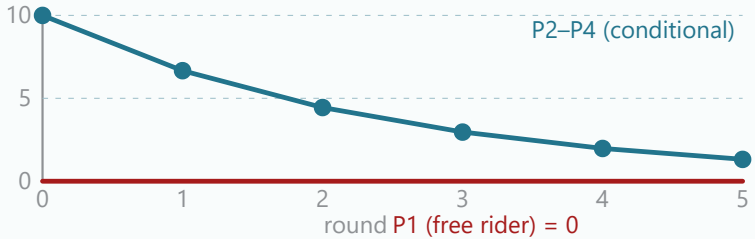
Step 10 / 12

```
1  def conditional(c, rounds):
2    history = [c]
▶    for t in range(rounds):
▶      new = [0]                    # P1 always free-rides
▶      for i in range(1, len(c)):
▶        others = sum(c) - c[i]
▶        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
10   return history
```

VARIABLES

`t = 4` `c = [0, 2.0, 2.0, 2.0]` `others = 4.0` `new = [0, 1.3, 1.3, 1.3]`

Code Trace: `conditional(c, rounds)`



round 5: contributions
[0, 1.3, 1.3, 1.3]

payoff u = 10 − k + 0.4·Σk
P1: 11.6 P2: 10.3

Round 5: contributions [0, 1.3, 1.3, 1.3].

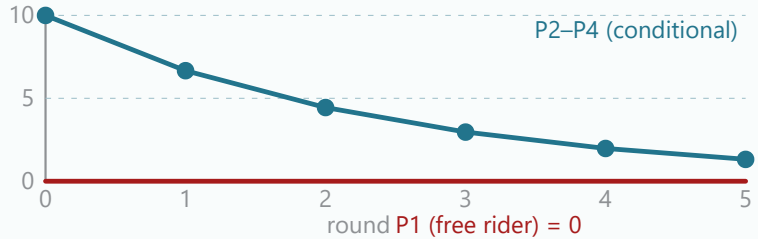
Step 11 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
▶      c = new
▶      history.append(c)
10   return history
```

VARIABLES

`t = 4` `c = [0, 1.3, 1.3, 1.3]` `history = 6 rounds`

Code Trace: `conditional(c, rounds)`



round 5: contributions

[0, 1.3, 1.3, 1.3]

payoff u = 10 − k + 0.4·Σk

P1: 11.6 P2: 10.3

In 5 rounds contributions fall from 10 to 1.3: a single free rider unravels the majority's cooperation (a simple model of the decay seen in the lab).

Step 12 / 12

```
1  def conditional(c, rounds):
2    history = [c]
3    for t in range(rounds):
4      new = [0]                    # P1 always free-rides
5      for i in range(1, len(c)):
6        others = sum(c) - c[i]
7        new.append(others / (len(c) - 1))
8      c = new
9      history.append(c)
▶    return history
```

VARIABLES

`history` = **6 rounds**

With and Without Punishment: Fehr and Gächter (2000)

Design

Groups of 4; each token contributed returns 0.4 to the contributor. After contributions are shown, subjects may pay to punish others. 10 rounds with, 10 without; fixed or reshuffled groups.

Without punishment

Contributions start high, then fall toward zero.

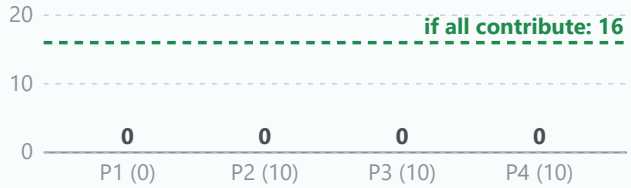
With punishment

Contributions rise, near full cooperation in fixed groups. The furthest below average are punished most.

People punished even in reshuffled groups, where it gains the punisher nothing: **altruistic punishment** (Fehr and Gächter, 2002).

Theory: money-only players never punish, so everyone still gives 0. In the lab people punished and contributions stayed high: both predictions failed.

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

nothing computed yet

Call `punish(c, p)`: P1 free-rides, P2–P4 contribute fully. Each cooperator gives P1 2 punishment points.

Step 1 / 6

```
▶ def punish(c, p):          # p[i][j]: i's points to j
2     n = len(c)
3     share = 1.6 * sum(c) / n
4     u = [10 - c[i] + share for i in range(n)]
5     for i in range(n):
6         for j in range(n):
7             u[i] -= p[i][j]       # costs 1 per point
8             u[j] -= 3 * p[i][j]   # target loses 3
9     return u
```

VARIABLES

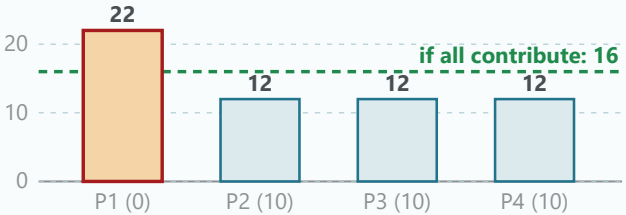
`c = [0, 10, 10, 10]`

`p = p[1][0]=p[2][0]=p[3][0]=2`

CALL STACK

`punish(c, p)`

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

before punishment

share = 1.6 · 30 / 4 = 12. Before punishment u = [22, 12, 12, 12]: the free rider gets 10 more than the contributors.

Step 2 / 6

