Chapter 14: Future Directions

Algorithmic Game Theory

AI agents | Learned mechanisms | Blockchain mechanisms | Computation and scale | Course map

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_14_Future_Directions.html

INTRODUCTION

Warm-up: An AI Agent Tournament

The rules

Agents play the chosen game for 60 or 120 rounds; in a full tournament every agent meets every other agent once and agents are ranked by total payoff.

Think about it

- Which agent comes first: one that always cooperates, one that always defects, one that plays at random, or one that best-responds to the opponent's most frequent move?
- Does the winner of one game also win another?

	1	2	3	4
1 · Random		?	?	?
2 · Always Cooperate			?	?
3 · Always Defect				?
4 · Best Response				

Prisoner's Dilemma, Level 1: 60 rounds per match; each agent plays each of the other three once (6 matches).

Learning Outcomes

New players

Identify the game-theoretic questions AI agents raise in bargaining and pricing.

Design from data

Explain with a numerical example how a mechanism is learned from data, and the idea of sample complexity.

On-chain mechanisms

Interpret transaction fee auctions, the base fee rule and MEV (value extracted from the power to order transactions).

Tying the course together

Relate all the chapters of the course on a single map.

Algorithmic Game Theory: A Growing Field

- Applications are growing in industry and academia: autonomous vehicles, online platforms, energy grids.
- Technology trends: AI and machine learning, big data and real-time decisions, blockchains and smart contracts.

Fairness and ethics
Fair and transparent mechanisms.

Dynamic, learning systems
Adaptive autonomous agents.

Large networks
Strategic interaction among millions of players.

Incentives and policy
Designing rules for social welfare and sustainability.

This chapter shows how the course's tools are used in these new areas, and where they fall short.

Application Areas

Finance

Algorithmic trading, portfolio management; trading algorithms play against each other.

Traffic and logistics

Autonomous vehicle networks, route choice; selfish routing and Braess.

Online platforms

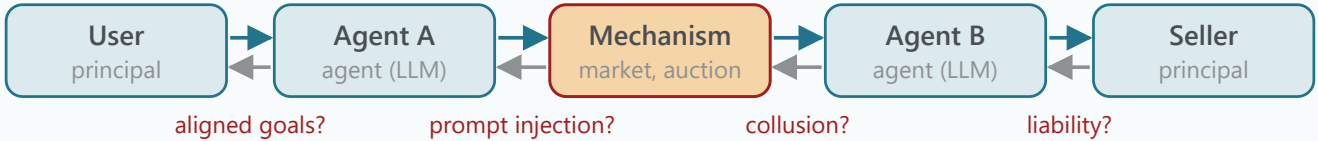
Recommender systems, ad auctions, social network analysis.

Energy grids

Load balancing, demand response, electricity markets.

Strategic AI Agents

- Agents based on large language models (LLMs) can shop, bargain, plan and share resources on behalf of users.
- The agent is now a **player**: an actor with goals, information and a strategy. Game theory's questions are back on the table.



Goal alignment

Does the agent share the user's goal? The principal–agent problem.

The rationality assumption

LLM behavior can be sensitive to framing and deviate from the Nash prediction.

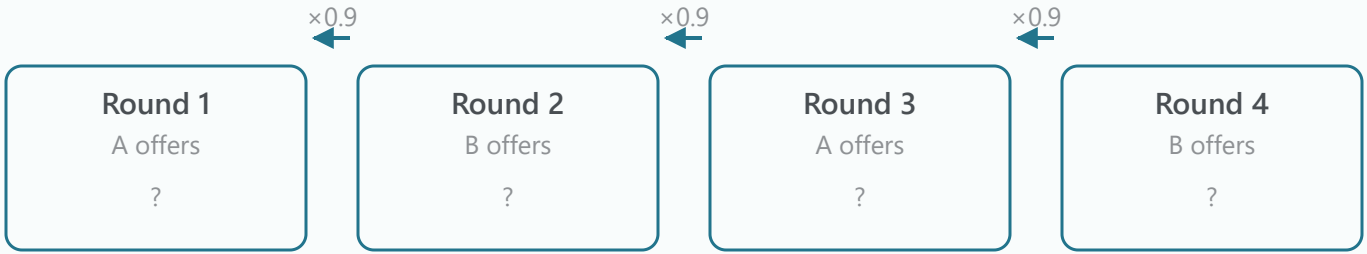
Manipulation

Instructions hidden in the other agent's messages (prompt injection).

Identity and liability

On whose behalf is it speaking, who is liable for errors, how is a contract verified?

Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share									

100 TL to split. A proposes in odd rounds, B in even rounds; each delay shrinks the value by 10% ($\delta = 0.9$). Round 4 is the last. Backward induction.

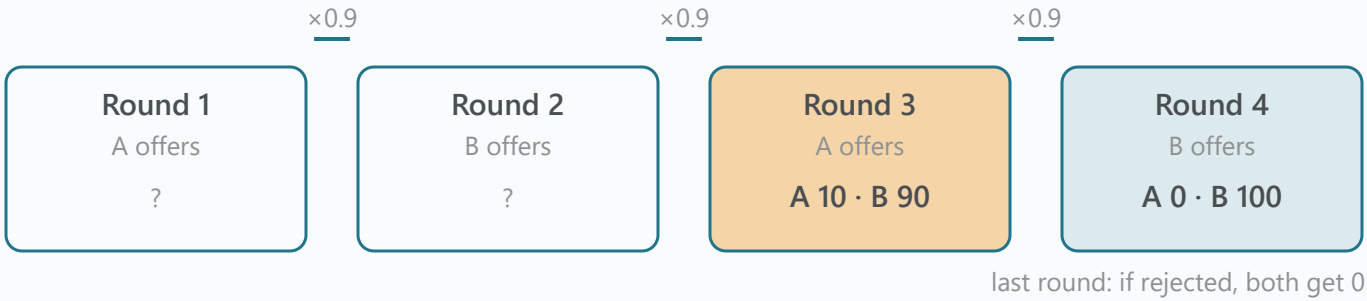
Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share									

Round 4: B proposes. If A rejects, both get 0 → A accepts anything (an indifferent player is assumed to accept): **B 100, A 0**.

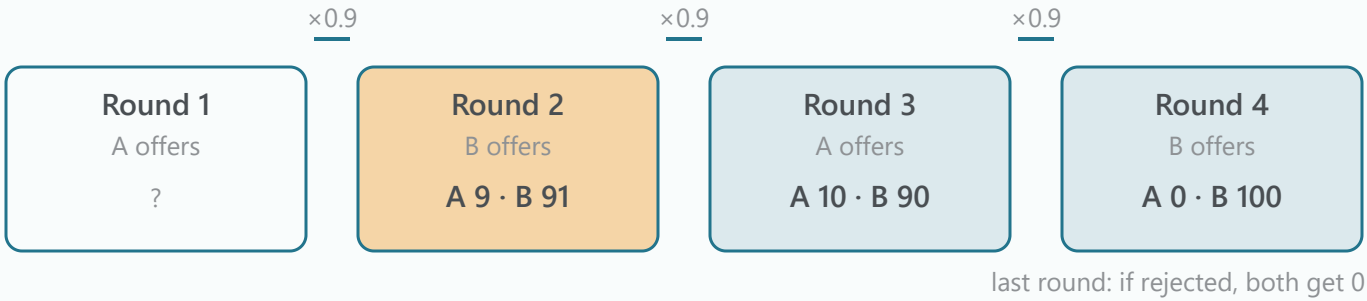
Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share									

Round 3: if B rejects, she gets 100 in round 4, worth 0.9·100 = 90 today. A offers B 90: **A 10, B 90**.

Step by Step: Agent-to-Agent Bargaining



Number of rounds T	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**∞**
A's share									

Round 2: A's value of waiting is 0.9·10 = 9. B offers A 9: **A 9, B 91**.

Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share									

Round 1: B's value of waiting is 0.9·91 = 81.9. A gets 18.1 and agreement is **immediate** (subgame perfect equilibrium).

Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share	100	10	91	18.1	83.7	24.7	77.8	30	52.6

The share oscillates with the number of rounds; T = 1 is the ultimatum game (A takes everything). With an infinite horizon (Rubinstein, 1982) the proposer's share is $1/(1 + \delta)$ = **52.6**.

Step by Step: Agent-to-Agent Bargaining



Number of rounds T	1	2	3	4	5	6	7	8	∞
A's share	100	10	91	18.1	83.7	24.7	77.8	30	52.6

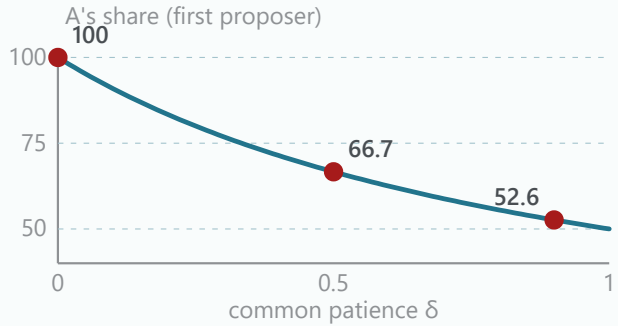
The lesson for agents: **patience (δ) and the last word are bargaining power**. A deadline decides who makes the last offer, so knowledge of the horizon is strategic too.

Infinite Horizon: The Rubinstein Solution

- No last round, so backward induction cannot start; use **stationary** strategies: A always keeps x* when proposing, B always leaves y* to A.
- B is indifferent: $1 - x^* = \delta_B \cdot (1 - y^*)$; so is A: $y^* = \delta_A \cdot x^*$.
- Substituting: $1 - x^* = \delta_B \cdot (1 - \delta_A \cdot x^*) \Rightarrow x^* \cdot (1 - \delta_A \delta_B) = 1 - \delta_B$:

$$x^* = (1 - \delta_B) / (1 - \delta_A \delta_B)$$

Equal patience δ: $x^* = 1/(1 + \delta)$; $\delta = 0.9$ gives $100/1.9 \approx$ **52.6**. Accepted in round 1: no delay, no loss.



δ_A	δ_B	A	B
0.9	0.9	52.6	47.4
0.9	0.8	71.4	28.6
0.8	0.9	35.7	64.3
0.99	0.9	91.7	8.3

For agent design: **patience is power**. An impatient agent (low δ) gets less; with incomplete information, so does one that merely *appears* impatient.

AI AGENTS

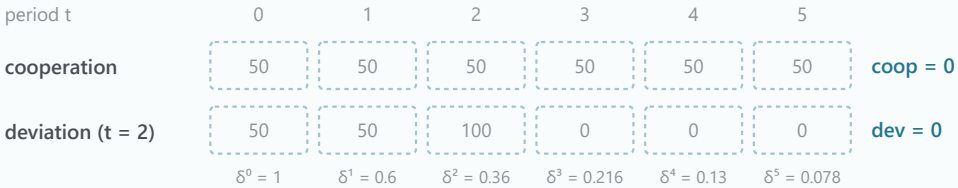
Can Pricing Algorithms Collude?

- Two firms sell the same product; at the monopoly price total profit is 100 (50 each). If one undercuts, profit drops to zero (Bertrand competition: the cheaper firm takes the whole market).
- The grim trigger strategy: "Price high until the rival undercuts; if it does, compete forever."
- The structure is the repeated Prisoner's Dilemma: a deviator gets 100 for one period, then 0; a complier gets 50 every period. The trigger condition $50/(1-\delta) \geq 100 \iff \mathbf{\delta \geq 1/2}$.

Simulations show that pricing agents using reinforcement learning can learn supracompetitive prices without communicating (Calvano et al., 2020).

The regulatory question: without an explicit agreement, is there **algorithmic collusion**? Competition law has traditionally rested on "agreements between people"; algorithms that learn high prices without communicating strain this framework.

Code Trace: Does Deviating Pay Under a Trigger?



Two paths: **cooperate** at 50 each period, or **deviate** at t = 2 for 100, then 0 under the trigger. δ = 0.6, 6 periods.

Step 1 / 9

```
1  def compare(delta, s, T):
▶      coop, dev = 0.0, 0.0          # cooperate / deviate
3      for t in range(T):
4          coop += delta**t * 50
5          if t < s:
6              dev += delta**t * 50
7          elif t == s:
8              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

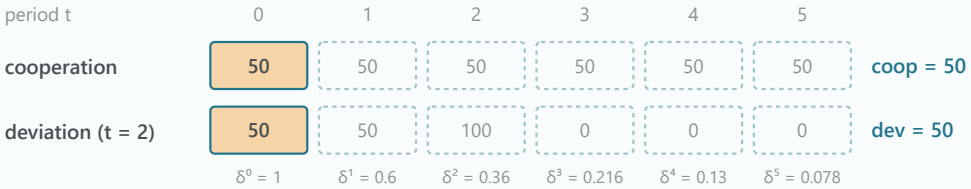
VARIABLES

`delta = 0.6` `s = 2` `T = 6`

`coop = 0.0` `dev = 0.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 0: both paths earn 50 ($\delta^0 = 1$).

Step 2 / 9

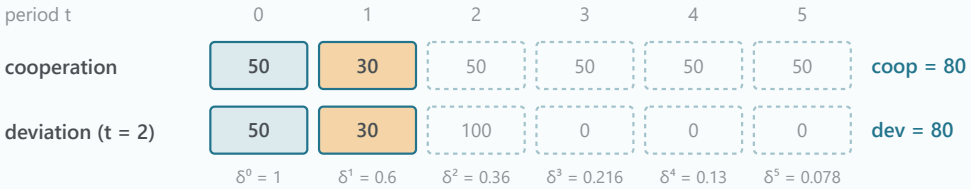
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
▶      for t in range(T):
▶          coop += delta**t * 50
▶          if t < s:
▶              dev += delta**t * 50
7          elif t == s:
8              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 0` `coop = 50.0` `dev = 50.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 1: +0.6·50 = 30; still equal: 80.

Step 3 / 9

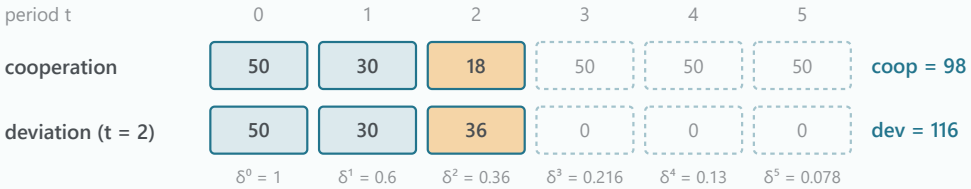
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
▶      for t in range(T):
▶          coop += delta**t * 50
▶          if t < s:
▶              dev += delta**t * 50
7          elif t == s:
8              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 1` `coop = 80.0` `dev = 80.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 2: A deviates and takes the market: +0.36·100 = 36 vs. +18. Deviation **18 ahead**.

Step 4 / 9

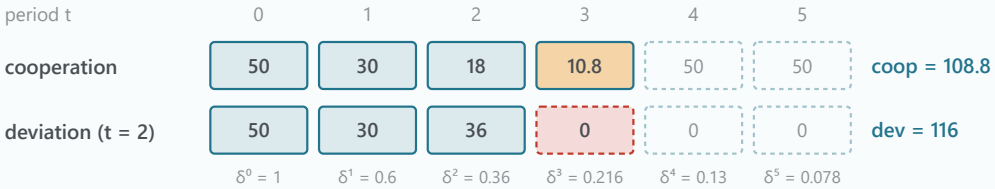
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
▶      for t in range(T):
▶          coop += delta**t * 50
5          if t < s:
6              dev += delta**t * 50
▶          elif t == s:
▶              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 2` `coop = 98.0` `dev = 116.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 3: the trigger fires; deviation gets 0, cooperation +10.8.

Step 5 / 9

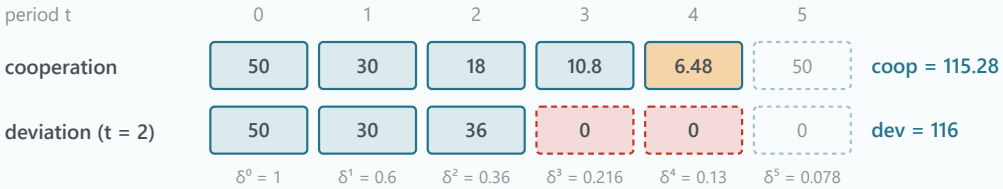
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
▶      for t in range(T):
▶          coop += delta**t * 50
▶          if t < s:
6              dev += delta**t * 50
▶          elif t == s:
8              dev += delta**t * 100  # A undercuts
▶          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 3` `coop = 108.8` `dev = 116.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 4: cooperation +6.48 = 115.28; gap 0.72.

Step 6 / 9

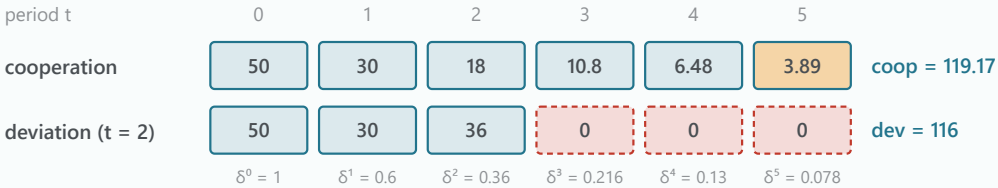
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
▶      for t in range(T):
▶          coop += delta**t * 50
▶          if t < s:
6              dev += delta**t * 50
▶          elif t == s:
8              dev += delta**t * 100  # A undercuts
▶          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 4` `coop = 115.28` `dev = 116.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



t = 5: +3.89, cooperation **pulls ahead**: 119.17 > 116. The gain of 18 is repaid in lost future profit.

Step 7 / 9

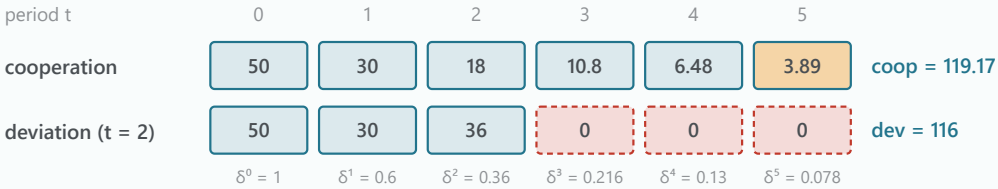
```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
       for t in range(T):
           coop += delta**t * 50
           if t < s:
6              dev += delta**t * 50
           elif t == s:
8              dev += delta**t * 100  # A undercuts
           # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

VARIABLES

`t = 5` `coop = 119.168` `dev = 116.0`

OUTPUT

Code Trace: Does Deviating Pay Under a Trigger?



Printed. With an infinite horizon: 50/(1 − 0.6) = 125 vs. 116.

Step 8 / 9

```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
3      for t in range(T):
4          coop += delta**t * 50
5          if t < s:
6              dev += delta**t * 50
7          elif t == s:
8              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
▶      return round(coop, 2), round(dev, 2)
▶  print(compare(0.6, 2, 6))
12 print(compare(0.4, 2, 6))
```

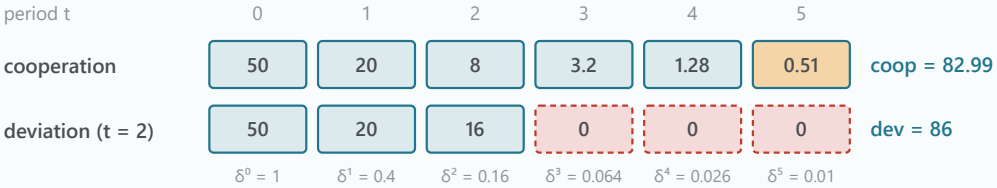
VARIABLES

`coop = 119.17` `dev = 116.0`

OUTPUT

```
(119.17, 116.0)
```

Code Trace: Does Deviating Pay Under a Trigger?



δ = 0.4: deviation 86 vs. 82.99 (83.3 at infinity): collusion collapses. Threshold δ = 1/2.

Step 9 / 9

```
1  def compare(delta, s, T):
2      coop, dev = 0.0, 0.0          # cooperate / deviate
3      for t in range(T):
4          coop += delta**t * 50
5          if t < s:
6              dev += delta**t * 50
7          elif t == s:
8              dev += delta**t * 100  # A undercuts
9          # t > s: trigger fired, profit 0
10     return round(coop, 2), round(dev, 2)
11 print(compare(0.6, 2, 6))
▶  print(compare(0.4, 2, 6))
```

VARIABLES

`coop = 82.99` `dev = 86.0`

OUTPUT

```
(119.17, 116.0)
(82.99, 86.0)
```

What Makes Collusion Easier or Harder

firms n	share 100/n	δ needed
2	50	0.50
3	33.3	0.67
4	25	0.75
5	20	0.80
10	10	0.90

$(100/n)/(1 - \delta) \geq 100 \iff \delta \geq 1 - 1/n$. With more firms, each one's cooperative share shrinks and deviating becomes more tempting.

Frequent price updates

If prices update every minute, the future is worth almost as much as today ($\delta \approx 1$); a deviator's gain lasts only a few minutes.

Transparent prices

If rival prices are visible instantly, a deviation is noticed at once and punishment is immediate.

Price-matching guarantees

"If a rival is cheaper, we refund the difference" wipes out the gain from undercutting and protects collusion.

Policy proposals

Test algorithms in simulated markets; open decision logs to audit.

Back to the Game: The AI Agent Tournament

Prisoner's Dilemma, Level 1

Payoffs (3,3), (0,5), (5,0), (1,1); totals over three matches:

Agent	Expected total
Always Defect	**544**
Best Response	535.5
Random	303.5
Always Cooperate	93

Matches of 60 rounds; Best Response cooperates in round 1 (no history),

then always defects (dominant): mutual cooperation is rare.

The environment picks the winner

In Matching Pennies, Best Response exploits the fixed agents and wins with about +118; only Random is unexploitable (expected 0 per match).

In practice: Tit-for-Tat won Axelrod's (1980) tournament: reciprocity and the threat of punishment sustained cooperation. These agents do not punish; the pricing agents of Calvano et al. (2020) remembered the last period, learned to punish and kept prices high.

No strategy is best everywhere: it depends on the game, the opponents, noise and memory.

Automated Mechanism Design

The classical route

- A person proposes a rule (Vickrey, VCG, Myerson).
- IC and optimality are proved by hand.
- Works for structured problems (one item); with many items and buyers the optimum is often unknown.

The automated route

- The mechanism solves an optimization problem: objective (revenue, welfare) + constraints IC (truth is best) and IR (participating does no harm).
- Linear programs over randomized mechanisms for finite types (Conitzer and Sandholm, 2002).
- Neural-network auctions: IC becomes a "regret" penalty, the extra utility a bidder could gain by lying (Dütting et al., 2019).

Approximate IC

A learned rule may keep a small regret.

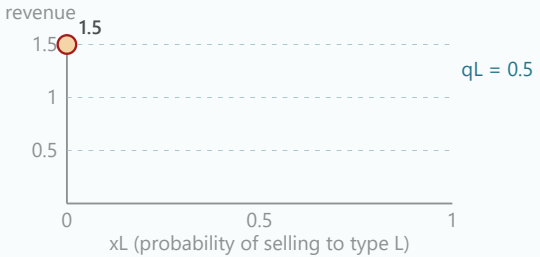
Generalization

If the distribution shifts, the rule may fail.

Explainability

"Why this price?" cannot be answered with network weights.

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 0, p = 0

H (value 3): x = 1, p = 3

H's utility: truthful 0, claiming L 0

One buyer, one item. Type L (value 1) with probability qL, type H (value 3). xL = 0: L never buys, H buys at 3. Revenue 0.5·3 = **1.5**. General rule: L's price is the most L will pay: pL = xL·1. If H acts like L, H gains 3·xL − pL; the highest price that keeps H truthful is pH = pL + 3·(1 − xL).

Step 1 / 8

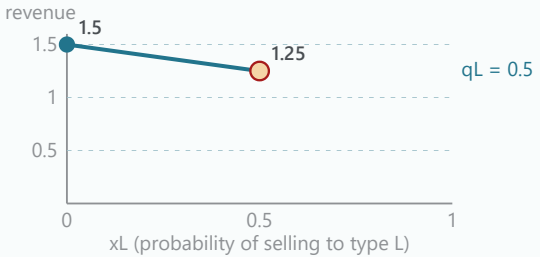
```
1 def revenue(xL, qL, vL=1, vH=3):
▶     pL = xL * vL                # L's IR binds
▶     pH = pL + vH * (1 - xL)     # H's IC binds
▶     return qL * pL + (1 - qL) * pH
▶ for qL in [0.5, 0.75]:
▶     G = [revenue(x, qL) for x in [0, 0.5, 1]]
7     print(qL, G)
```

VARIABLES

qL = 0.5 | xL = 0 | pL = 0 | pH = 3 | revenue = 1.5

OUTPUT

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 0.5, p = 0.5

H (value 3): x = 1, p = 2

H's utility: truthful 1, claiming L 1

xL = 0.5: selling to L with probability one half for an expected payment of 0.5 (price 1 when sold) gives H an option to lie (claiming L yields 3·0.5 − 0.5 = 1). H's price drops to 2. Revenue 0.25 + 1 = **1.25**.

Step 2 / 8

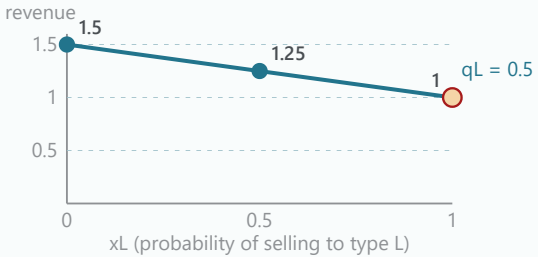
```
1 def revenue(xL, qL, vL=1, vH=3):
▶     pL = xL * vL                # L's IR binds
▶     pH = pL + vH * (1 - xL)     # H's IC binds
▶     return qL * pL + (1 - qL) * pH
▶ for qL in [0.5, 0.75]:
▶     G = [revenue(x, qL) for x in [0, 0.5, 1]]
7     print(qL, G)
```

VARIABLES

qL = 0.5 | xL = 0.5 | pL = 0.5 | pH = 2 | revenue = 1.25

OUTPUT

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 1, p = 1

H (value 3): x = 1, p = 1

H's utility: truthful 2, claiming L 2

xL = 1: everyone buys at 1; H pays the same price even if it lies. Revenue **1**.

Step 3 / 8

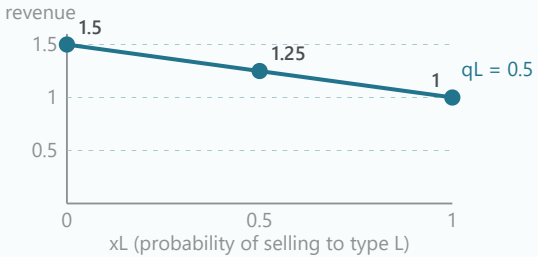
```
1 def revenue(xL, qL, vL=1, vH=3):
▶     pL = xL * vL                # L's IR binds
▶     pH = pL + vH * (1 - xL)     # H's IC binds
▶     return qL * pL + (1 - qL) * pH
▶ for qL in [0.5, 0.75]:
▶     G = [revenue(x, qL) for x in [0, 0.5, 1]]
7     print(qL, G)
```

VARIABLES

qL = 0.5 | xL = 1 | pL = 1 | pH = 1 | revenue = 1.0

OUTPUT

Code Trace: Automated Design for a Two-Type Buyer



mechanism

qL = 0.5: best is xL = 0, i.e. **sell only to H, at 3**.

Step 4 / 8

```
1 def revenue(xL, qL, vL=1, vH=3):
2     pL = xL * vL                  # L's IR binds
3     pH = pL + vH * (1 - xL)       # H's IC binds
4     return qL * pL + (1 - qL) * pH
5 for qL in [0.5, 0.75]:
6     G = [revenue(x, qL) for x in [0, 0.5, 1]]
▶     print(qL, G)
```

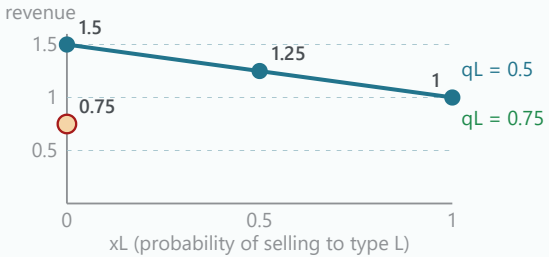
VARIABLES

qL = **0.5** G = **[1.5, 1.25, 1.0]**

OUTPUT

```
0.5 [1.5, 1.25, 1.0]
```

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 0, p = 0

H (value 3): x = 1, p = 3

H's utility: truthful 0, claiming L 0

qL = 0.75: type L is now the majority. Selling only to H earns little: 0.25·3 = **0.75**.

Step 5 / 8

```
1 def revenue(xL, qL, vL=1, vH=3):
    pL = xL * vL                # L's IR binds
    pH = pL + vH * (1 - xL)     # H's IC binds
    return qL * pL + (1 - qL) * pH
for qL in [0.5, 0.75]:
    G = [revenue(x, qL) for x in [0, 0.5, 1]]
7   print(qL, G)
```

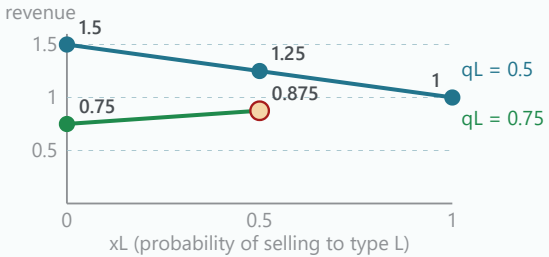
VARIABLES

qL = 0.75 | xL = 0 | pL = 0 | pH = 3 | revenue = 0.75

OUTPUT

```
0.5 [1.5, 1.25, 1.0]
```

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 0.5, p = 0.5

H (value 3): x = 1, p = 2

H's utility: truthful 1, claiming L 1

xL = 0.5: 0.75·0.5 + 0.25·2 = **0.875**.

Step 6 / 8

```
1 def revenue(xL, qL, vL=1, vH=3):
      pL = xL * vL                # L's IR binds
      pH = pL + vH * (1 - xL)     # H's IC binds
      return qL * pL + (1 - qL) * pH
  for qL in [0.5, 0.75]:
      G = [revenue(x, qL) for x in [0, 0.5, 1]]
7     print(qL, G)
```

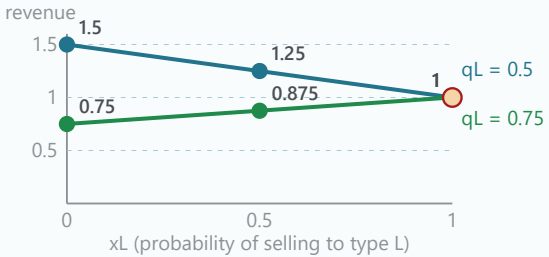
VARIABLES

qL = **0.75** xL = **0.5** pL = **0.5** pH = **2** revenue = **0.875**

OUTPUT

```
0.5 [1.5, 1.25, 1.0]
```

Code Trace: Automated Design for a Two-Type Buyer



mechanism

L (value 1): x = 1, p = 1

H (value 3): x = 1, p = 1

H's utility: truthful 2, claiming L 2

xL = 1: sell to everyone at 1: **1**.

Step 7 / 8

```
1 def revenue(xL, qL, vL=1, vH=3):
▶     pL = xL * vL                  # L's IR binds
▶     pH = pL + vH * (1 - xL)       # H's IC binds
▶     return qL * pL + (1 - qL) * pH
▶ for qL in [0.5, 0.75]:
▶     G = [revenue(x, qL) for x in [0, 0.5, 1]]
7     print(qL, G)
```

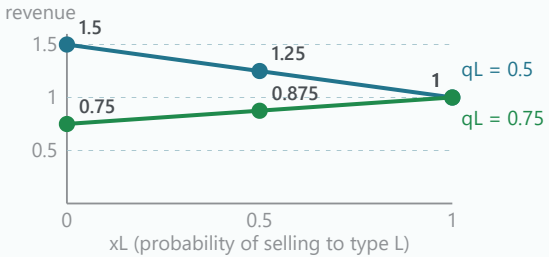
VARIABLES

qL = **0.75** xL = **1** pL = **1** pH = **1** revenue = **1.0**

OUTPUT

```
0.5 [1.5, 1.25, 1.0]
```

Code Trace: Automated Design for a Two-Type Buyer



mechanism

qL = 0.75: best: **sell to everyone at 1**. Revenue is linear in xL, so at least one optimum is at an endpoint (a vertex of the linear program; at qL = 2/3 every xL gives the same revenue): both solutions are posted prices.

Step 8 / 8

```
1 def revenue(xL, qL, vL=1, vH=3):
2     pL = xL * vL                  # L's IR binds
3     pH = pL + vH * (1 - xL)       # H's IC binds
4     return qL * pL + (1 - qL) * pH
5 for qL in [0.5, 0.75]:
6     G = [revenue(x, qL) for x in [0, 0.5, 1]]
▶     print(qL, G)
```

VARIABLES

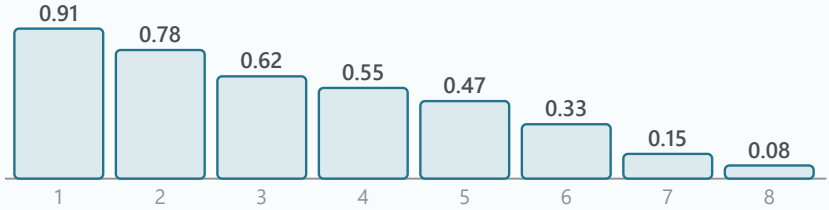
qL = **0.75**

G = **[0.75, 0.875, 1.0]**

OUTPUT

```
0.5 [1.5, 1.25, 1.0]
0.75 [0.75, 0.875, 1.0]
```

Code Trace: Learning a Reserve Price from Data



The 8 buyer values observed in the past are sorted in decreasing order.

Step 1 / 12

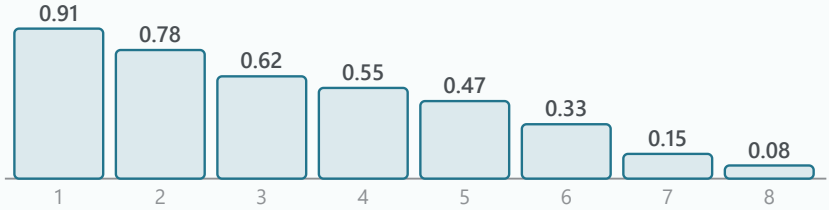
```
1  def best_price(samples):
▶    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
6      p = v[k - 1]          # k-th highest
7      g = p * k / n         # k buyers buy
8      if g > revenue:
9        price, revenue = p, g
10   return price, revenue
```

VARIABLES

```
samples = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
```

```
v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]
```

Code Trace: Learning a Reserve Price from Data



n = 8 samples. At price p, the samples with value at least p buy.

Step 2 / 12

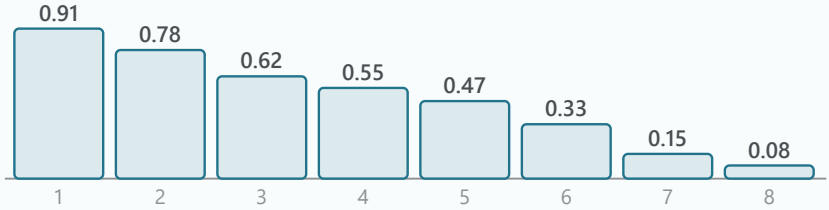
```
1 def best_price(samples):
2   v = sorted(samples, reverse=True)
▶   n = len(v)
4   price, revenue = 0, 0
5   for k in range(1, n + 1):
6     p = v[k - 1]        # k-th highest
7     g = p * k / n       # k buyers buy
8     if g > revenue:
9       price, revenue = p, g
10  return price, revenue
```

VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8

Code Trace: Learning a Reserve Price from Data



Best price so far and its empirical revenue.

Step 3 / 12

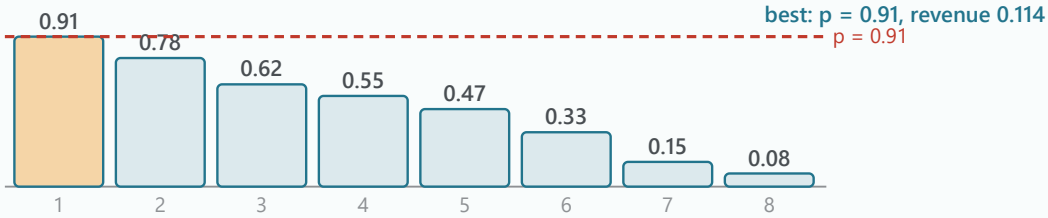
```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
▶    price, revenue = 0, 0
5    for k in range(1, n + 1):
6      p = v[k - 1]        # k-th highest
7      g = p * k / n       # k buyers buy
8      if g > revenue:
9        price, revenue = p, g
10   return price, revenue
```

VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 price = 0 revenue = 0

Code Trace: Learning a Reserve Price from Data



k = 1: p = 0.91 → 1 buyer buys, g = 0.91·1/8 = **0.114** → new best.