```
1  def punish(c, p):          # p[i][j]: i's points to j
▶      n = len(c)
▶      share = 1.6 * sum(c) / n
▶      u = [10 - c[i] + share for i in range(n)]
5      for i in range(n):
6          for j in range(n):
7              u[i] -= p[i][j]        # costs 1 per point
8              u[j] -= 3 * p[i][j]    # target loses 3
9      return u
```

VARIABLES

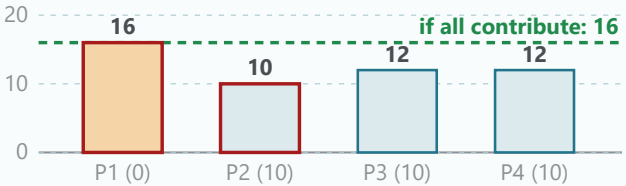
`n = 4` `share = 12.0`

`u = [22, 12, 12, 12]`

CALL STACK

`punish(c, p)`

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

P2 → P1: 2 points

i = 0 (P1) punishes no one. i = 1: P2 pays 2 for 2 points (12 → 10); P1 loses 2 · 3 = 6 (22 → 16).

Step 3 / 6

```
1 def punish(c, p):          # p[i][j]: i's points to j
2     n = len(c)
3     share = 1.6 * sum(c) / n
4     u = [10 - c[i] + share for i in range(n)]
▶     for i in range(n):
▶         for j in range(n):
▶             u[i] -= p[i][j]       # costs 1 per point
▶             u[j] -= 3 * p[i][j]   # target loses 3
9     return u
```

VARIABLES

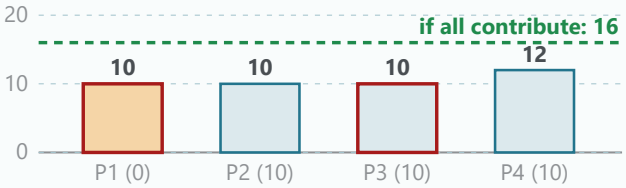
`i = 1` `j = 0` `p[i][j] = 2`

`u = [16, 10, 12, 12]`

CALL STACK

`punish(c, p)`

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

P3 → P1: 2 points

i = 2: P3 also pays 2 (12 → 10); P1 16 → 10. The free rider's advantage is gone.

Step 4 / 6

```
1 def punish(c, p):          # p[i][j]: i's points to j
2     n = len(c)
3     share = 1.6 * sum(c) / n
4     u = [10 - c[i] + share for i in range(n)]
▶     for i in range(n):
▶         for j in range(n):
▶             u[i] -= p[i][j]       # costs 1 per point
▶             u[j] -= 3 * p[i][j]   # target loses 3
9     return u
```

VARIABLES

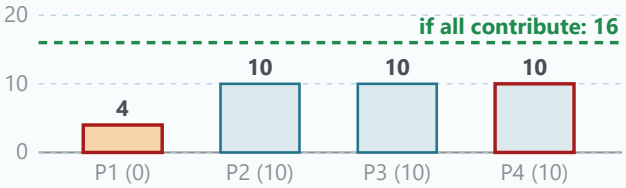
`i = 2` `j = 0` `p[i][j] = 2`

`u = [10, 10, 10, 12]`

CALL STACK

`punish(c, p)`

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

P4 → P1: 2 points

i = 3: P4 also pays 2 (12 → 10); P1 10 → 4.

Step 5 / 6

```
1 def punish(c, p):          # p[i][j]: i's points to j
2     n = len(c)
3     share = 1.6 * sum(c) / n
4     u = [10 - c[i] + share for i in range(n)]
▶     for i in range(n):
▶         for j in range(n):
▶             u[i] -= p[i][j]       # costs 1 per point
▶             u[j] -= 3 * p[i][j]   # target loses 3
9     return u
```

VARIABLES

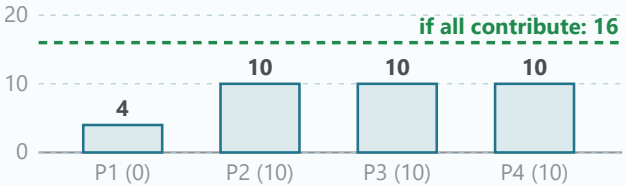
`i = 3` `j = 0` `p[i][j] = 2`

`u = [4, 10, 10, 10]`

CALL STACK

`punish(c, p)`

Code Trace: Does Punishment Make Free Riding Unprofitable?



contributions c = [0, 10, 10, 10]

a punishment point costs 1, removes 3

return u

return u: the free rider gets 4; by contributing (no punishment, everyone 10) he would have had 16. If the threat is credible, contributing pays, but each punisher also lost 2.

Step 6 / 6