Step 4 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
▶        price, revenue = p, g
10   return price, revenue
```

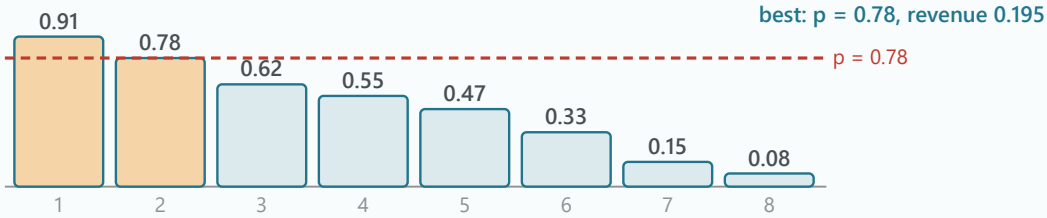
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 1 | p = 0.91 | g = 0.114

price = 0.91 | revenue = 0.114

Code Trace: Learning a Reserve Price from Data



k = 2: p = 0.78 → 2 buyers buy, g = 0.78·2/8 = **0.195** → new best.

Step 5 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
▶        price, revenue = p, g
10   return price, revenue
```

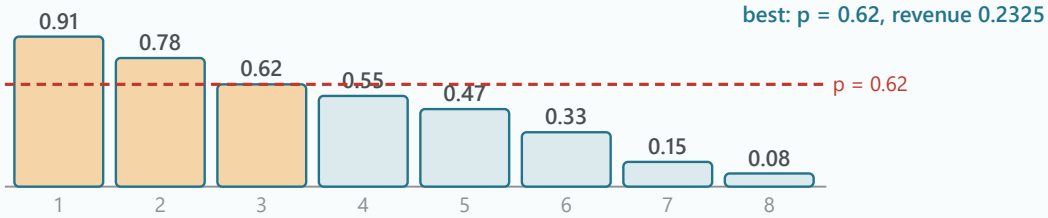
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 2 | p = 0.78 | g = 0.195

price = 0.78 | revenue = 0.195

Code Trace: Learning a Reserve Price from Data



k = 3: p = 0.62 → 3 buyers buy, g = 0.62·3/8 = **0.2325** → new best.

Step 6 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
▶        price, revenue = p, g
10   return price, revenue
```

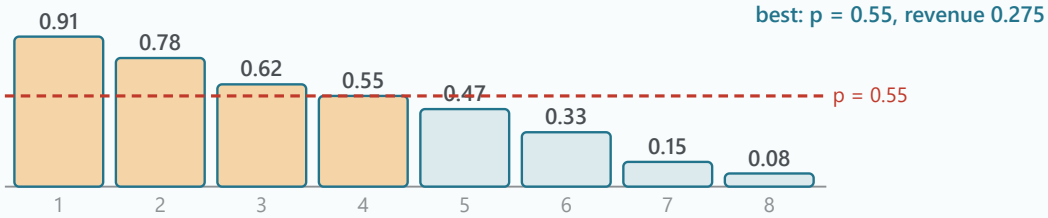
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 3 | p = 0.62 | g = 0.2325

price = 0.62 | revenue = 0.2325

Code Trace: Learning a Reserve Price from Data



k = 4: p = 0.55 → 4 buyers buy, g = 0.55·4/8 = **0.275** → new best.

Step 7 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
▶        price, revenue = p, g
10   return price, revenue
```

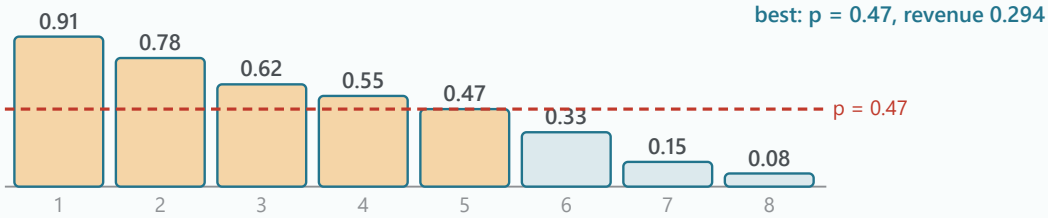
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 4 | p = 0.55 | g = 0.275

price = 0.55 | revenue = 0.275

Code Trace: Learning a Reserve Price from Data



k = 5: p = 0.47 → 5 buyers buy, g = 0.47·5/8 = **0.294** → new best.

Step 8 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
▶        price, revenue = p, g
10   return price, revenue
```

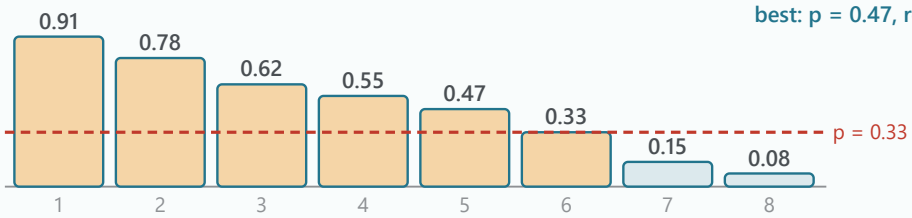
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 5 | p = 0.47 | g = 0.294

price = 0.47 | revenue = 0.294

Code Trace: Learning a Reserve Price from Data



k = 6: p = 0.33 → 6 buyers buy, g = 0.33·6/8 = **0.2475** ≤ 0.294, no change.

Step 9 / 12

```
1 def best_price(samples):
2   v = sorted(samples, reverse=True)
3   n = len(v)
4   price, revenue = 0, 0
5   for k in range(1, n + 1):
▶     p = v[k - 1]          # k-th highest
▶     g = p * k / n         # k buyers buy
▶     if g > revenue:
9       price, revenue = p, g
10  return price, revenue
```

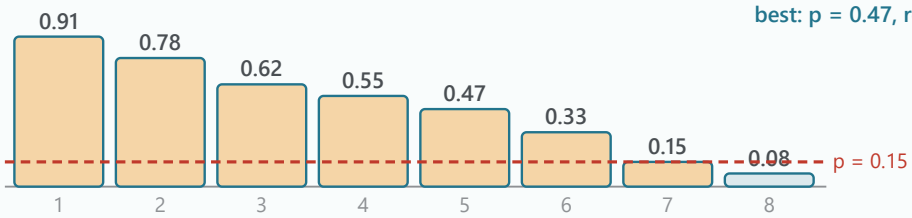
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 6 | p = 0.33 | g = 0.2475

price = 0.47 | revenue = 0.294

Code Trace: Learning a Reserve Price from Data



k = 7: p = 0.15 → 7 buyers buy, g = 0.15·7/8 = **0.131** ≤ 0.294, no change.

Step 10 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
9        price, revenue = p, g
10   return price, revenue
```

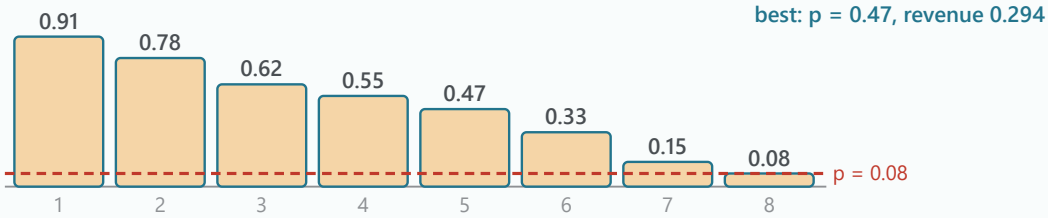
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 7 | p = 0.15 | g = 0.131

price = 0.47 | revenue = 0.294

Code Trace: Learning a Reserve Price from Data



k = 8: p = 0.08 → 8 buyers buy, g = 0.08·8/8 = **0.08** ≤ 0.294, no change.

Step 11 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
▶      p = v[k - 1]          # k-th highest
▶      g = p * k / n         # k buyers buy
▶      if g > revenue:
9        price, revenue = p, g
10   return price, revenue
```

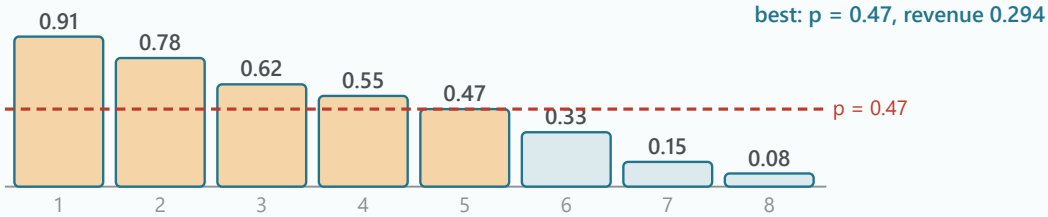
VARIABLES

v = [0.91, 0.78, 0.62, 0.55, 0.47, 0.33, 0.15, 0.08]

n = 8 | k = 8 | p = 0.08 | g = 0.08

price = 0.47 | revenue = 0.294

Code Trace: Learning a Reserve Price from Data



Learned price **0.47**, empirical revenue 0.294. If the values actually come from U[0, 1] (every value between 0 and 1 equally likely), the true revenue is 0.47·0.53 ≈ **0.249** (the best is 0.25).

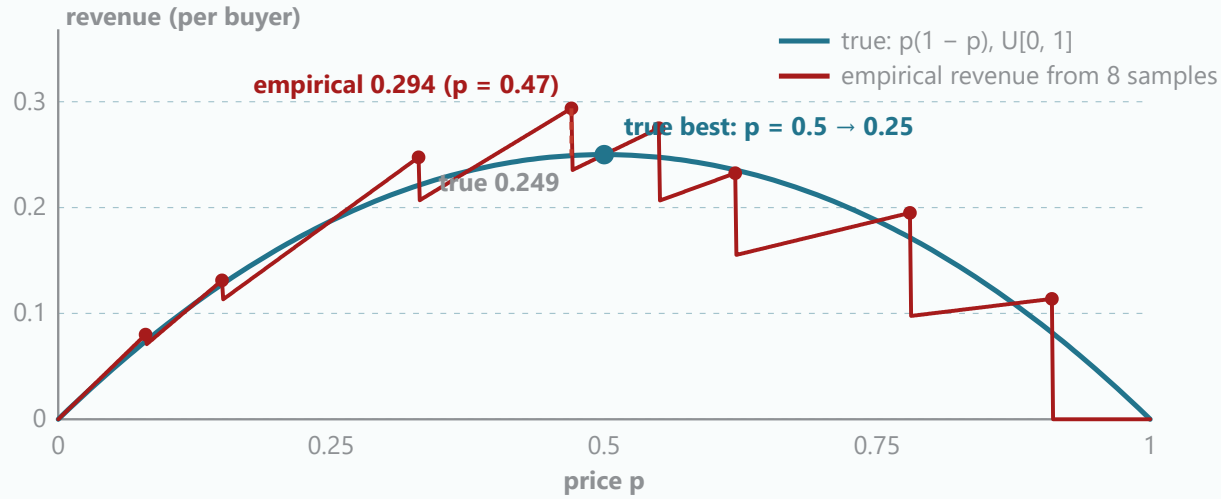
Step 12 / 12

```
1  def best_price(samples):
2    v = sorted(samples, reverse=True)
3    n = len(v)
4    price, revenue = 0, 0
5    for k in range(1, n + 1):
6      p = v[k - 1]        # k-th highest
7      g = p * k / n       # k buyers buy
8      if g > revenue:
9        price, revenue = p, g
▶    return price, revenue
```

VARIABLES

`price = 0.47` `revenue = 0.294`

How Far Does the Empirical Revenue Curve Stray from the Truth?



- The empirical curve $p \cdot \#\{v_i \geq p\}/n$ is piecewise linear: it rises between sample values and drops at them; its maximum is always at a sample value (here 0.47).
- The chosen price's performance on the sample (0.294) overstates its true performance (0.249): because the same data is used to choose and to measure, there is an **optimistic bias**. The gap to the true best of 0.25 is small, but it grows as samples get scarce.

How Many Samples Are Enough? Sample Complexity

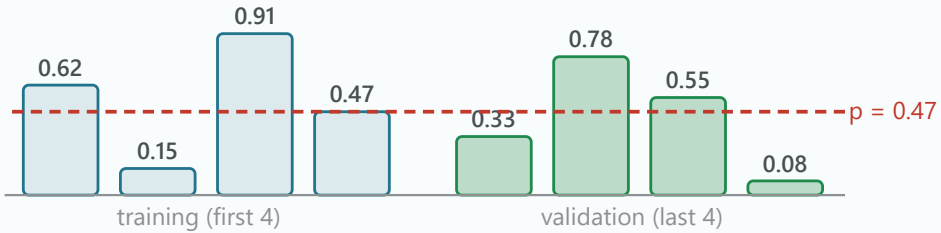
samples n	empirical revenue	true revenue	0.25 − true
10	0.311	0.224	0.0262
100	0.263	0.245	0.005
1,000	0.253	0.249	0.0011
10,000	0.25	0.25	0.0002

Values U[0, 1]; averages of hundreds of simulated runs. Best revenue 0.25 (p = 0.5).

- **Overfitting**: with few samples, empirical revenue overstates the truth: the price "memorized" the samples.
- Sample complexity: how many samples give (with high probability) a near-optimal price?
- Without extreme values, the samples needed grow polynomially in bidders and accuracy (Cole and Roughgarden, 2014); heavy tails may allow no such guarantee.

Lesson: with scarce data, a learned mechanism overstates its success. Separate training and evaluation data, as in machine learning.

Separate Data for Choosing and Measuring



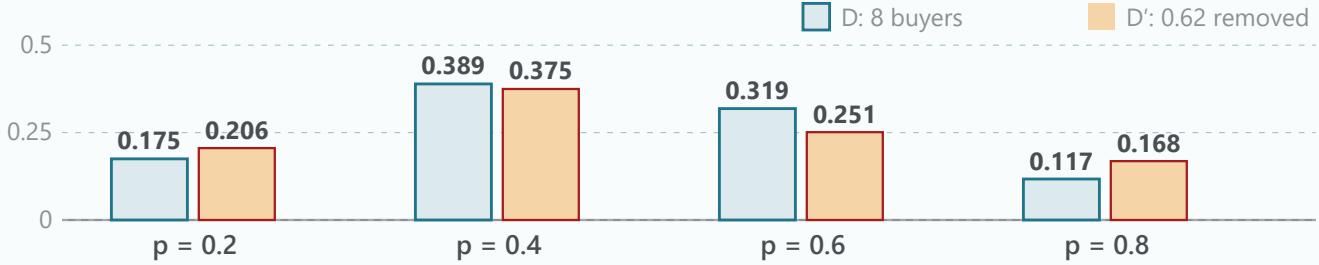
Measurement (p = 0.47)	revenue
on training (where it was chosen): 0.47·3/4	0.353
on validation: 0.78 and 0.55 buy → 0.47·2/4	0.235
true, U[0, 1]: 0.47·0.53	0.249

- The training measurement is optimistic (0.353); the validation measurement is unbiased and close to the truth.
- Swap the roles: training = last 4 → p = 0.55, revenue 0.275 on the first 4. **Cross-validation** averages the two estimates: (0.235 + 0.275)/2 = 0.255.
- The cost: with only 4 samples to choose from, the price is noisier.

A risk specific to mechanisms: if buyers know their past bids set future prices, they may bid low today. Data can be strategic too.

Differential Privacy: One Person's Data Should Barely Matter

- **Differential privacy**: adding or removing a single person's data changes the probability of any outcome only slightly. Here ε measures privacy: a smaller ε means more privacy (a change by a factor of at most e^{ε}).
- **The exponential mechanism** (McSherry and Talwar, 2007): instead of picking the best price outright, it runs a lottery; good prices get exponentially higher probability, bad prices a small but nonzero probability.
- **The game theory link**: since one person's report barely affects the outcome, lying gains little too (at most $e^{\varepsilon} - 1$ if utilities are in [0, 1]) → *approximate* incentive compatibility, with privacy on top.



Reserve price example, ε = 2: removing one buyer changes the probabilities by a factor of at most 1.44 (the bound is $e^2 \approx 7.39$).

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



Data: 8 buyers' values (the same as in the price-learning example). Candidate reserve prices 0.2 – 0.8. Δ: the most a price's score can change when one person is removed; prices are below 1, so Δ = 1.

Step 1 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
3      R = [p * sum(x >= p for x in v) for p in prices]
4      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5      W = sum(w)
6      return [x / W for x in w]
▶ v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
▶ P = [0.2, 0.4, 0.6, 0.8]
9  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

v = **8 buyers**

P = **[0.2, 0.4, 0.6, 0.8]**

OUTPUT

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



R = p·(number of buyers): 0.2·6 = 1.20, 0.4·5 = 2.00, 0.6·3 = 1.80, 0.8·1 = 0.80.

Step 2 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
▶      R = [p * sum(x >= p for x in v) for p in prices]
4      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5      W = sum(w)
6      return [x / W for x in w]
7  v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8  P = [0.2, 0.4, 0.6, 0.8]
▶  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

`n = 8`

`R = [1.20, 2.00, 1.80, 0.80]`

OUTPUT

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



w = exp(ε·R/2) = exp(R): 3.32, 7.39, 6.05, 2.23. The gaps grow exponentially, but a bad price still has a chance.

Step 3 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
3      R = [p * sum(x >= p for x in v) for p in prices]
▶      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5      W = sum(w)
6      return [x / W for x in w]
7  v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8  P = [0.2, 0.4, 0.6, 0.8]
9  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

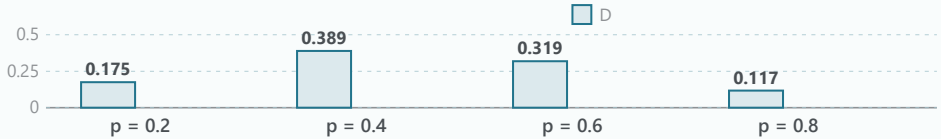
VARIABLES

R = [1.20, 2.00, 1.80, 0.80]

w = [3.32, 7.39, 6.05, 2.23]

OUTPUT

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



W = 18.98; probabilities q = w/W = 0.175, 0.389, 0.319, 0.117.

Step 4 / 9

```
1 import math
2 def exp_mech(v, prices, eps):
3     R = [p * sum(x >= p for x in v) for p in prices]
4     w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
▶     W = sum(w)
▶     return [x / W for x in w]
7 v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8 P = [0.2, 0.4, 0.6, 0.8]
9 q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

W = 18.984

q = [0.17, 0.39, 0.32, 0.12]

OUTPUT

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



The output is printed. p = 0.4 is the most likely (38.9%), but not the only one.

Step 5 / 9

```
1 import math
2 def exp_mech(v, prices, eps):
3     R = [p * sum(x >= p for x in v) for p in prices]
4     w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5     W = sum(w)
6     return [x / W for x in w]
7 v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8 P = [0.2, 0.4, 0.6, 0.8]
9 q = exp_mech(v, P, 2)
▶ print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

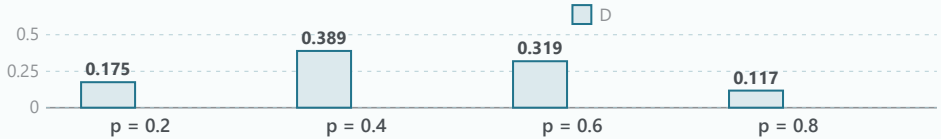
VARIABLES

q = [0.17, 0.39, 0.32, 0.12]

OUTPUT

```
[0.175, 0.389, 0.319, 0.117]
```

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



R = p·(number of buyers): 0.2·5 = 1.00, 0.4·4 = 1.60, 0.6·2 = 1.20, 0.8·1 = 0.80. Without 0.62, the revenue of p = 0.6 drops by 0.6.

Step 6 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
▶      R = [p * sum(x >= p for x in v) for p in prices]
4      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5      W = sum(w)
6      return [x / W for x in w]
7  v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8  P = [0.2, 0.4, 0.6, 0.8]
9  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
▶  q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

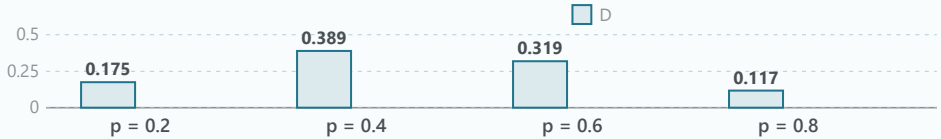
`n = 7`

`R = [1.00, 1.60, 1.20, 0.80]`

OUTPUT

```
[0.175, 0.389, 0.319, 0.117]
```

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



w = exp(ε·R/2) = exp(R): 2.72, 4.95, 3.32, 2.23. The gaps grow exponentially, but a bad price still has a chance.

Step 7 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
3      R = [p * sum(x >= p for x in v) for p in prices]
▶      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
5      W = sum(w)
6      return [x / W for x in w]
7  v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8  P = [0.2, 0.4, 0.6, 0.8]
9  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

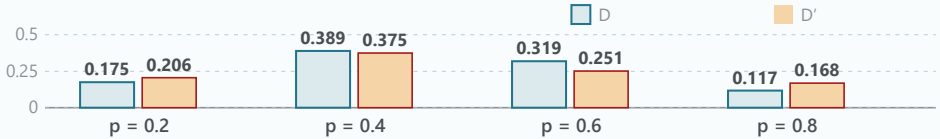
R = [1.00, 1.60, 1.20, 0.80]

w = [2.72, 4.95, 3.32, 2.23]

OUTPUT

```
[0.175, 0.389, 0.319, 0.117]
```

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



W = 13.22; probabilities q = w/W = 0.206, 0.375, 0.251, 0.168.

Step 8 / 9

```
1  import math
2  def exp_mech(v, prices, eps):
3      R = [p * sum(x >= p for x in v) for p in prices]
4      w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
▶      W = sum(w)
▶      return [x / W for x in w]
7  v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
8  P = [0.2, 0.4, 0.6, 0.8]
9  q = exp_mech(v, P, 2)
10 print([round(x, 3) for x in q])
11 q = exp_mech(v[1:], P, 2)       # 0.62 removed
12 print([round(x, 3) for x in q])
```

VARIABLES

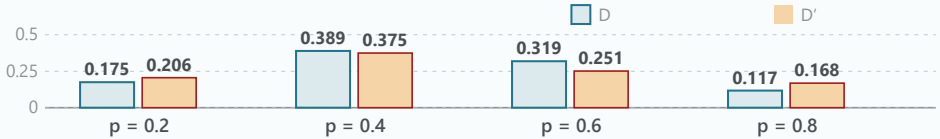
W = 13.217

q = [0.21, 0.37, 0.25, 0.17]

OUTPUT

```
[0.175, 0.389, 0.319, 0.117]
```

Further Reading · Code Trace: Choosing a Price with the Exponential Mechanism



The two distributions side by side: the largest ratio is $1.44 \le e^2 = 7.39$. One person's data has a bounded effect on the outcome: ε-differential privacy.

Step 9 / 9

```python
import math
def exp_mech(v, prices, eps):
    R = [p * sum(x >= p for x in v) for p in prices]
    w = [math.exp(eps * r / 2) for r in R]   # Δ = 1
    W = sum(w)
    return [x / W for x in w]
v = [0.62, 0.15, 0.91, 0.47, 0.33, 0.78, 0.55, 0.08]
P = [0.2, 0.4, 0.6, 0.8]
q = exp_mech(v, P, 2)
print([round(x, 3) for x in q])
q = exp_mech(v[1:], P, 2)       # 0.62 removed
print([round(x, 3) for x in q])
```

VARIABLES

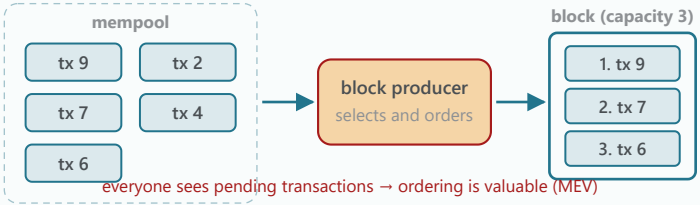
`q = [0.21, 0.37, 0.25, 0.17]`

OUTPUT