```
1 def punish(c, p):        # p[i][j]: i's points to j
2     n = len(c)
3     share = 1.6 * sum(c) / n
4     u = [10 - c[i] + share for i in range(n)]
5     for i in range(n):
6         for j in range(n):
7             u[i] -= p[i][j]       # costs 1 per point
8             u[j] -= 3 * p[i][j]   # target loses 3
▶     return u
```

VARIABLES

`u = [4, 10, 10, 10]`

CALL STACK

`punish(c, p)`

Who Will Punish? Second-Order Free Riding

Punishers (2 points each)	P1	P2	P3	P4
No one	22	12	12	12
P2, P3, P4	4	10	10	10
P2, P3 (P4 does not)	10	10	10	12

Contributions [0, 10, 10, 10]; a punishment point costs 1, removes 3

- Punishment is itself a public good: the punisher bears the cost, everyone shares the benefit (higher contributions). P4, who does not punish, earns the most.
- Backward induction: in the last round punishing is pure cost ⇒ no one punishes ⇒ the threat is not credible.

Antisocial punishment

In public goods experiments in 16 cities, including Istanbul, high contributors were also punished in some places, which could wipe out punishment's boost to cooperation. Antisocial punishment was more common where civic norms and the rule of law were weak (Herrmann, Thöni and Gächter, 2008).

The lesson: punishment is an institution, and how it works depends on the norms around it.

Back to the Game: Public Goods

Theory's answer: 0 tokens

Each token returns only 1.6/4 = 0.4 to you: whatever others do, keeping it is 0.6 better. Giving 0 is dominant; all 0 → 10 each, all 10 → 16 each.

Scenario	equilibrium	per player
Basic	0	10
Punishment (fine 2 if below 3)	3	11.8
Optimum (everyone 10)	10	16

With punishment, giving 3 costs 3 · 0.6 = 1.8 but avoids the fine of 2: 3 becomes dominant.

In practice

In repeated experiments, contributions start at 40–60% of the endowment and decline (Ledyard, 1995). With costly punishment they stay high (Fehr and Gächter, 2000).

Punishment is an **institution** that changes incentives, like the commons tax: it moves the equilibrium from 0 to 3, not to the optimum (10).

Threshold Projects: Contributing Becomes a Coordination Game

A threshold variant of the public goods game: 4 players, 10 tokens each; if total contributions reach 22, everyone gets 12 extra tokens, otherwise contributions are lost. $u_i = 10 - k_i + 12 \cdot [\Sigma k \geq 22]$.

Equilibrium 1: nobody contributes

One player can give at most 10, so the threshold of 22 is out of reach: contributing only loses. Everyone gets 10, total 40.

Equilibrium 2: exactly 22

Every contributor is pivotal: withdrawing 1 token misses the threshold and loses 12; anything extra is wasted. (6, 6, 5, 5) ⇒ (16, 16, 17, 17), total 66.

Which equilibrium?

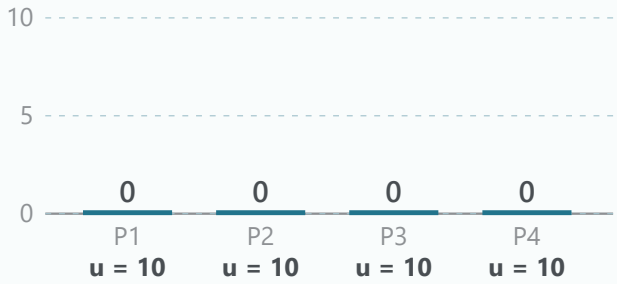
A contributor loses her tokens if the others do not give: giving 6 drops her to 4. Without trust, the zero equilibrium may prevail.

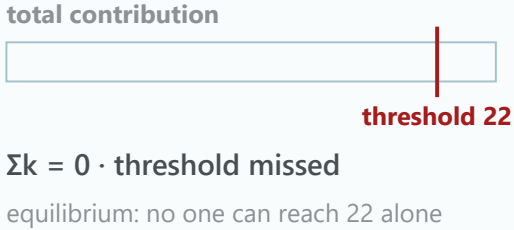
Who gives how much?

The profile (10, 10, 2, 0) is also an equilibrium, with payoffs (12, 12, 20, 22). Threshold equilibria are efficient but may be unequal.

A threshold plus a refund (money returned when a crowdfunding goal is missed) removes the risk of "giving and losing" and makes coordination easier.

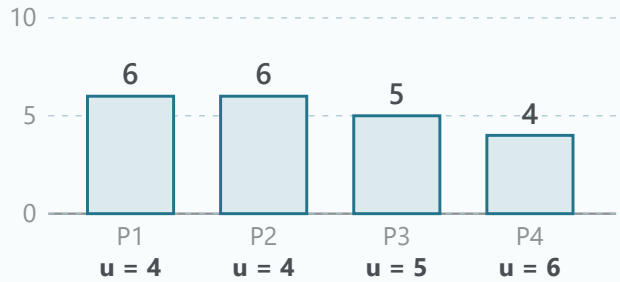
Step by Step: Which Contribution Profiles Are Equilibria?





(0, 0, 0, 0): if P1 gave 10 alone, the total would be 10 < 22 and her 10 tokens would be lost. No one wants to deviate: this all-zero profile is a **Nash equilibrium**; everyone gets 10.

Step by Step: Which Contribution Profiles Are Equilibria?



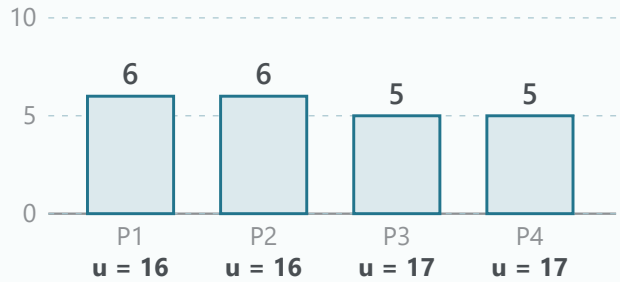


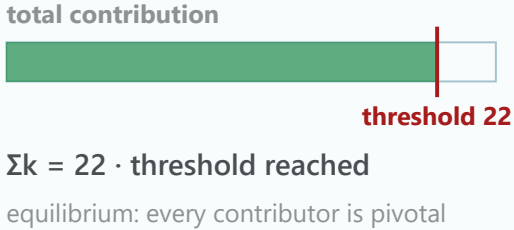
Σk = 21 · threshold missed

not an equilibrium: P4 adds one token

(6, 6, 5, 4): total 21, threshold missed ⇒ payoffs (4, 4, 5, 6). If P4 gives 5, the total is 22 and she gets 10 − 5 + 12 = 17 > 6: **not an equilibrium**.

Step by Step: Which Contribution Profiles Are Equilibria?



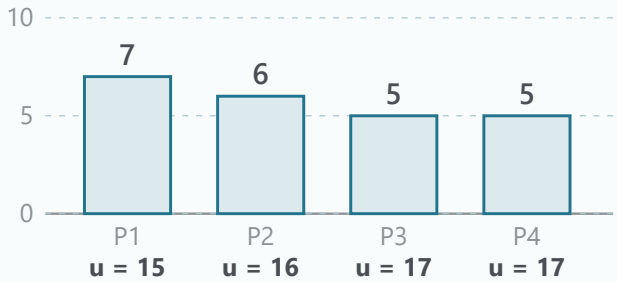


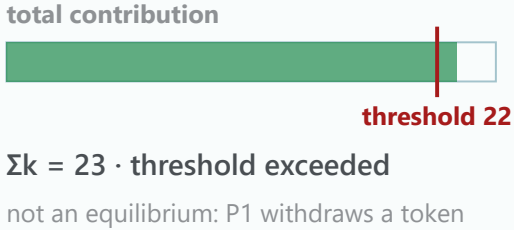
Σk = 22 · threshold reached

equilibrium: every contributor is pivotal

(6, 6, 5, 5): total 22 ⇒ payoffs (16, 16, 17, 17). If P1 drops to 5, the total is 21 and P1 gets 5 < 16. Giving more is wasted: a **Nash equilibrium**, total 66.

Step by Step: Which Contribution Profiles Are Equilibria?



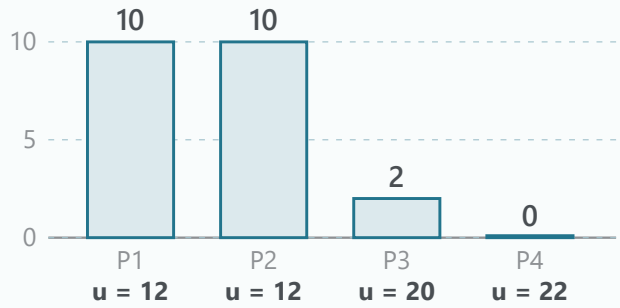


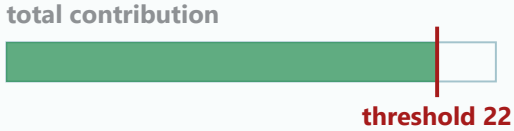
Σk = 23 · threshold exceeded

not an equilibrium: P1 withdraws a token

(7, 6, 5, 5): total 23. If P1 drops to 6, the threshold still holds and she gains 1 token: overcontributing is **not an equilibrium**.

Step by Step: Which Contribution Profiles Are Equilibria?





Σk = 22 · threshold reached

equilibrium, but unequal: (12, 12, 20, 22)

The profile (10, 10, 2, 0) is also an equilibrium (if P3 withdraws, the threshold is missed). There are many efficient equilibria; who gives what is a bargaining and coordination problem.

Wisdom of Crowds

- Galton (1907): at a 1906 livestock fair, 787 guesses of an ox's weight were collected. The median guess was 1,207 lb, the true weight 1,198 lb: an error under 1%.
- Why? Independent errors cancel out.
- **Diversity prediction theorem** (Page, 2007): (average − truth)^2 = average squared individual error − diversity of the predictions. Diversity is the average squared deviation of the predictions from their own mean.
- The identity always holds; for a crowd to predict well it needs **independence** and **diversity**. If everyone copies everyone else, diversity collapses.

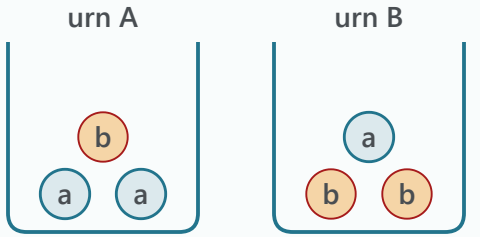
guess x	individual error $(x - 50)^2$	diversity $(x - 54)^2$
40	100	196
45	25	81
55	25	1
60	100	36
70	400	256
avg 54	130	114

Truth 50: crowd error $(54 - 50)^2 = 16 = 130 - 114$

The crowd always predicts at least as well as its average member; the greater the diversity, the larger the gain.

Information Cascades: Everyone Can Be Rationally Wrong

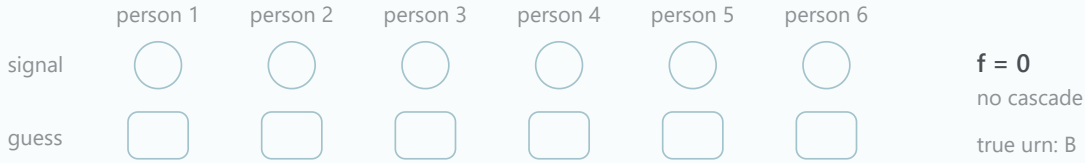
- **Model** (Bikhchandani, Hirshleifer and Welch, 1992): an urn is chosen by a coin flip. In turn, each person privately sees one ball, hears the earlier people's *guesses* (not their balls) and guesses the urn. Each ball matches its urn's color with probability 2/3.
- **Bayes:** each a doubles the odds in favor of A: when a's outnumber b's by f, the odds are 2^f to 1, i.e. $P(A) = 2^f / (2^f + 1)$. On a tie ($P(A) = 1/2$) a person follows her own signal.
- **Cascade:** once the inferred difference is f = 2, a person who sees b has f = 1 and $P(A) = 2/3 > 1/2$: she ignores her own signal and says A. Those after her learn nothing from her guess.
- **Fragility:** a cascade rests on just two signals; a single piece of public information can break it. Fashion, product reviews, panics.



each draw: correct signal with probability 2/3

In the lab (Anderson and Holt, 1997), cascades in this urn experiment did form in most of the situations where they could.

Code Trace: How Does a Cascade Start?



Call `cascade(signals)`; the true urn is B. The difference inferred from earlier guesses is f = 0.

Step 1 / 8