```
[0.175, 0.389, 0.319, 0.117]
[0.206, 0.375, 0.251, 0.168]
```

Blockchains and Smart Contracts

- **Smart contract**: a program whose code is public and runs automatically on-chain → a self-enforcing mechanism.
- Mechanism design's assumption that "the designer is bound by the rules" (commitment) is enforced by code; but pending transactions are visible to everyone too.
- The block producer decides which transactions enter the block and in what **order**: a scarce resource → an auction.



- **Transaction fee auction**: Scarce block space is allocated to bidders.
- **Decentralized exchange**: An automated market maker (AMM): the price is set by a formula.
- **On-chain governance**: Token-weighted voting; the problems of voting rules apply here too.

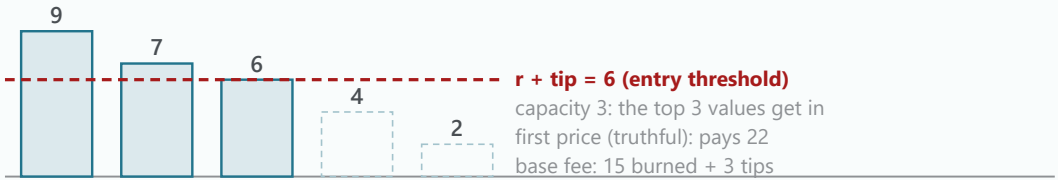
Transaction Fee Auctions

First price (the original design)

- Each transaction bids a fee; the producer takes the highest bids and everyone pays their own bid.
- Values 9, 7, 6, 4, 2 and capacity 3: truthful bids pay 9 + 7 + 6 = 22.
- But truthful bidding is not optimal: everyone shades, and "how much should I bid?" burdens the user.

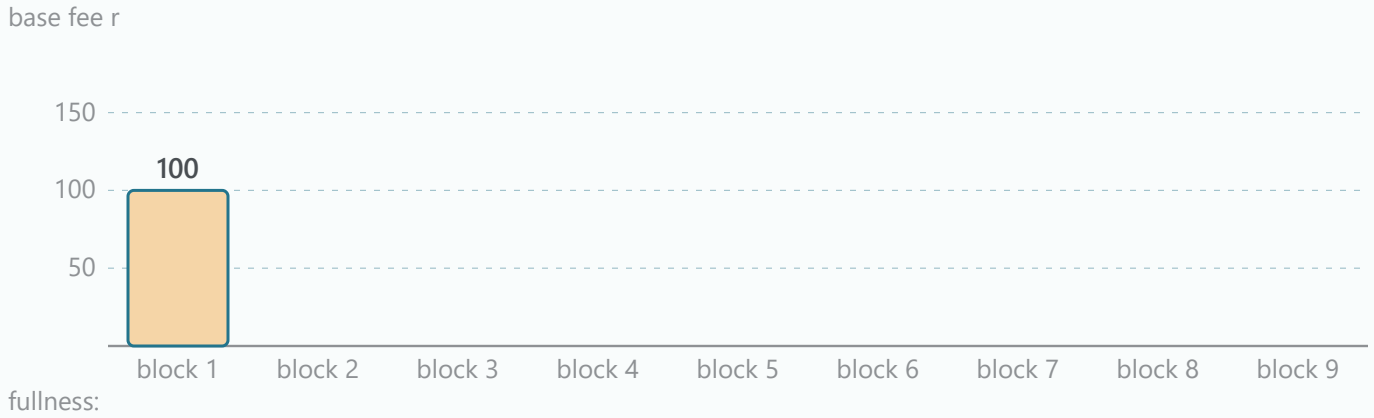
Base fee + tip (EIP-1559 style)

- The protocol sets a base fee r (here 5); r is **burned**, i.e. paid to no one and removed from circulation. Only the tip (1) goes to the producer.
- Transactions with value ≥ r + tip get in: 3 transactions, 15 burned, 3 in tips.
- If demand does not exceed capacity, the best strategy is simple: pay "r + a small tip"; no price guessing needed.



Design goals (Roughgarden, 2021): simple bidding for users, honest behavior for producers, resistance to collusion. A rule that fully achieves all of them is hard to find.

Step by Step: The Base Fee Tracks Demand

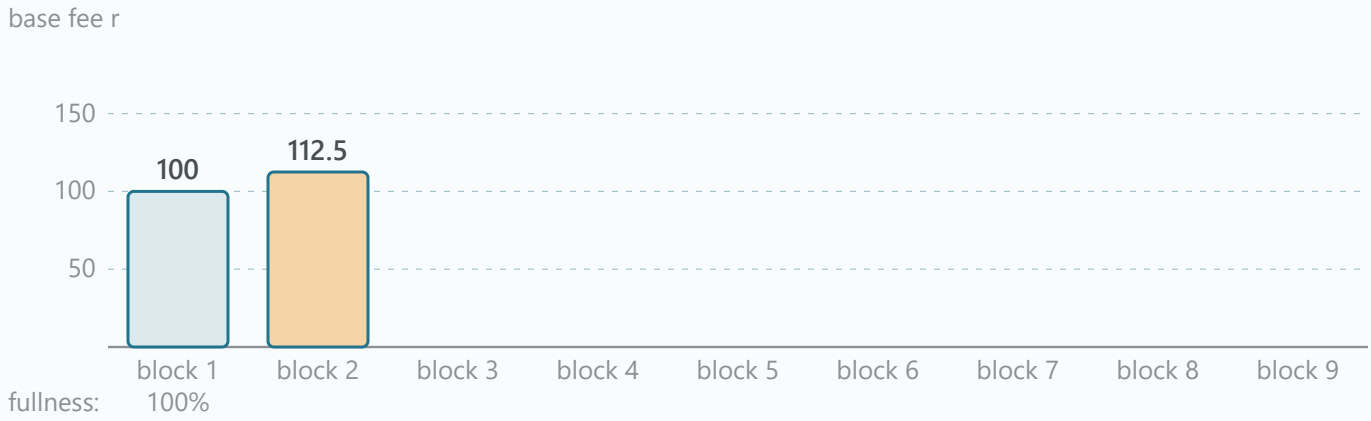


The general principle: the fee rises when usage is above target and falls when below. With EIP-1559's launch parameters: $r_{t+1} = r_t \cdot (1 + \frac{1}{8} \cdot (g_t - g_{target})/g_{target})$, with g_{target} half the block capacity. Initially r = 100.

Step 1 / 9

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

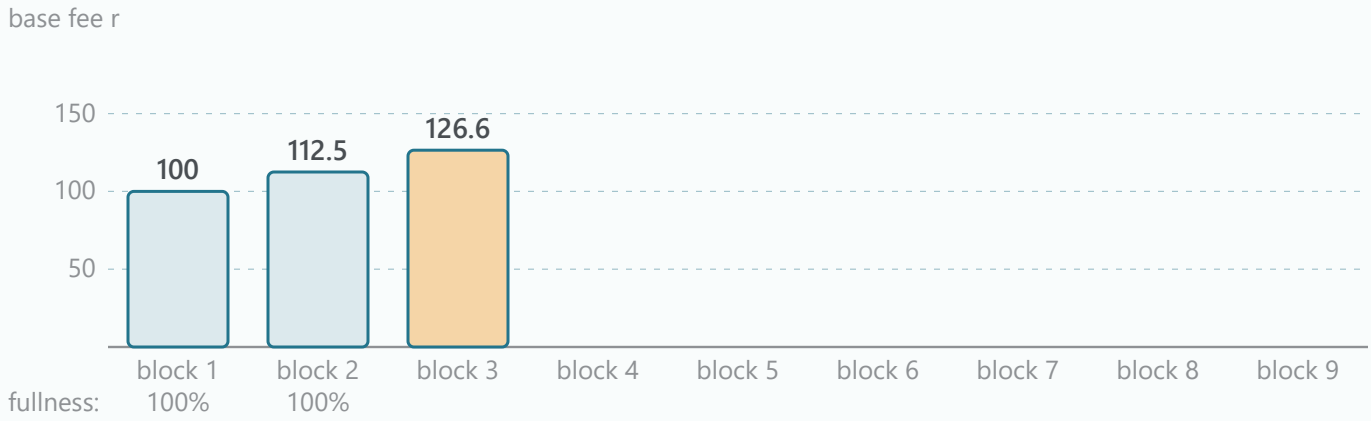
Step by Step: The Base Fee Tracks Demand



Block 1 is 100% full → change +12.5% → r = **112.5**.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

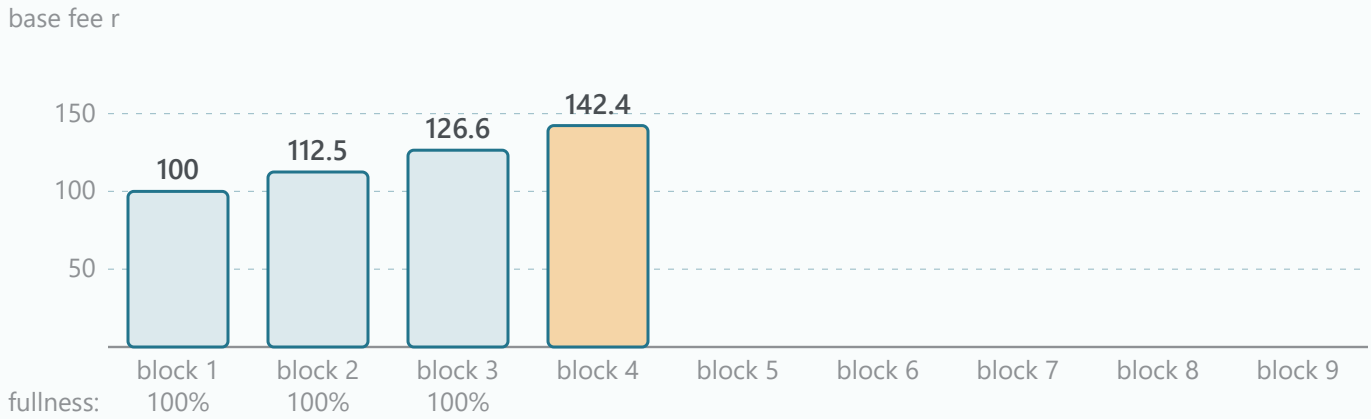
Step by Step: The Base Fee Tracks Demand



Block 2 is 100% full → change +12.5% → r = **126.6**.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

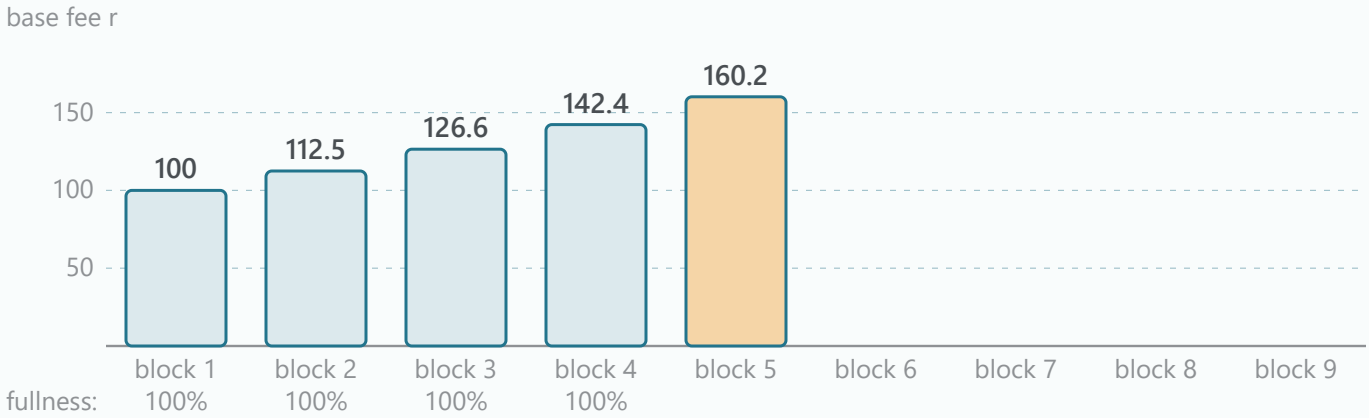
Step by Step: The Base Fee Tracks Demand



Block 3 is 100% full → change +12.5% → r = **142.4**.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

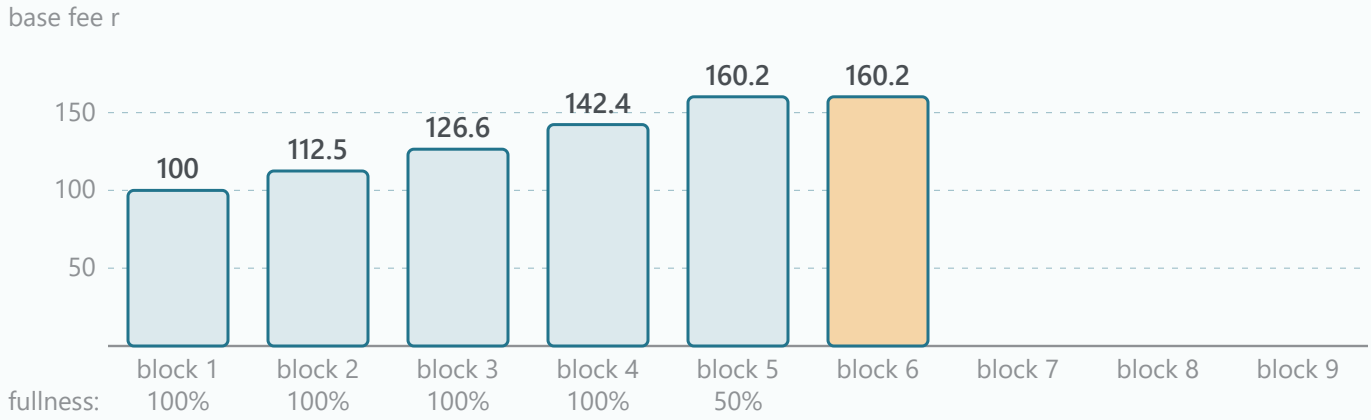
Step by Step: The Base Fee Tracks Demand



Block 4 is 100% full → change +12.5% → r = **160.2**. A demand spike: the fee rises by at most 12.5% per block.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

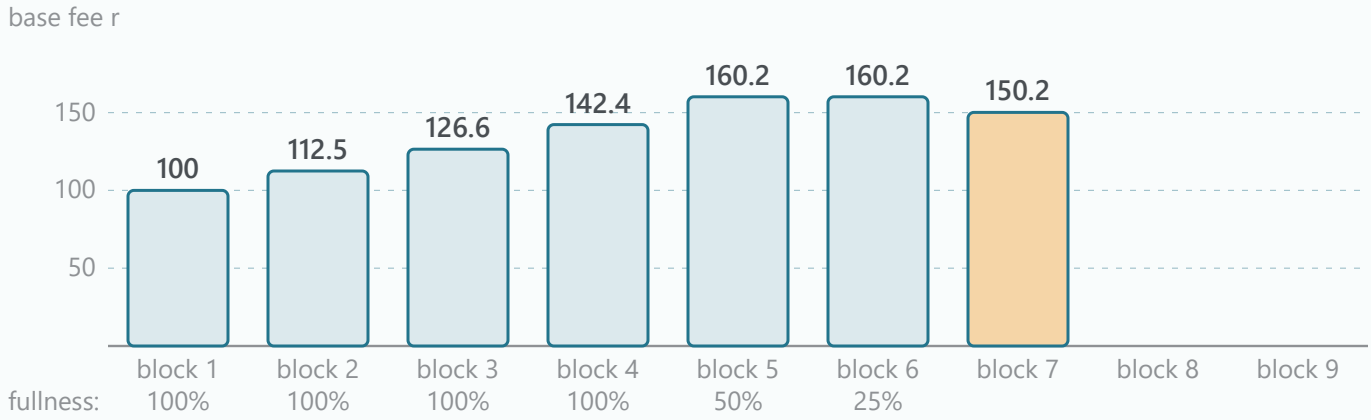
Step by Step: The Base Fee Tracks Demand



Block 5 is 50% full → change ±0% → r = **160.2**. Exactly on target, the fee stays put.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

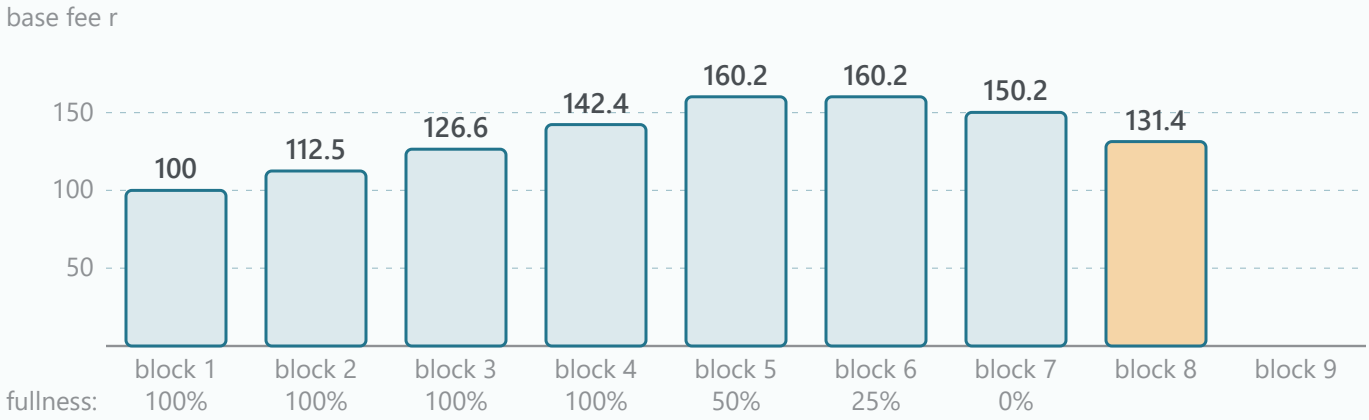
Step by Step: The Base Fee Tracks Demand



Block 6 is 25% full → change −6.25% → r = **150.2**.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

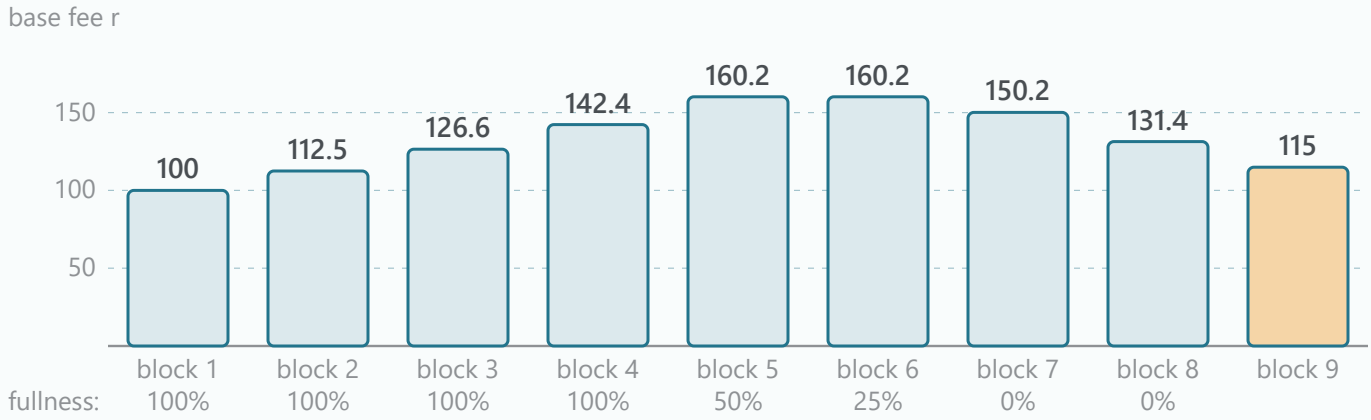
Step by Step: The Base Fee Tracks Demand



Block 7 is 0% full → change −12.5% → r = **131.4**.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

Step by Step: The Base Fee Tracks Demand



Block 8 is 0% full → change −12.5% → r = **115**. Empty blocks lower the fee; when demand falls, the price comes back down.

The base fee works like a "posted price" that tracks demand; when blocks are full it reverts to a tip-based auction.

Why Burn? Properties of a Good Fee Mechanism

User incentive compatibility

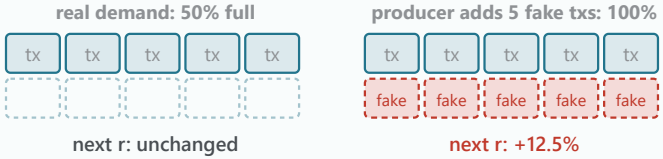
If demand does not exceed capacity, base fee + a small tip is the best bid; the user need not think "how much should I shade?".

Producer incentive compatibility

The producer earns the most by following the rule: adding fake transactions or withholding transactions does not pay.

Collusion resistance

The producer and users cannot raise their joint gain through off-chain agreements.



- If the base fee went to the producer, fake transactions would be **free** (the producer pays itself) and it could inflate the fee at will.
- With burning, every fake transaction is a real loss of r: 5 fake transactions burn 5r.
- Not free, but not unlimited either: in a full block the tip race (first price) comes back.

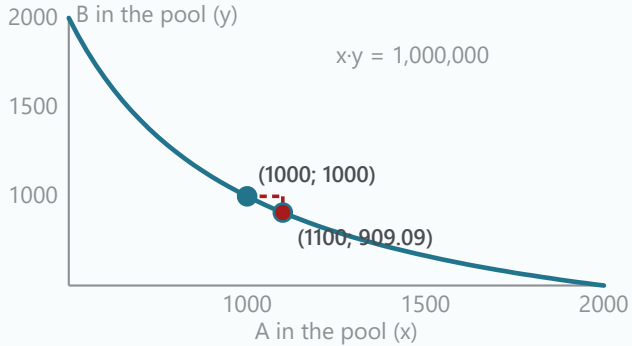
These three properties were formalized by Roughgarden (2021); when block space is scarce, the three cannot all hold together (Chung and Shi, 2023, with the side-contract definition; Chung, Roughgarden and Shi, 2024, with Roughgarden's definition).

Automated Market Maker: The Constant-Product Rule

- The pool holds x A and y B; every trade keeps **x·y = k**. The counterparty is a formula.
- Selling Δx A returns Δy B: $(x + \Delta x)\cdot(y - \Delta y) = x\cdot y \Rightarrow \Delta y = y\cdot\Delta x / (x + \Delta x)$.
- Slippage: B lost versus the starting price (9.90 B, not 10 B, for 10 A → 1%).

Δx (A)	B received	slippage
10	9.90	1.0%
100	90.91	9.1%
250	200.00	20.0%
500	333.33	33.3%

Pool (1000, 1000), price 1 A = 1 B. With a 0.3% fee, 100 A → 90.66 B; fees go to liquidity providers.



- Large trades move the price against themselves (**price impact**): after 100 A, 1 B = 1.21 A.
- Arbitrageurs close gaps with outside markets.
- Price impact is predictable, so front-running pays.

Step by Step: MEV and the Sandwich Attack

Order	Transaction	Pool (A, B)	Outcome
0	Start (x·y = 1,000,000)	(1000; 1000)	

A constant-product pool: selling A depletes B and the price slips. Alone, the user would get 90.91 B for 100 A.

The user's transaction is visible to everyone before it enters a block (the mempool).

Step by Step: MEV and the Sandwich Attack

Order	Transaction	Pool (A, B)	Outcome
0	Start (x·y = 1,000,000)	(1000; 1000)	
1	Attacker sells 50 A (front-run)	(1050; 952.38)	gets 47.62 B

Whoever controls the order puts their own transaction **in front of** the user's: buys B while it is cheap and pushes the price up.

Front-running.

Step by Step: MEV and the Sandwich Attack

Order	Transaction	Pool (A, B)	Outcome
0	Start (x·y = 1,000,000)	(1000; 1000)	
1	Attacker sells 50 A (front-run)	(1050; 952.38)	gets 47.62 B
2	User sells 100 A	(1150; 869.57)	gets 82.82 B (90.91 alone)

The user now trades at a worse price: 82.82 B → **8.09 B lost**.

The user's trade pushes the price up further.

Step by Step: MEV and the Sandwich Attack

Order	Transaction	Pool (A, B)	Outcome
0	Start (x·y = 1,000,000)	(1000; 1000)	
1	Attacker sells 50 A (front-run)	(1050; 952.38)	gets 47.62 B
2	User sells 100 A	(1150; 869.57)	gets 82.82 B (90.91 alone)
3	Attacker sells the B back (back-run)	(1090.29; 917.18)	gets 59.71 A

The attacker sells the B back at the high price: 59.71 A → **profit 9.71 A** (before fees).

Back-running: the "sandwich" is complete.

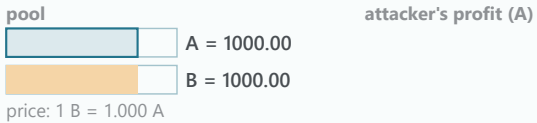
Step by Step: MEV and the Sandwich Attack

Order	Transaction	Pool (A, B)	Outcome
0	Start (x·y = 1,000,000)	(1000; 1000)	
1	Attacker sells 50 A (front-run)	(1050; 952.38)	gets 47.62 B
2	User sells 100 A	(1150; 869.57)	gets 82.82 B (90.91 alone)
3	Attacker sells the B back (back-run)	(1090.29; 917.18)	gets 59.71 A

MEV (maximal extractable value): the value that can be extracted from the power to order, insert and drop transactions. Mitigation ideas: batch auctions (one price), encrypted mempools, auctioning the right to order.

The mechanism design question: who holds the ordering power, and who pays for it?

Code Trace: The Sandwich Attack and the Slippage Limit



The user will sell 100 A and wants at least 82 B (slippage tolerance). The attacker tries three front-run sizes a.

Step 1 / 12

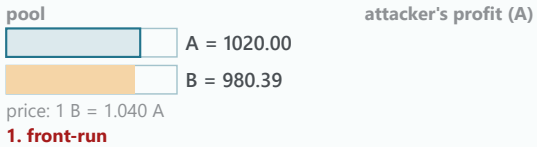
```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
▶ for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

VARIABLES

a = 20

OUTPUT

Code Trace: The Sandwich Attack and the Slippage Limit



a = 20: the attacker first sells 20 A and gets 19.61 B. B fell → the price rose.

Step 2 / 12

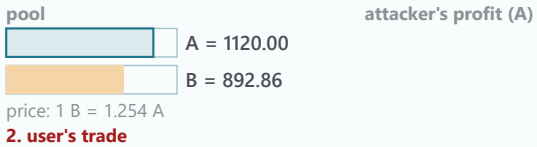
```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
▶      b1 = swap(x, y, a)           # front-run: sell a A
▶      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

VARIABLES

a = 20 | b1 = 19.61 | x = 1020.0 | y = 980.39

OUTPUT

Code Trace: The Sandwich Attack and the Slippage Limit



The user gets 87.54 B (≥ 82, the trade goes through; alone it would get 90.91).

Step 3 / 12

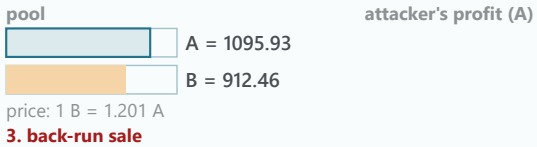
```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
▶      b2 = swap(x, y, u)           # user: sell u A
▶      if b2 < min_out:
7          return None              # tx reverted
▶      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

VARIABLES

a = 20 | b2 = 87.54 | x = 1120.0 | y = 892.86

OUTPUT

Code Trace: The Sandwich Attack and the Slippage Limit



The attacker sells the 19.61 B back for 24.07 A: profit = 24.07 − 20 = **4.07 A**.

Step 4 / 12

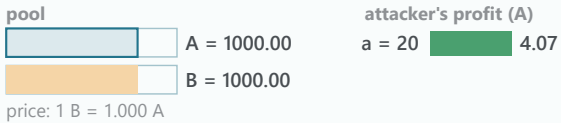
```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
▶      a2 = swap(y, x, b1)          # back-run: sell b1 B
▶      return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

VARIABLES

a = **20** a2 = **24.07** profit = **4.07**

OUTPUT

Code Trace: The Sandwich Attack and the Slippage Limit



The result is printed.

Step 5 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
▶ for a in [20, 50, 80]:
▶     print(a, sandwich(1000, 1000, a, 100, 82))
```

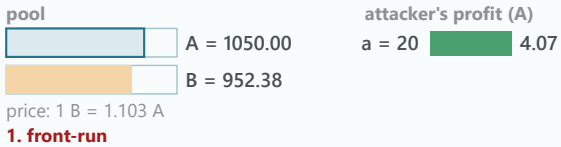
VARIABLES

a = 20 result = 4.07

OUTPUT

```
20 4.07
```

Code Trace: The Sandwich Attack and the Slippage Limit



a = 50: the attacker first sells 50 A and gets 47.62 B. B fell → the price rose.

Step 6 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
▶      b1 = swap(x, y, a)           # front-run: sell a A
▶      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

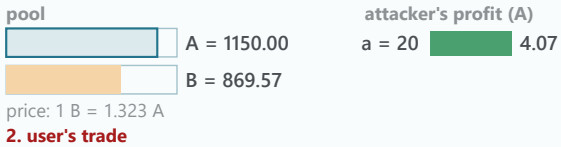
VARIABLES

a = 50 | b1 = 47.62 | x = 1050.0 | y = 952.38

OUTPUT

```
20 4.07
```

Code Trace: The Sandwich Attack and the Slippage Limit



The user gets 82.82 B (≥ 82, the trade goes through; alone it would get 90.91).

Step 7 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
▶      b2 = swap(x, y, u)           # user: sell u A
▶      if b2 < min_out:
7          return None              # tx reverted
▶      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

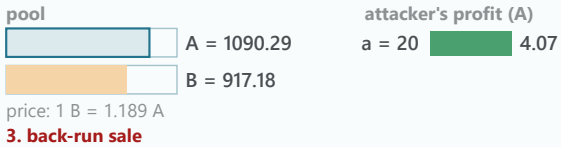
VARIABLES

a = 50 | b2 = 82.82 | x = 1150.0 | y = 869.57

OUTPUT

```
20 4.07
```

Code Trace: The Sandwich Attack and the Slippage Limit



The attacker sells the 47.62 B back for 59.71 A: profit = 59.71 − 50 = **9.71 A**.

Step 8 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
▶      a2 = swap(y, x, b1)          # back-run: sell b1 B
▶      return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

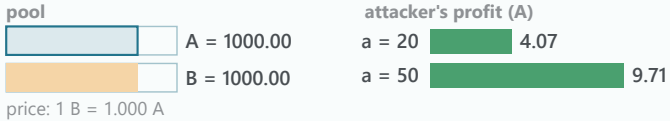
VARIABLES

a = **50** a2 = **59.71** profit = **9.71**

OUTPUT

```
20 4.07
```

Code Trace: The Sandwich Attack and the Slippage Limit



The result is printed.