```
▶ def cascade(signals):
▶     f = 0                  # a − b inferred from guesses
▶     guesses = []
4     for s in signals:
5         if f >= 2: t = 'A'      # A cascade
6         elif f <= -2: t = 'B'   # B cascade
7         else:
8             t = 'A' if s == 'a' else 'B'
9             f += 1 if s == 'a' else -1
10        guesses.append(t)
11    return guesses
```

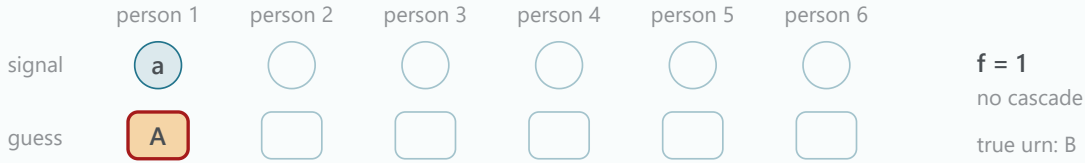
VARIABLES

`signals = [a, a, b, b, b, b]` `f = 0` `guesses = []`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 1, signal a: f = 0 ⇒ follows her signal, **A**. The guess reveals the signal: f = 1, P(A) = 2/3.

Step 2 / 8

```
1  def cascade(signals):
2      f = 0                  # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
5          if f >= 2: t = 'A'      # A cascade
6          elif f <= -2: t = 'B'   # B cascade
▶          else:
▶              t = 'A' if s == 'a' else 'B'
▶              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

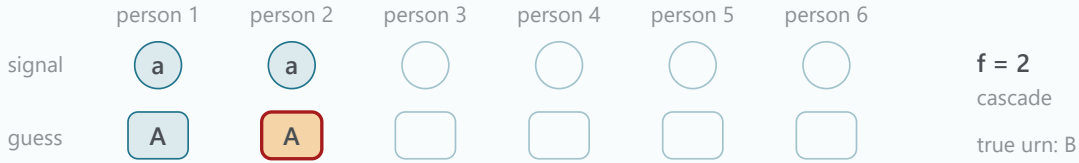
`s = a` `f = 1` `t = A`

`guesses = [A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 2, signal a: f = 1 ⇒ with her own a, P(A) = 0.8, **A**. f = 2.

Step 3 / 8

```
1  def cascade(signals):
2      f = 0                    # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
5          if f >= 2: t = 'A'       # A cascade
6          elif f <= -2: t = 'B'    # B cascade
▶          else:
▶              t = 'A' if s == 'a' else 'B'
▶              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

`s = a` `f = 2` `t = A`

`guesses = [A, A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 3, signal b: f = 2 ⇒ even with her b, P(A) = 2/3 ⇒ **A**. The cascade has started; f no longer changes.

Step 4 / 8

```
1  def cascade(signals):
2      f = 0                    # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
▶          if f >= 2: t = 'A'      # A cascade
6          elif f <= -2: t = 'B'   # B cascade
7          else:
8              t = 'A' if s == 'a' else 'B'
9              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

`s = b` `f = 2` `t = A`

`guesses = [A, A, A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 4, signal b: same computation ⇒ **A**. Her guess carries no information either.

Step 5 / 8

```
1  def cascade(signals):
2      f = 0                    # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
▶          if f >= 2: t = 'A'      # A cascade
6          elif f <= -2: t = 'B'   # B cascade
7          else:
8              t = 'A' if s == 'a' else 'B'
9              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

`s = b` `f = 2` `t = A`

`guesses = [A, A, A, A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 5, signal b ⇒ **A**.

Step 6 / 8

```
1  def cascade(signals):
2      f = 0                    # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
▶          if f >= 2: t = 'A'      # A cascade
6          elif f <= -2: t = 'B'   # B cascade
7          else:
8              t = 'A' if s == 'a' else 'B'
9              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

`s = b` `f = 2` `t = A`

`guesses = [A, A, A, A, A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



Person 6, signal b ⇒ **A**.

Step 7 / 8

```
1  def cascade(signals):
2      f = 0                    # a − b inferred from guesses
3      guesses = []
▶      for s in signals:
▶          if f >= 2: t = 'A'      # A cascade
6          elif f <= -2: t = 'B'   # B cascade
7          else:
8              t = 'A' if s == 'a' else 'B'
9              f += 1 if s == 'a' else -1
▶          guesses.append(t)
11     return guesses
```

VARIABLES

`s = b` `f = 2` `t = A`

`guesses = [A, A, A, A, A, A]`

CALL STACK

`cascade(signals)`

Code Trace: How Does a Cascade Start?



return: everyone said A, yet 4 of the 6 signals were b. Pooling the signals would give f = −2, P(A) = 0.2 ⇒ B. Everyone was rational; the information did not accumulate.

Step 8 / 8

```
1  def cascade(signals):
2      f = 0                  # a − b inferred from guesses
3      guesses = []
4      for s in signals:
5          if f >= 2: t = 'A'     # A cascade
6          elif f <= -2: t = 'B'  # B cascade
7          else:
8              t = 'A' if s == 'a' else 'B'
9              f += 1 if s == 'a' else -1
10         guesses.append(t)
▶      return guesses
```

VARIABLES

`guesses = [A, A, A, A, A, A]`

CALL STACK

`cascade(signals)`

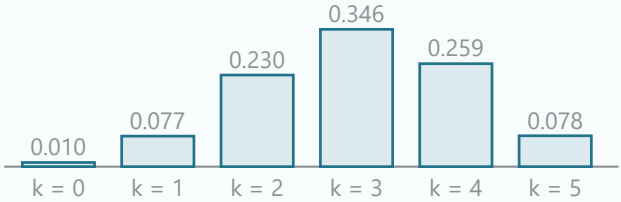
The Condorcet Jury Theorem: How Smart Is the Majority?

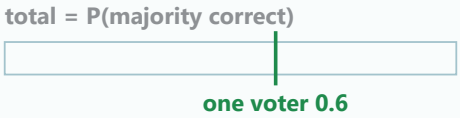
- **Question:** there is a right answer (is the defendant guilty or not?). How does majority voting change the chance of finding it? This is different from the Condorcet paradox, where preferences cycle.
- **Theorem** (Condorcet, 1785): if n (odd) voters vote *independently* and each picks the right answer with probability $p > 1/2$, the probability that the majority is right increases with n and tends to 1.
- If $p < 1/2$ it is the reverse: as the crowd grows, the majority is almost surely wrong.
- $P_n = \Sigma_{k > n/2} C(n, k) \cdot p^k (1 - p)^{n - k}$

n	p = 0.6	p = 0.4
1	0.600	0.400
3	0.648	0.352
5	0.683	0.317
11	0.753	0.247
51	0.926	0.074
101	0.979	0.021

Probability that the majority decides correctly

Code Trace: A Five-Member Jury





total = 0.0000

Call `jury(5, 0.6)`: the majority is right if at least 3 of the 5 voters vote correctly. k: number of correct votes.

Step 1 / 5

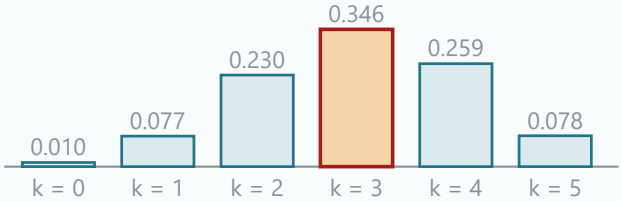
```
1 from math import comb
▶ def jury(n, p):
▶     total = 0
4     for k in range(n // 2 + 1, n + 1):
5         term = comb(n, k) * p**k * (1 - p)**(n - k)
6         total += term
7     return total
▶ print(round(jury(5, 0.6), 4))
```

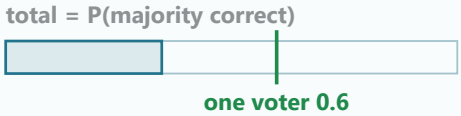
VARIABLES

`n = 5` `p = 0.6` `total = 0`

OUTPUT

Code Trace: A Five-Member Jury





total = 0.3456

k = 3: $C(5, 3) \cdot 0.6^3 \cdot 0.4^2 = 10 \cdot 0.2160 \cdot 0.1600 =$ **0.3456**. Total 0.3456.