Step 9 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
▶ for a in [20, 50, 80]:
▶     print(a, sandwich(1000, 1000, a, 100, 82))
```

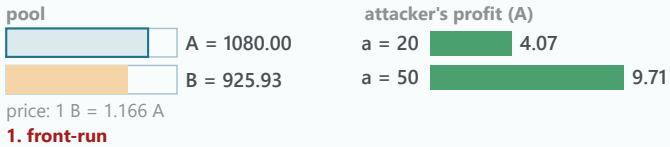
VARIABLES

`a = 50` `result = 9.71`

OUTPUT

```
20 4.07
50 9.71
```

Code Trace: The Sandwich Attack and the Slippage Limit



a = 80: the attacker first sells 80 A and gets 74.07 B. B fell → the price rose.

Step 10 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
▶      b1 = swap(x, y, a)           # front-run: sell a A
▶      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

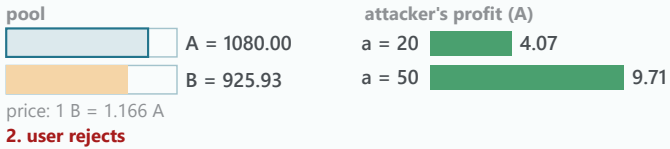
VARIABLES

a = 80 | b1 = 74.07 | x = 1080.0 | y = 925.93

OUTPUT

```
20 4.07
50 9.71
```

Code Trace: The Sandwich Attack and the Slippage Limit



The user would get only 78.47 B < 82 → the trade is **reverted** and the attack fails (None).

Step 11 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
▶      b2 = swap(x, y, u)           # user: sell u A
▶      if b2 < min_out:
▶          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
11 for a in [20, 50, 80]:
12     print(a, sandwich(1000, 1000, a, 100, 82))
```

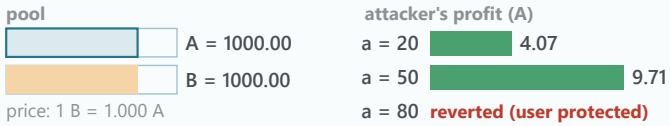
VARIABLES

a = **80** b2 = **78.47** min_out = **82**

OUTPUT

```
20 4.07
50 9.71
```

Code Trace: The Sandwich Attack and the Slippage Limit



A bigger front-run brings more profit, but the user's slippage limit cuts it off: the attacker's best a pushes the limit exactly. A tight tolerance is a simple defense against MEV.

Step 12 / 12

```
1  swap = lambda x, y, dx: y - x * y / (x + dx)  # xy=k
2  def sandwich(x, y, a, u, min_out):
3      b1 = swap(x, y, a)           # front-run: sell a A
4      x, y = x + a, y - b1
5      b2 = swap(x, y, u)           # user: sell u A
6      if b2 < min_out:
7          return None              # tx reverted
8      x, y = x + u, y - b2
9      a2 = swap(y, x, b1)          # back-run: sell b1 B
10     return round(a2 - a, 2)
▶ for a in [20, 50, 80]:
▶     print(a, sandwich(1000, 1000, a, 100, 82))
```

VARIABLES

a = 80 result = None

OUTPUT

```
20 4.07
50 9.71
80 None
```

Defenses Against MEV

Method	Idea	Cost
Tight slippage tolerance	Minimum amount out; a big front-run reverts the trade	Trades fail more often
Commit–reveal	Publish a hash first; reveal once the order is fixed	Two steps, delay; unrevealed transactions
Encrypted mempool	Decrypted after its position is fixed	Trust in key holders
Batch auction	A slot's orders clear **at one price**; order is irrelevant	Waiting for the slot
Auctioning the ordering right	Block-building rights are sold openly; MEV is competed away, not hidden	MEV remains; ordering concentrates in a few builders

Practical forms change often; the lasting idea: ordering power has value, to be reduced or priced openly.

A sandwich fails in a batch auction: the attacker's "buy in front" and "sell behind" clear with the user's **at one price**, so the profit is 0.

Computation and Scale

- Finding equilibria in large games can be hard (PPAD) → **approximate equilibria** and learning dynamics.
- Superhuman results in multiplayer poker came from regret minimization and search (Brown and Sandholm, 2019).
- Winner determination in combinatorial auctions is NP-hard; bidding languages and approximation algorithms are needed.
- Latency, data access and energy cost matter as much as theoretical guarantees.

items m	bundles $2^m - 1$
5	31
10	1,023
20	1,048,575
30	1,073,741,823

If every bidder reports a value for every bundle, the input grows exponentially. An n-player game with two actions each has 2^n strategy profiles: 1,048,576 for $n = 20$.

Winner Determination in Combinatorial Auctions

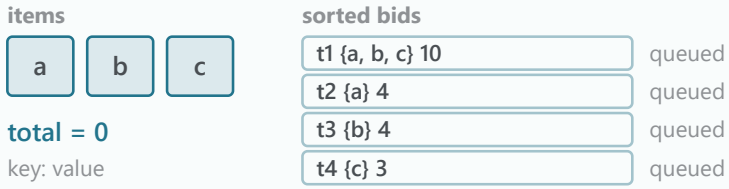
- Each bid is a bundle and a value: (S_i, v_i). Choosing **non-overlapping** bids that maximize total value is the **winner determination problem**.
- It is weighted set packing: NP-hard and, in general, hard to approximate well.
- Brute force tries 2^n subsets for n bids; in practice, solvers prune this search cleverly.
- VCG payments require solving the problem n + 1 times (removing each bidder once). Solving it approximately can break VCG's truthfulness guarantee.

Bid	bundle	value
t1	{a, b, c}	10
t2	{a}	4
t3	{b}	4
t4	{c}	3

Feasible choice	total
{t1}	10
{t2, t3}	8
{t2, t3, t4}	11

t1 overlaps the other three; the optimum is 11.

Code Trace: Greedy Winner Determination



Sorted by value: t1 (10), t2 (4), t3 (4), t4 (3). Total 0, no items used.

Step 1 / 11

```
1  def greedy(bids, key):
▶       ranked = sorted(bids, key=key, reverse=True)
▶       total, used = 0, set()
4       for S, v in ranked:
5           if not (S & used):        # no overlap
6               total += v
7               used |= S
8       return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
▶  print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

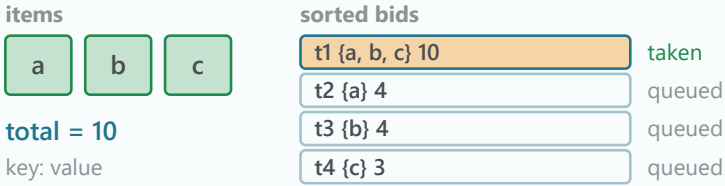
VARIABLES

`ranked = [t1, t2, t3, t4]`

`total = 0` `used = {}`

OUTPUT

Code Trace: Greedy Winner Determination



t1 = {a, b, c}: no item used yet → taken. Total 10; all three items are now used.

Step 2 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
▶              total += v
▶              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

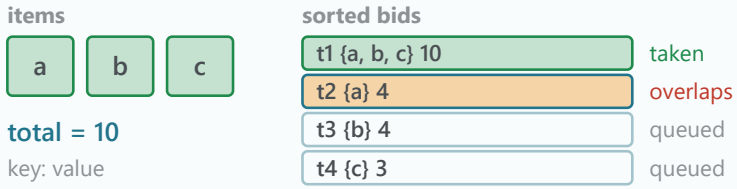
VARIABLES

S = {a,b,c} v = 10

total = 10 used = {a,b,c}

OUTPUT

Code Trace: Greedy Winner Determination



t2 = {a}: a is already used → nonempty intersection, skipped.

Step 3 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
6              total += v
7              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

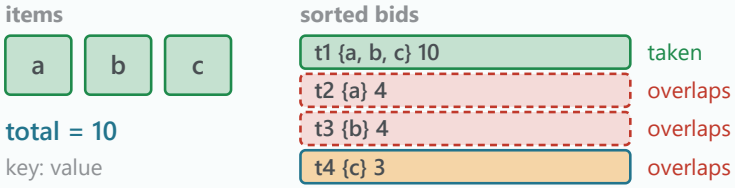
VARIABLES

S = {a} v = 4 total = 10

used = {a,b,c}

OUTPUT

Code Trace: Greedy Winner Determination



t3 and t4 overlap too. Because the big bundle was taken early, all the small bids were left out.

Step 4 / 11

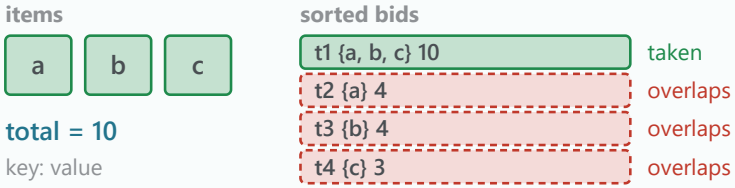
```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
6              total += v
7              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

VARIABLES

`S = {c}` `v = 3` `total = 10` `used = {a,b,c}`

OUTPUT

Code Trace: Greedy Winner Determination



Result **10**; the optimum was 11. Greedy by value is fooled by big bundles.

Step 5 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
4      for S, v in ranked:
5          if not (S & used):        # no overlap
6              total += v
7              used |= S
▶      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
▶  print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

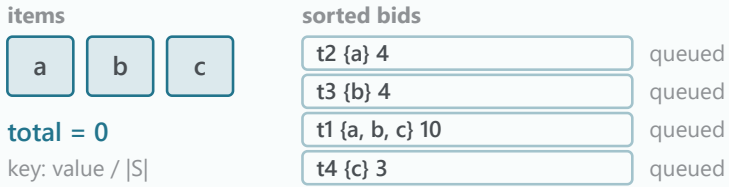
VARIABLES

`total = 10`

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



Value per item: t2 and t3 (4), t1 (10/3 ≈ 3.33), t4 (3). On ties, sorted keeps the original order.

Step 6 / 11

```
1  def greedy(bids, key):
▶      ranked = sorted(bids, key=key, reverse=True)
▶      total, used = 0, set()
4      for S, v in ranked:
5          if not (S & used):        # no overlap
6              total += v
7              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
▶  print(greedy(T, lambda t: t[1] / len(t[0])))
```

VARIABLES

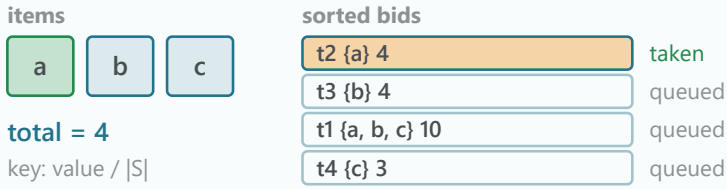
`ranked = [t2, t3, t1, t4]`

`total = 0` `used = {}`

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



t2 = {a} is taken: total 4.

Step 7 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
▶              total += v
▶              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

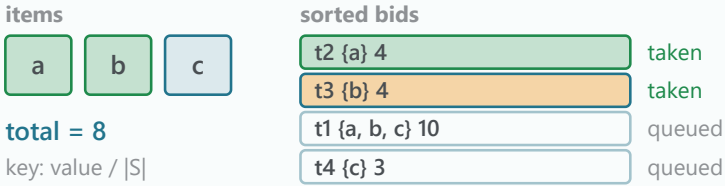
VARIABLES

`S = {a}` `v = 4` `total = 4` `used = {a}`

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



t3 = {b} is taken: total 8.

Step 8 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
▶              total += v
▶              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

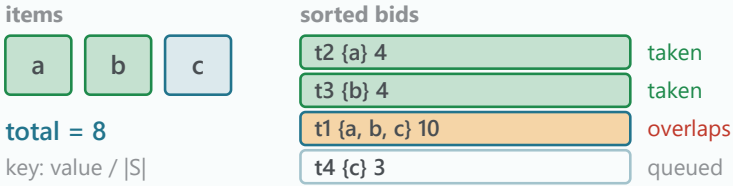
VARIABLES

S = {b} v = 4 total = 8 used = {a,b}

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



t1 = {a, b, c}: a and b are used → skipped (even though it is worth 10).

Step 9 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
6              total += v
7              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

VARIABLES

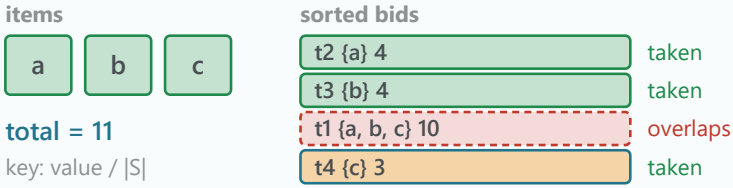
`S = {a,b,c}` `v = 10`

`total = 8` `used = {a,b}`

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



t4 = {c} is free → taken: total 11.

Step 10 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
▶      for S, v in ranked:
▶          if not (S & used):        # no overlap
▶              total += v
▶              used |= S
8      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
11 print(greedy(T, lambda t: t[1] / len(t[0])))
```

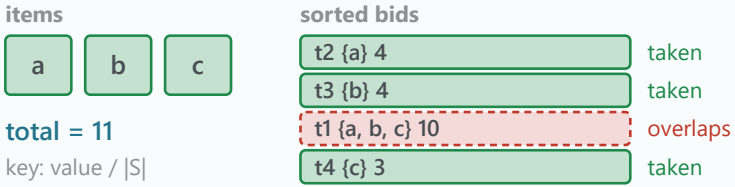
VARIABLES

`S = {c}` `v = 3` `total = 11` `used = {a,b,c}`

OUTPUT

```
10
```

Code Trace: Greedy Winner Determination



Result **11** = the optimum. But this key can fail too: with ({a, b}, 10) and ({a}, 6) it takes 6 first. Sorting by value/√|S| guarantees a factor of √m in the worst case (Lehmann et al., 2002).

Step 11 / 11

```
1  def greedy(bids, key):
2      ranked = sorted(bids, key=key, reverse=True)
3      total, used = 0, set()
4      for S, v in ranked:
5          if not (S & used):        # no overlap
6              total += v
7              used |= S
▶      return total
9  T = [({'a','b','c'}, 10), ({'a'}, 4), ({'b'}, 4), ({'c'}, 3)]
10 print(greedy(T, lambda t: t[1]))
▶  print(greedy(T, lambda t: t[1] / len(t[0])))
```

VARIABLES

`total = 11`

OUTPUT