Step 2 / 5

```
1 from math import comb
2 def jury(n, p):
3     total = 0
▶     for k in range(n // 2 + 1, n + 1):
▶         term = comb(n, k) * p**k * (1 - p)**(n - k)
▶         total += term
7     return total
8 print(round(jury(5, 0.6), 4))
```

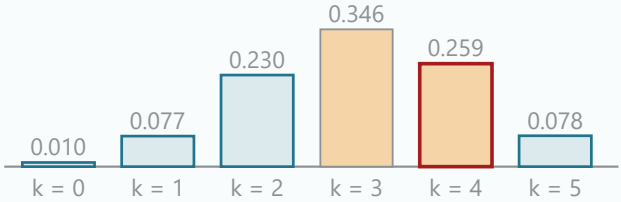
VARIABLES

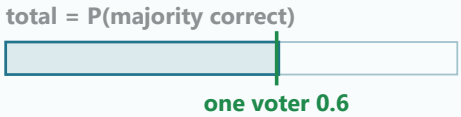
`k = 3` `term = 0.3456`

`total = 0.3456`

OUTPUT

Code Trace: A Five-Member Jury





total = 0.6048

k = 4: C(5, 4)·0.6^4·0.4^1 = 5·0.1296·0.4000 = **0.2592**. Total 0.6048.

Step 3 / 5

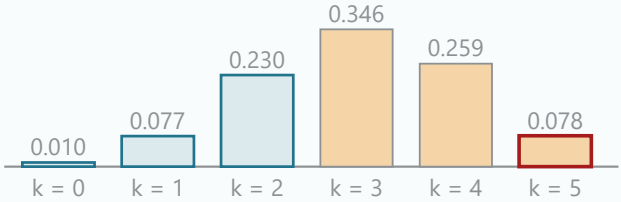
```
1 from math import comb
2 def jury(n, p):
3     total = 0
▶     for k in range(n // 2 + 1, n + 1):
▶         term = comb(n, k) * p**k * (1 - p)**(n - k)
▶         total += term
7     return total
8 print(round(jury(5, 0.6), 4))
```

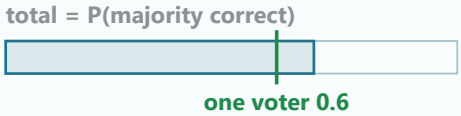
VARIABLES

`k = 4` `term = 0.2592` `total = 0.6048`

OUTPUT

Code Trace: A Five-Member Jury





total = 0.6826

k = 5: C(5, 5)·0.6^5·0.4^0 = 1·0.0778·1 = **0.0778**. Total 0.6826.

Step 4 / 5

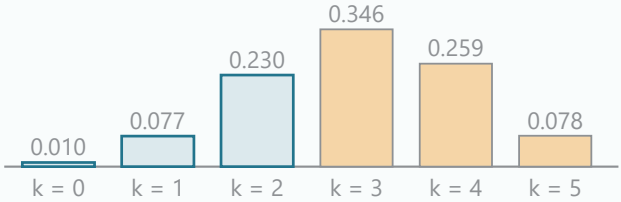
```
1 from math import comb
2 def jury(n, p):
3     total = 0
▶     for k in range(n // 2 + 1, n + 1):
▶         term = comb(n, k) * p**k * (1 - p)**(n - k)
▶         total += term
7     return total
8 print(round(jury(5, 0.6), 4))
```

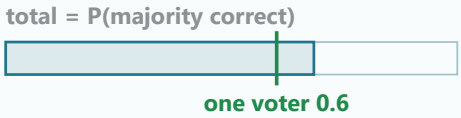
VARIABLES

`k = 5` `term = 0.0778` `total = 0.6826`

OUTPUT

Code Trace: A Five-Member Jury





total = 0.6826

return: P_5 = 0.6826 > 0.6. Five independent, moderately competent voters decide better than one.

Step 5 / 5

```
1 from math import comb
2 def jury(n, p):
3     total = 0
4     for k in range(n // 2 + 1, n + 1):
5         term = comb(n, k) * p**k * (1 - p)**(n - k)
6         total += term
▶     return total
▶ print(round(jury(5, 0.6), 4))
```

VARIABLES

`total = 0.6826`

OUTPUT

```
0.6826
```

When Does the Jury Theorem Break Down?

Dependence

If 4 jurors on a 5-member jury follow an opinion leader, the majority is that one person's vote: P = 0.6. With independent votes it was 0.683. Cascades and shared information sources destroy independence.

Competence

On a question where $p < 1/2$ (a misleading clue), a larger crowd makes the error certain: with $p = 0.4$ and 101 voters, the majority is right with probability only 2.1%.

Strategic voting

A voter who reasons that her vote matters only when it is pivotal may rationally vote against her own signal (Austen-Smith and Banks, 1996). Under unanimity this can raise the chance of convicting the innocent (Feddersen and Pesendorfer, 1998).

The wisdom of crowds, cascades and the jury theorem teach the same lesson: a group knows only as much as it can pool of its members' **independent** information.

SUMMARY

Common Mistakes: Collective Decisions

"Crowds are always wise."

Independence and p > 1/2 are needed. In the cascade, all six people were wrong.

"Those who follow a cascade are irrational."

Each acted correctly by Bayes' rule; the problem is that information does not accumulate.

"Punishment always increases cooperation."

Punishment is costly and invites second-order free riding; antisocial punishment can cancel its effect.

"A threshold project has a single equilibrium."

Both zero contribution and every profile summing to exactly 22 are equilibria.

"The commons needs either the state or the market."