```
10
11
```

Other Research Directions

- **Influence maximization in social networks**: a firm gives free products to k people; which k "seeds" maximize the spread? (Kempe, Kleinberg and Tardos, 2003)
- **The independent cascade model**: each newly influenced person gets one chance to influence each neighbor, with probability p.
- Optimal seeding is NP-hard; but spread has **diminishing returns** (submodular), so greedy selection guarantees **1 − 1/e ≈ 63%** of the optimum.

Multi-agent reinforcement learning

Other learners make the environment non-stationary; single-agent convergence guarantees fail.

Sustainable systems

Carbon permit markets, electricity demand response: pricing externalities (Pigouvian tax).

Robustness

Mechanisms that survive adversarial, faulty or boundedly rational players.

Also: information design (Bayesian persuasion: steering decisions via a disclosure rule), preference aggregation in AI alignment (social choice), prediction markets, and peer prediction for unverifiable reports.

Open Research Questions

Human + AI markets

How should rules change when people and agents interact in the same market?

Measuring trade-offs

How should we measure the trade-offs among efficiency, privacy and fairness?

Adaptive robustness

How can strategyproof mechanisms adapt to changing behavior?

Theory and data

How do we test models' assumptions against real platform data?

Equilibria of learning agents

Where do algorithms learning against each other converge, and how can collusion be prevented?

Verifiable mechanisms

How can we prove or audit that a learned rule is incentive compatible?

Summary

Topic	Description	Number from this chapter
Agent-to-agent bargaining	Backward induction; patience and the last word are power	T = 4: A 18.1; infinite horizon 52.6
Algorithmic collusion	A trigger strategy sustains cooperation if δ is large enough	$\delta \geq 1/2$
Prices from data	Empirical revenue maximization; overfitting with few samples	p = 0.47: empirical 0.294, true 0.249
Exponential mechanism	Probability ∝ exp(ε·revenue/2Δ); privacy + approximate IC	$\varepsilon = 2$: ratio ≤ $e^2 \approx 7.39$ (1.44 in the example)
Base fee	+12.5% for a full block, −12.5% for an empty one; the fee is burned	100 → 160.2 → 115
MEV	Ordering power carries value; the sandwich attack	user −8.09 B, attacker +9.71 A
Map	Equilibrium → design → applications → learning → the future	one concept map from preliminaries to future directions

Common Mistakes

Forgetting the number of firms

Patience that suffices for two firms does not suffice in a crowded market: the threshold is $\delta \geq 1 - 1/n$, 0.80 for $n = 5$.

Confusing finite and infinite horizons

With a finite horizon the last proposer is strong (A gets 18.1 at $T = 4$); with an infinite horizon the first proposer is slightly ahead (52.6).

Mistaking empirical revenue for true revenue

A price chosen and measured on the same data looks better than it is; separate validation data are needed.

Linear proportions in the exponential mechanism

Probability is proportional to $\exp(\varepsilon \cdot R/2\Delta)$, not to revenue; revenue gaps show up exponentially in probability ratios.

Updating the base fee additively

The change is multiplicative: two full blocks give $100 \to 112.5 \to 126.6$ (not 125). The base fee is burned, not paid to the producer.

Thinking a loose slippage tolerance is safe

It raises the chance that the trade goes through but leaves the attacker more room to front-run.

WRAP-UP

Sample Problems

1. 100 TL, two rounds (A then B proposes), $\delta = 0.9$. What is the split in subgame perfect equilibrium? What is A's share with an infinite horizon?
2. Three firms split the monopoly profit (100) equally; a deviator gets 100 for one period, then 0. For which δ does the trigger strategy hold?
3. Sample values 0.9, 0.6, 0.5, 0.2. Which price maximizes empirical revenue, and what is the revenue?
4. With $r = 100$, if three full blocks and one empty block arrive, what is the base fee?
5. In the GSP–VCG comparison at the Online Platforms node of the map, which topic does VCG come from, and which topic does the second-price idea behind GSP come from?
6. In the exponential mechanism, two candidate prices have revenues 2 and 1, $\varepsilon = 2$, $\Delta = 1$. What are the probabilities?

▶ **Show answers**

References

- T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, Cambridge University Press, 2016.
- A. Rubinstein, "Perfect Equilibrium in a Bargaining Model", *Econometrica*, 1982.
- E. Calvano, G. Calzolari, V. Denicolò, S. Pastorello, "Artificial Intelligence, Algorithmic Pricing, and Collusion", *American Economic Review*, 2020.
- V. Conitzer, T. Sandholm, "Complexity of Mechanism Design", *UAI*, 2002.
- P. Dütting, Z. Feng, H. Narasimhan, D. Parkes, S. Ravindranath, "Optimal Auctions through Deep Learning", *ICML*, 2019.
- F. McSherry, K. Talwar, "Mechanism Design via Differential Privacy", *FOCS*, 2007.
- R. Cole, T. Roughgarden, "The Sample Complexity of Revenue Maximization", *STOC*, 2014.
- T. Roughgarden, "Transaction Fee Mechanism Design", *ACM EC*, 2021.
- P. Daian et al., "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability", *IEEE S&P*, 2020.
- N. Brown, T. Sandholm, "Superhuman AI for Multiplayer Poker", *Science*, 2019.
- D. Kempe, J. Kleinberg, É. Tardos, "Maximizing the Spread of Influence through a Social Network", *KDD*, 2003.
- H. Chung, E. Shi, *SODA*, 2023; H. Chung, T. Roughgarden, E. Shi, *ACM EC*, 2024 (impossibility results for transaction fee mechanisms).
- D. Lehmann, L. I. O'Callaghan, Y. Shoham, *J. ACM*, 2002; R. Axelrod, *J. Conflict Resolution*, 1980.

After the Course: Books, Course Pages, Tools

Books

- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge UP, 2007.
- A. R. Karlin, Y. Peres, *Game Theory, Alive*, AMS, 2017.
- D. Easley, J. Kleinberg, ***Networks, Crowds, and Markets***, Cambridge UP, 2010.
- R. A. Hearn, E. D. Demaine, *Games, Puzzles, and Computation*, A K Peters, 2009.

Course pages

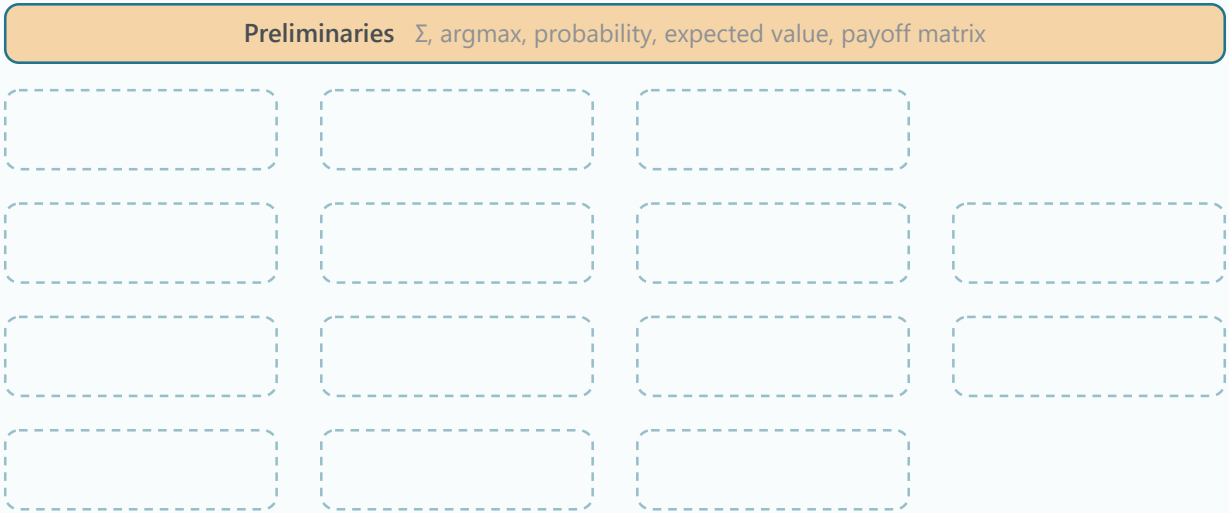
- T. Roughgarden, CS364A, Stanford, 2013
- N. Shah, CSC304, Toronto, 2019
- M. Dinitz, 601.436/636, Johns Hopkins, 2022