Ostrom's local institutions are a third way.

"The jury theorem and the Condorcet paradox are the same."

The theorem is about finding the truth, the paradox about cyclic preferences.

Summary

Prospect theory

Reference point, $\lambda \approx 2.25$, probability weighting: the ±100 TL bet is rejected, the acceptance threshold is ≈ 251 TL; the endowment effect and framing share the same root.

Boundedly rational equilibrium

Quantal response: better options are chosen more often; in asymmetric matching pennies, play shifts toward the big payoff instead of Nash's 1/2.

Social preferences

Fehr–Schmidt: a responder with $\alpha = 1$ rejects offers below 10/3 out of 10; reciprocity in the trust game.

The commons

With 4 herders, the equilibrium total is 1,600 vs an optimum of 2,500; taxes, quotas and Ostrom's local institutions.

Public goods and punishment

Contributions start at 40–60% and erode; punishment sustains cooperation but can breed second-order free riding and antisocial punishment.

Information aggregation

The diversity theorem, information failing to accumulate in a cascade, the jury theorem (5 voters, $p = 0.6 \Rightarrow 0.683$).

Sample Problems

1. With the prospect theory parameters, is a +300 / −150 coin-flip bet accepted?
2. In the commons model with n = 9 herders, what are the equilibrium G* and the total payoff? What tax achieves the optimum?
3. In the punishment variant of the public goods game, what would the equilibrium contribution be if the fine were 1 token instead of 2?
4. In the Fehr–Schmidt model, at least how many of 10 tokens does a responder with $\alpha = 2$ demand?
5. In the Condorcet jury theorem with $p = 0.7$, with what probability is the majority of a 3-member jury right?

▶ **Show answers**

References

- H. A. Simon, “A behavioral model of rational choice”, *Quarterly J. Economics* 69, 1955.
- D. Kahneman, A. Tversky, “Prospect theory: An analysis of decision under risk”, *Econometrica* 47(2), 1979.
- A. Tversky, D. Kahneman, “The framing of decisions and the psychology of choice”, *Science* 211, 1981; “Advances in prospect theory”, *J. Risk and Uncertainty* 5, 1992.
- D. Kahneman, J. L. Knetsch, R. H. Thaler, “Experimental tests of the endowment effect and the Coase theorem”, *J. Political Economy* 98(6), 1990.
- R. D. McKelvey, T. R. Palfrey, “Quantal response equilibria for normal form games”, *Games and Economic Behavior* 10(1), 1995; J. K. Goeree, C. A. Holt, “Ten little treasures of game theory and ten intuitive contradictions”, *American Economic Review* 91(5), 2001.
- E. Fehr, K. M. Schmidt, “A theory of fairness, competition, and cooperation”, *Quarterly J. Economics* 114(3), 1999; C. Engel, “Dictator games: A meta study”, *Experimental Economics* 14(4), 2011.
- J. Berg, J. Dickhaut, K. McCabe, “Trust, reciprocity, and social history”, *Games and Economic Behavior* 10(1), 1995; M. Rabin, “Incorporating fairness into game theory and economics”, *American Economic Review* 83(5), 1993.
- G. Hardin, “The tragedy of the commons”, *Science* 162, 1968; E. Ostrom, *Governing the Commons*, Cambridge University Press, 1990.

References (2/2)

- J. O. Ledyard, “Public goods: A survey of experimental research”, in J. H. Kagel, A. E. Roth (eds.), *The Handbook of Experimental Economics*, Princeton University Press, 1995.
- U. Fischbacher, S. Gächter, E. Fehr, “Are people conditionally cooperative? Evidence from a public goods experiment”, *Economics Letters* 71(3), 2001.
- E. Fehr, S. Gächter, “Cooperation and punishment in public goods experiments”, *American Economic Review* 90(4), 2000; “Altruistic punishment in humans”, *Nature* 415, 2002.
- B. Herrmann, C. Thöni, S. Gächter, “Antisocial punishment across societies”, *Science* 319, 2008.
- F. Galton, “Vox populi”, *Nature* 75, 1907; S. E. Page, *The Difference*, Princeton University Press, 2007.
- S. Bikhchandani, D. Hirshleifer, I. Welch, “A theory of fads, fashion, custom, and cultural change as informational cascades”, *J. Political Economy* 100(5), 1992; L. R. Anderson, C. A. Holt, “Information cascades in the laboratory”, *American Economic Review* 87(5), 1997.
- Marquis de Condorcet, *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*, 1785; D. Austen-Smith, J. S. Banks, “Information aggregation, rationality, and the Condorcet jury theorem”, *American Political Science Review* 90(1), 1996; T. Feddersen, W. Pesendorfer, “Convicting the innocent”, *American Political Science Review* 92(1), 1998.
- R. Nagel, “Unraveling in guessing games: An experimental study”, *American Economic Review* 85(5), 1995.
- J. C. Harsanyi, “Cardinal welfare, individualistic ethics, and interpersonal comparisons of utility”, *J. Political Economy* 63, 1955; J. Rawls, *A Theory of Justice*, Harvard University Press, 1971.

The End

Chapter 12: Behavioral and Collective Decisions

Previous: Chapter 11: Algorithmic Fairness and Ethics · Next: Chapter 13: Learning and Adaptation