- A. Procaccia, 15-896, CMU, 2016
- M. Hoefer, Algorithmic Game Theory, Frankfurt, 2022–2023

Tools

- **Gambit**: drawing game trees and matrices, computing equilibria.
- **Nashpy**: equilibria of two-player games in Python.
- **Game Theory 101**: a video series.

APPENDIX

Course Map: Every Chapter in One Picture



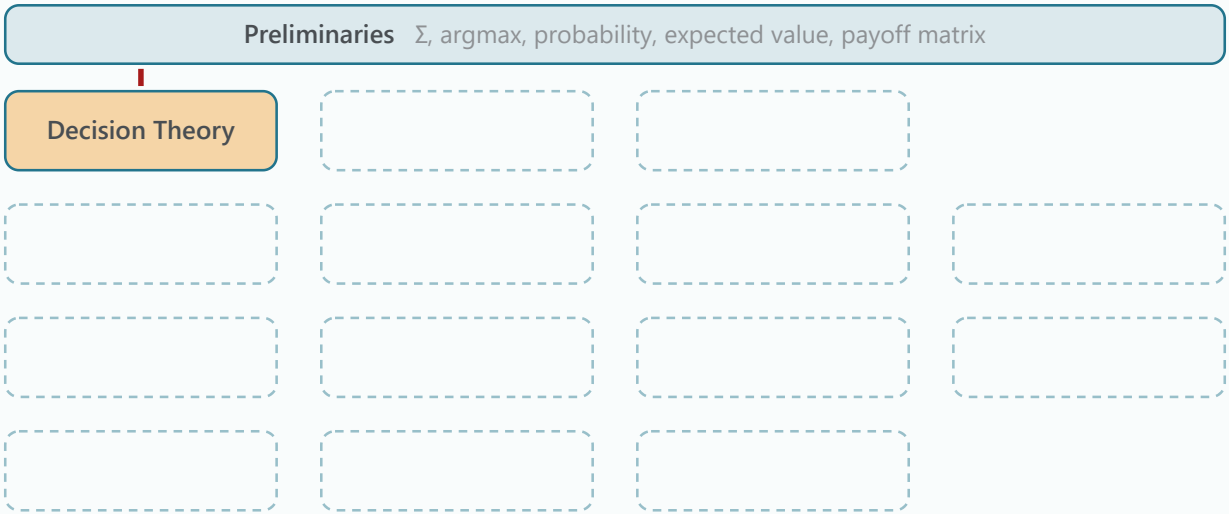
Preliminaries: Σ and argmax, probability and expected value, reading a payoff matrix, the indifference equation. Every chapter uses these tools.

Step 1 / 15

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

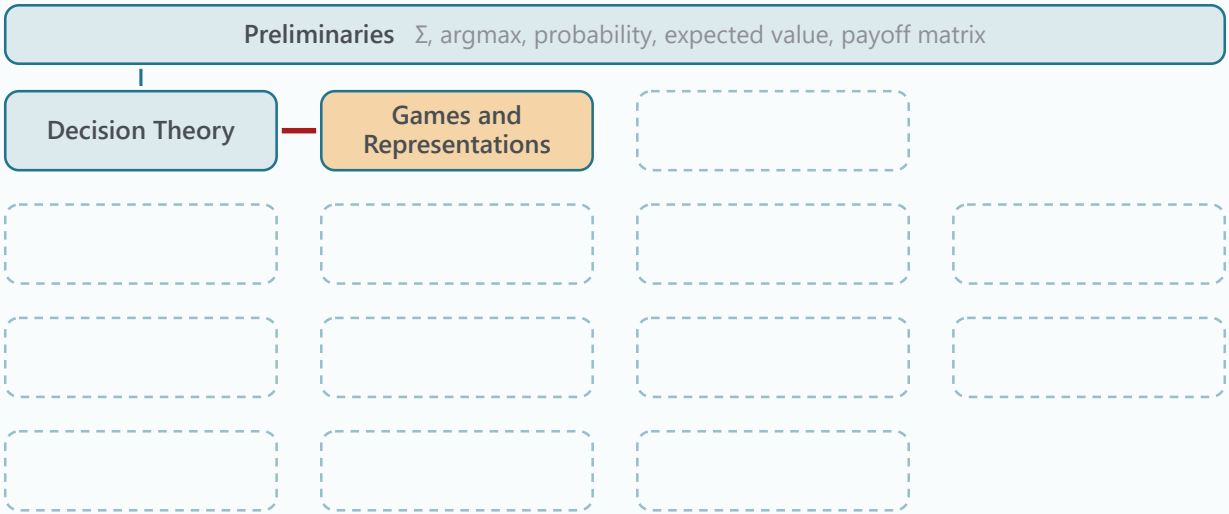


Decision Theory ← Preliminaries: single-person decisions: expected utility, risk attitudes, Bayesian decisions and the value of information.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

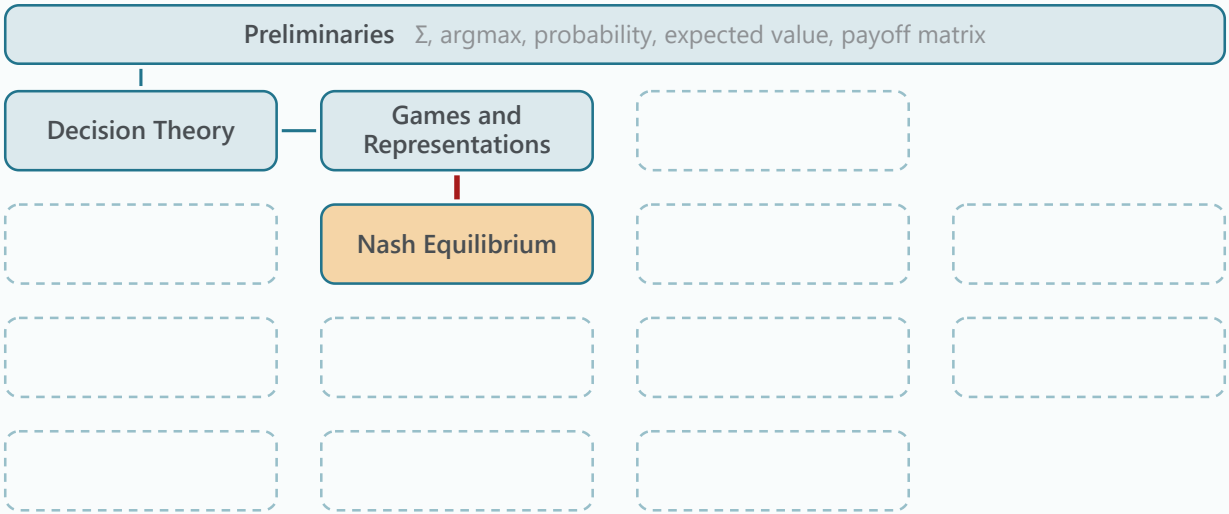


Games and Representations ← Decision Theory: with more than one decision maker, a game arises: normal and extensive form, backward induction.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

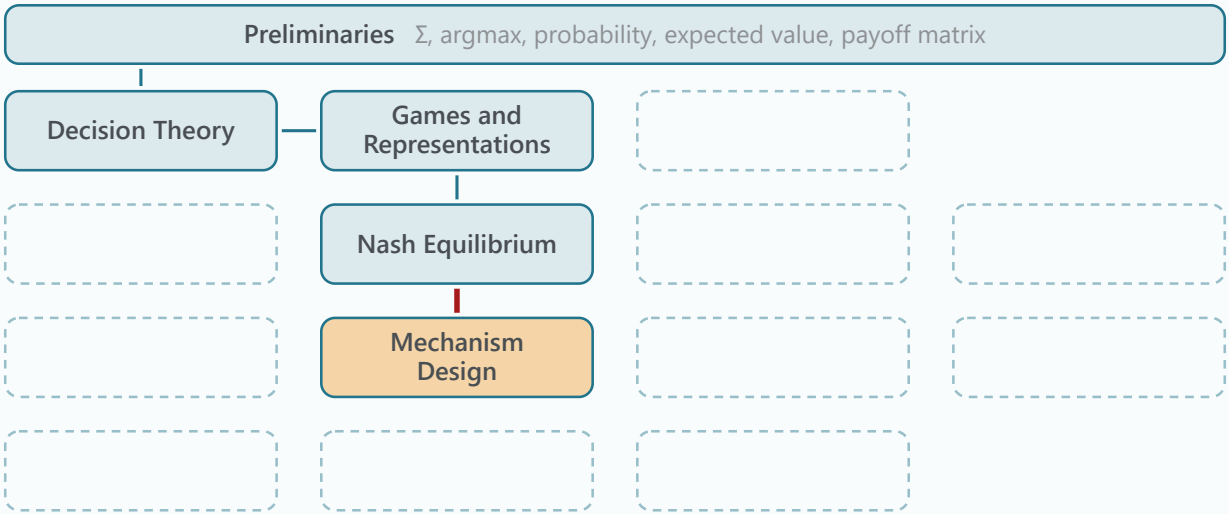


Nash Equilibrium ← Games and Representations: equilibria marked in 2×2 games lead to the general definition of Nash equilibrium and the existence theorem.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

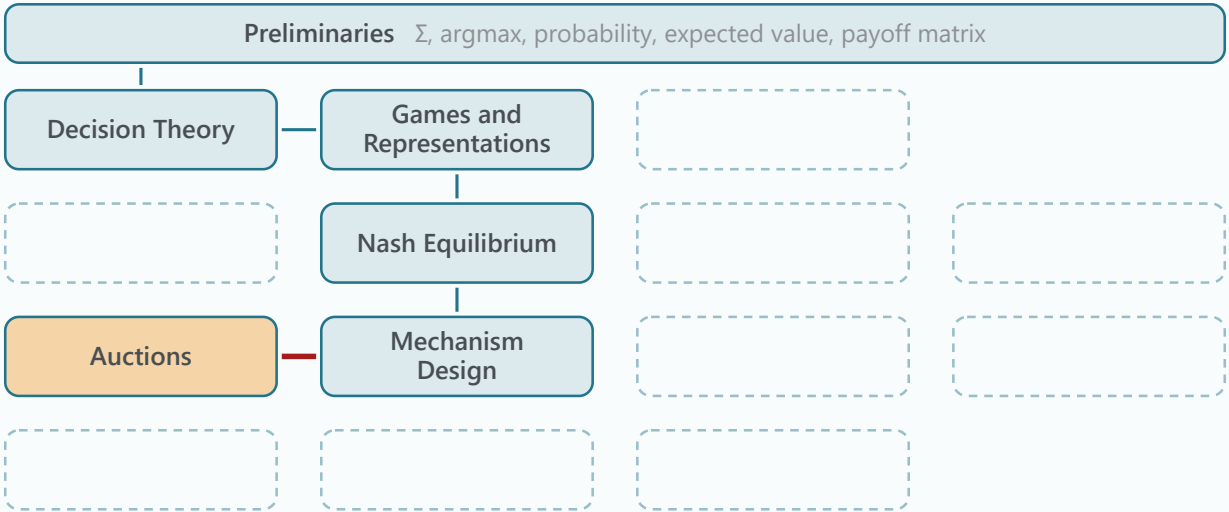


Mechanism Design ← Nash Equilibrium: equilibrium + incentives: instead of solving a game, design the rule that yields the desired equilibrium.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

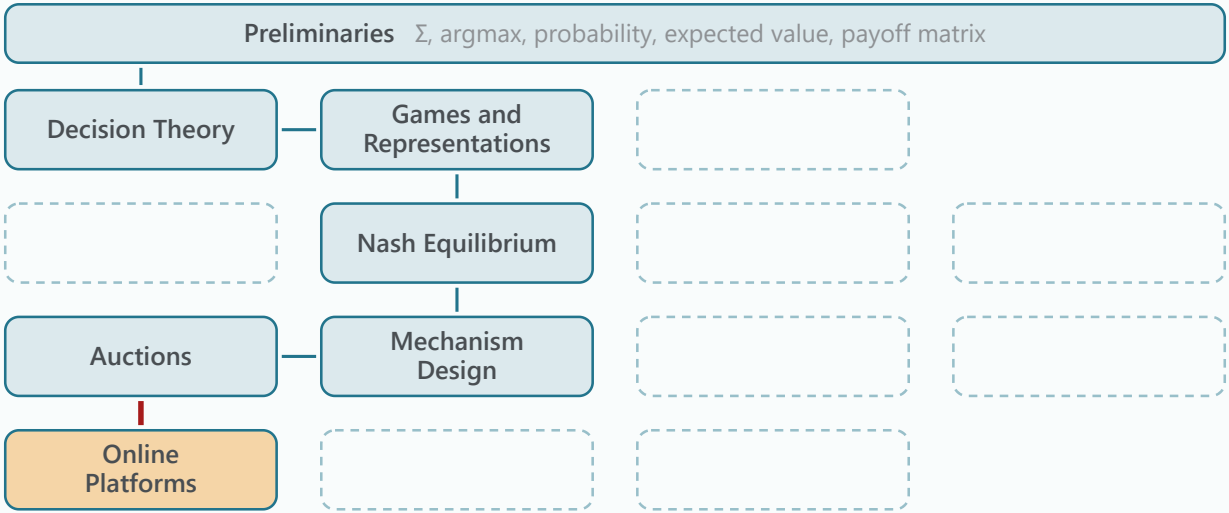


Auctions ← Mechanism Design: VCG for a single item is the second-price (Vickrey) auction.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

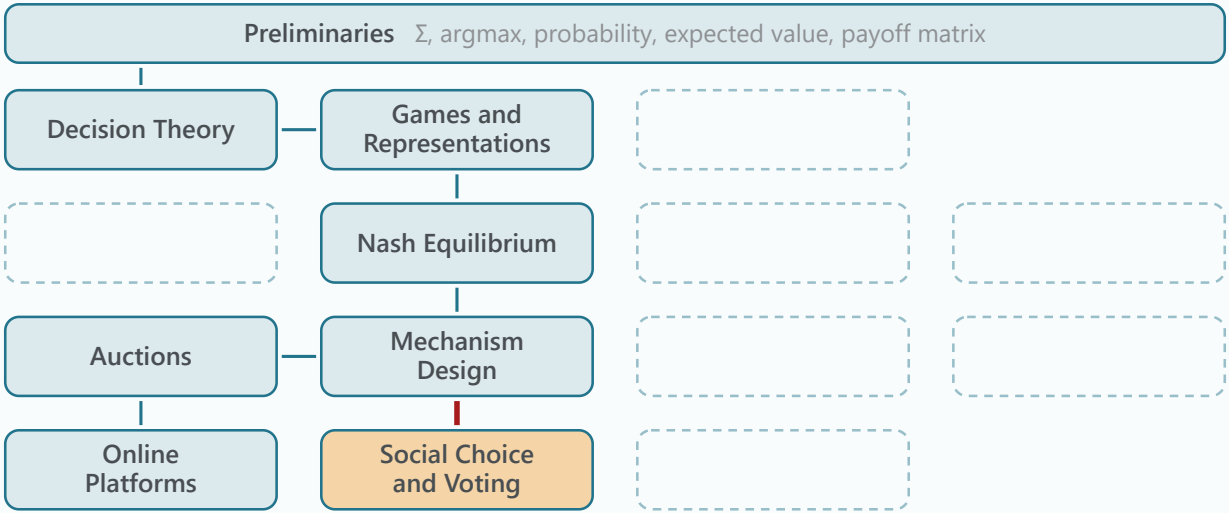


Online Platforms ← Auctions: sponsored search auctions: GSP generalizes Vickrey's idea to many slots.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

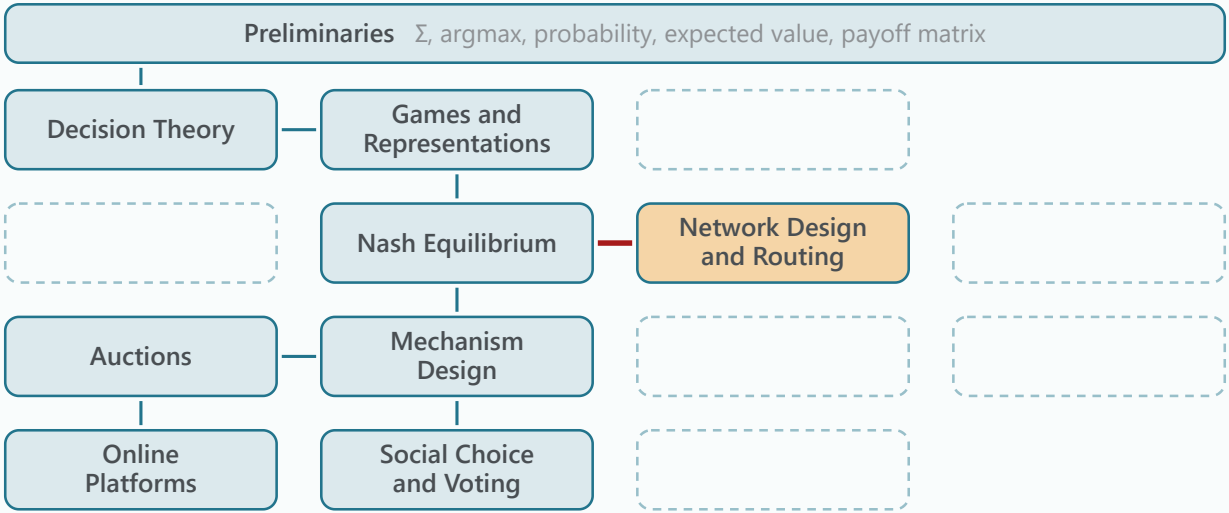


Social Choice and Voting ← Mechanism Design: social choice functions → voting rules; Gibbard–Satterthwaite.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

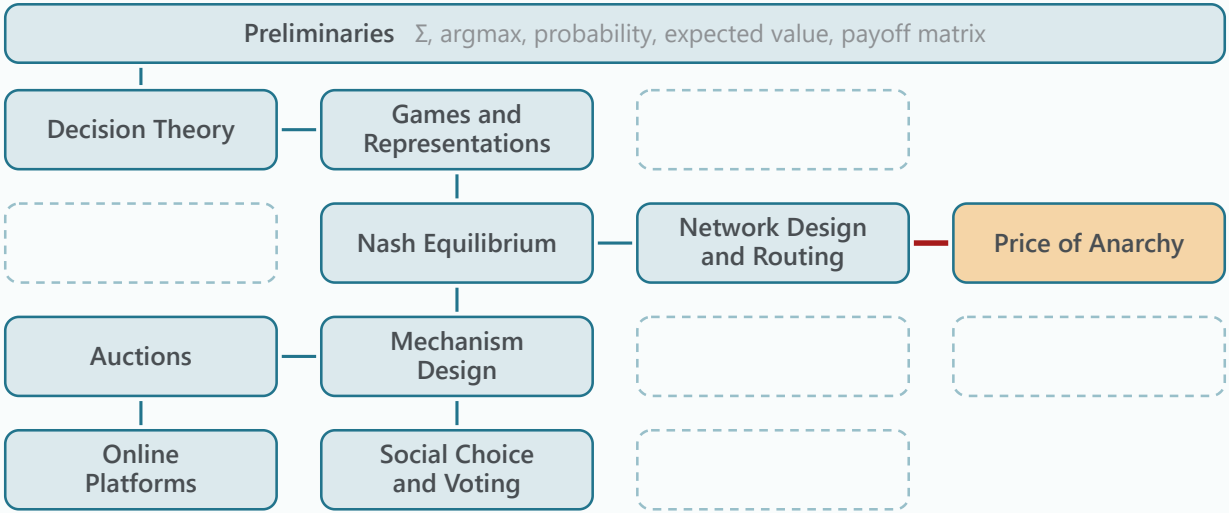
Course Map: Every Chapter in One Picture



Network Design and Routing ← Nash Equilibrium: Wardrop (Beckmann) for nonatomic traffic, pure Nash (Rosenthal) for atomic games.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

Course Map: Every Chapter in One Picture

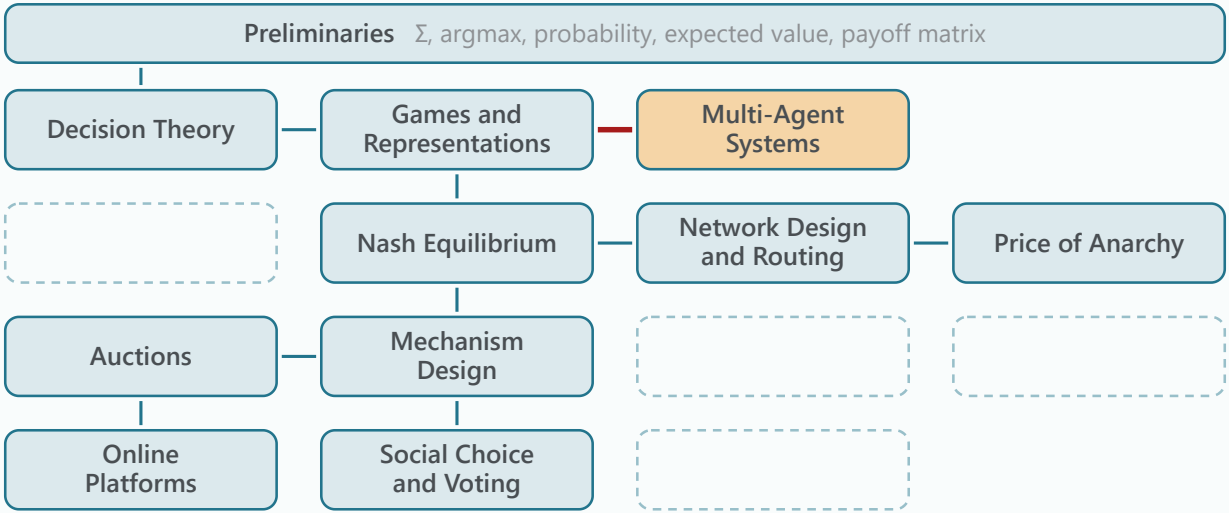


Price of Anarchy ← Network Design: equilibrium quality in the Pigou and Braess networks → Price of Anarchy 4/3; congestion tolls.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

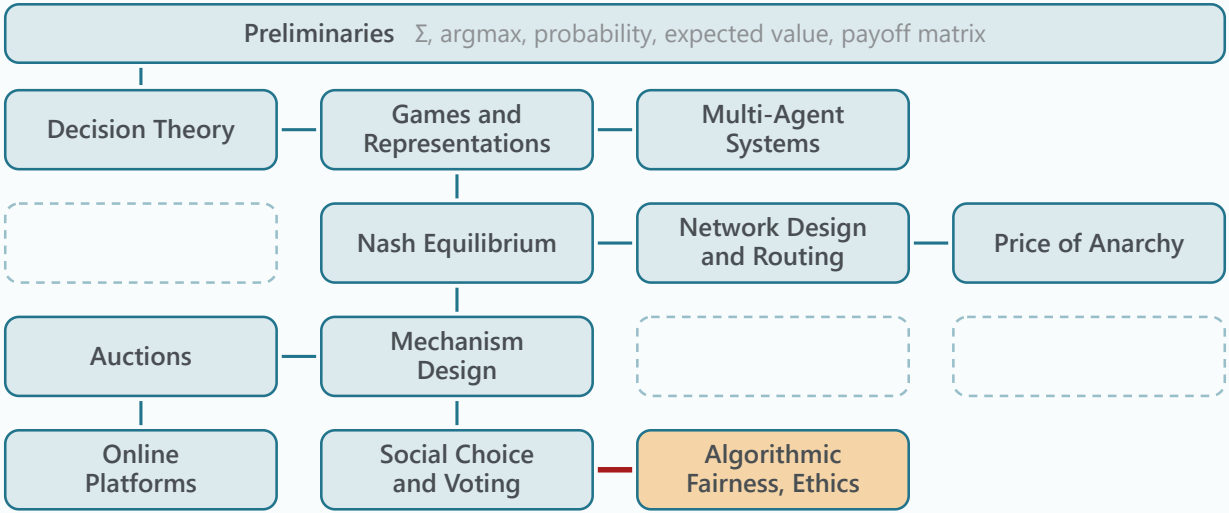


Multi-Agent Systems ← Games and Representations: cooperative vs noncooperative → coalitional games, the Shapley value.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

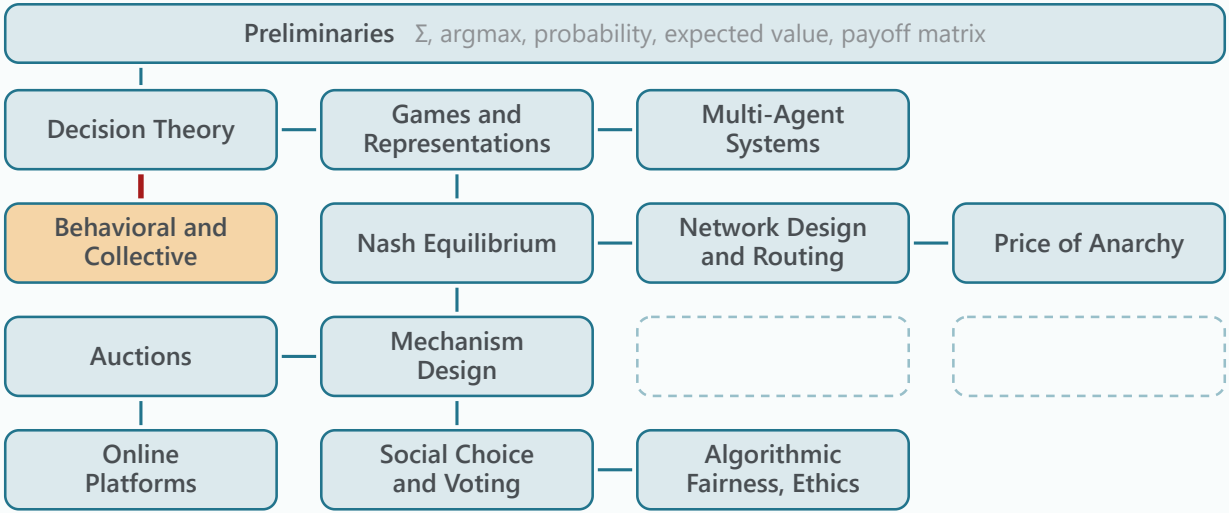
Course Map: Every Chapter in One Picture



Algorithmic Fairness and Ethics ← Social Choice and Voting: from collective decisions to fairness: criteria and fair division.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

Course Map: Every Chapter in One Picture

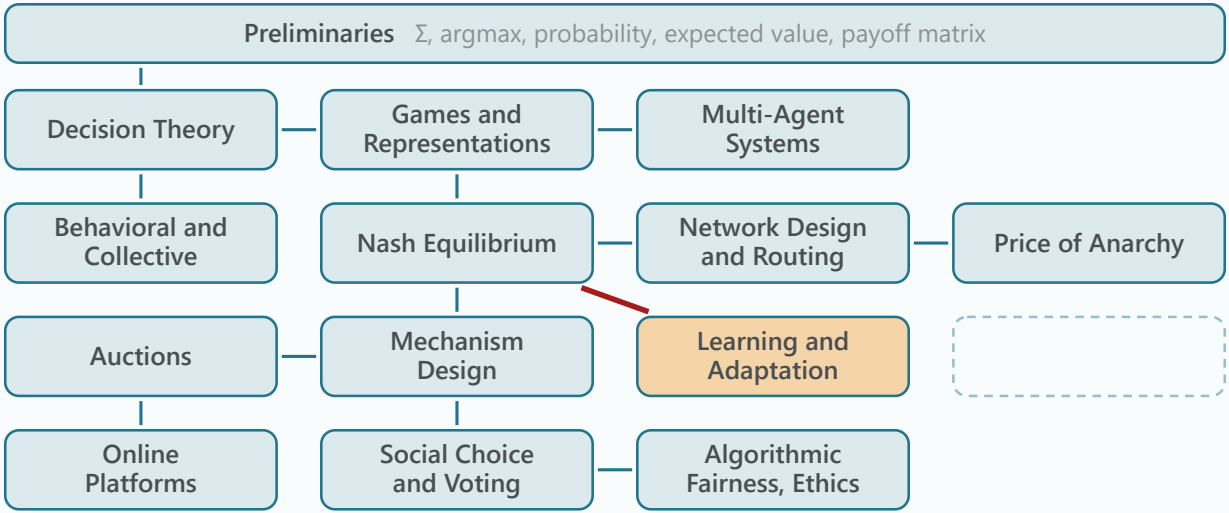


Behavioral and Collective Decisions ← Decision Theory: people deviate from expected utility (loss aversion); in the commons individual and collective interests diverge.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture

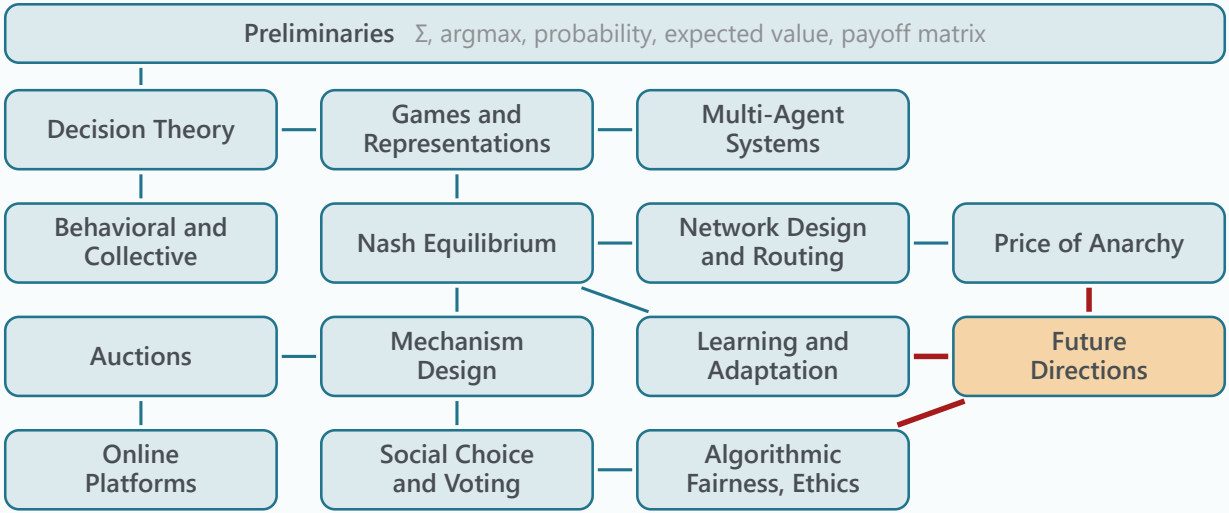


Learning and Adaptation ← Nash Equilibrium: how are equilibria found? Players can converge to equilibrium by learning.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Course Map: Every Chapter in One Picture



Future Directions: scale (Price of Anarchy), learning agents (learning) and ethics (fairness) meet in today's open questions.

Chapter numbers: Preliminaries 0 · Decision Theory 1 · Games and Representations 2 · Nash Equilibrium 3 · Mechanism Design 4 · Auctions 5 · Online Platforms 6 · Social Choice and Voting 7 · Network Design and Routing 8 · Price of Anarchy 9 · Multi-Agent Systems 10 · Algorithmic Fairness and Ethics 11 · Behavioral and Collective Decisions 12 · Learning and Adaptation 13 · Future Directions 14.

APPENDIX

Agent Tournament: A Reproducible Experiment Protocol

Language-model agents are stochastic: the same pairing can score differently on each run. A result is trustworthy if someone else can repeat the experiment and get the same result (reproducibility).

What is recorded

- A fixed random seed
- The full text of the prompt
- Model version and temperature
- Opponent order and number of rounds: kept fixed and stated in the report

What is reported

- Each pairing is repeated at least N times (e.g. N = 30)
- Mean and spread: standard deviation or a 95% confidence interval
- Code and game logs are shared in a public repository

Example: over 30 repetitions the mean score is 2.6 with standard deviation 0.4. The approximate 95% confidence interval is $2.6 \pm 1.96 \cdot 0.4 / \sqrt{30} \approx 2.6 \pm 0.14$, i.e. [2.46, 2.74].

APPENDIX

An End-to-End Example: Pizza Voting

Person	1.	2.	3.
Ali	mushroom	sucuk	olive
Banu	sucuk	mushroom	olive
Cem	olive	sucuk	mushroom

- **The model:** 3 players; strategy = the reported ranking; payoff = 2, 1, 0 by the true ranking.
- **Plurality:** each candidate gets one first place, a 1–1–1 tie.
- **Borda (2, 1, 0):** mushroom 3, sucuk 4, olive 2 → sucuk. Sucuk wins both pairwise comparisons 2–1: the Condorcet winner.

Analysis: strategic voting

If Ali buries sucuk and reports "mushroom, olive, sucuk", the Borda scores become 3–3–3. With alphabetical tie-breaking, **mushroom** wins: by lying, Ali gets his favorite.

Try it yourself

Console version: read the rankings and tally plurality, two-round runoff, Borda, Condorcet and approval voting side by side. Web version: let the user change one player's report and watch the winner shift. Changing the tie-breaking rule also changes the winner.

Burying in Borda (putting the strongest rival last) is a small instance of the Gibbard–Satterthwaite theorem.

The End

Chapter 14: Future Directions

Previous: Chapter 13: Learning and Adaptation