Chapter 4: Mechanism Design

Algorithmic Game Theory

Social choice functions | Direct mechanisms | Revelation principle | Incentive compatibility | Myerson's lemma | VCG | Gale–Shapley | Median mechanism | TTC | Bayesian games

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_04_Mechanism_Design.html

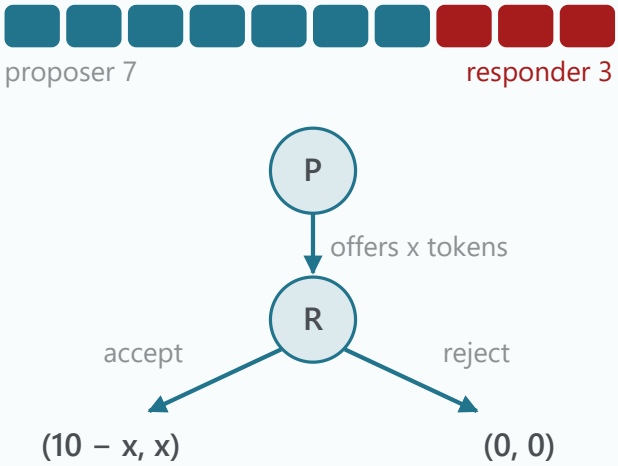
Warm-up Game: The Ultimatum Game

The rules

The proposer splits 10 tokens and offers the responder a share; if the responder accepts, the split goes ahead, and if not, both get 0.

Think about it

- As the responder, would you accept an offer of 1 token?
- As the proposer, how many tokens would you offer?



Learning Outcomes

Concepts

Explain social choice functions, direct mechanisms, the revelation principle and Bayes–Nash equilibrium.

Incentive analysis

Test whether a mechanism is incentive compatible (IC) and individually rational (IR).

Computing VCG

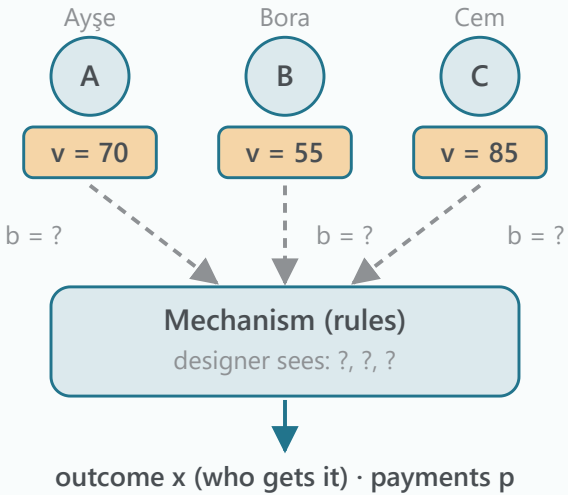
Compute the efficient outcome and each player's VCG payment from “what if they were absent” welfare.

Design without money

Find stable matchings with Gale–Shapley and exchanges with TTC; show why the median mechanism is strategyproof.

What Is Mechanism Design?

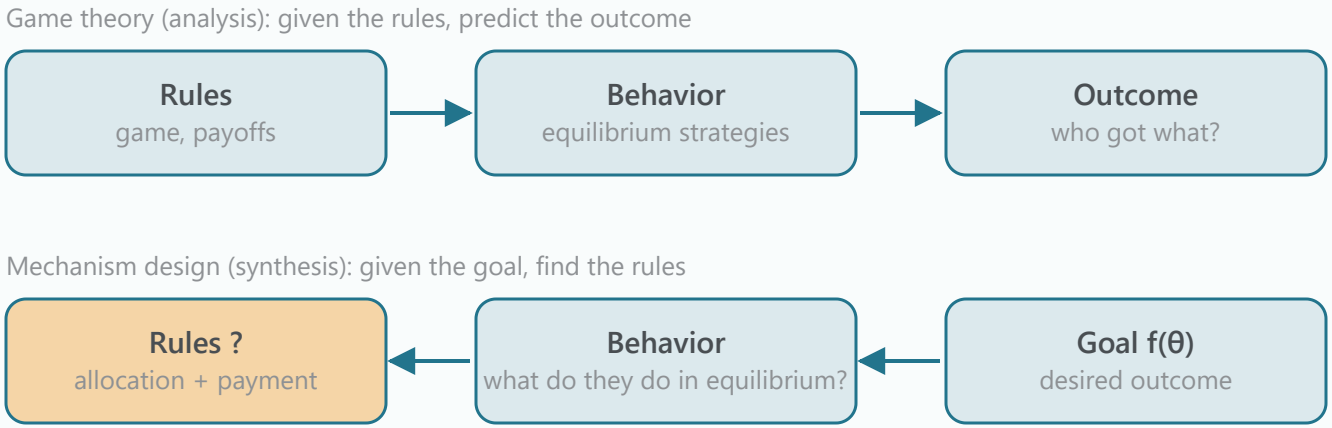
- Game theory “in reverse”: the game is not given — **we build it**.
- We design the rules (who gets what, who pays what) to reach a desired outcome.
- The difficulty: players hold **private information** (values, preferences, costs) hidden from the designer.
- Goal: reach the desired outcome even though players are selfish and strategic.
- Examples: taxes, procurement, voting systems, school admissions, ad auctions.



2007 Nobel Prize in Economics: Hurwicz, Maskin and Myerson, “for having laid the foundations of mechanism design theory”. In 2012 Roth and Shapley were honored for stable matching and market design.

BASIC CONCEPTS

Game Theory and Mechanism Design



Game theory: **Rules** → **Outcome**. Mechanism design: **Outcome** → **Rules**. The designer cannot force the players; it only chooses the rules.

BASIC CONCEPTS

Incentives

People respond to incentives

If the rules make some behavior profitable, players move toward it.

A good mechanism

Rewards honesty and deters strategic lying: *incentive compatibility*.

The free-rider problem

When funding a public good, everyone wants to understate their share and benefit from what others pay.

Strategic voting

Voters may back a candidate with a real chance instead of their true favorite.

Misaligned incentives → misreported information → inefficient or unfair outcomes.

Real-World Applications

Tax systems

Income is private information; audits and rates must encourage honest reporting.

Funding public goods

Parks, roads, libraries: who should pay how much, and should the project go ahead?

Procurement and auctions

Public procurement, spectrum licenses, artworks.

Online advertising

Ad auctions that run within milliseconds of every search query.

Matching markets

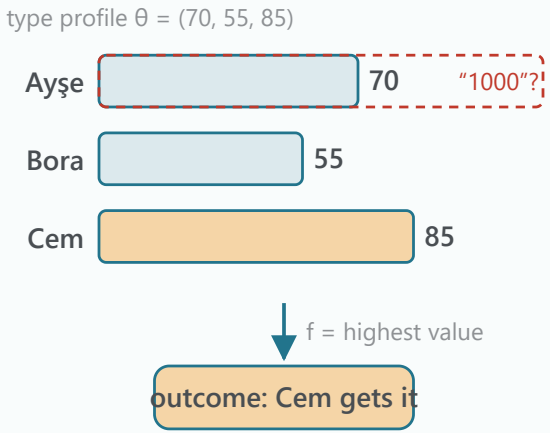
School admissions, medical residencies, dorm room swaps: no money, only rules.

Grading group projects

If students report their own contributions, the grading rule must deter exaggeration.

Social Choice Functions

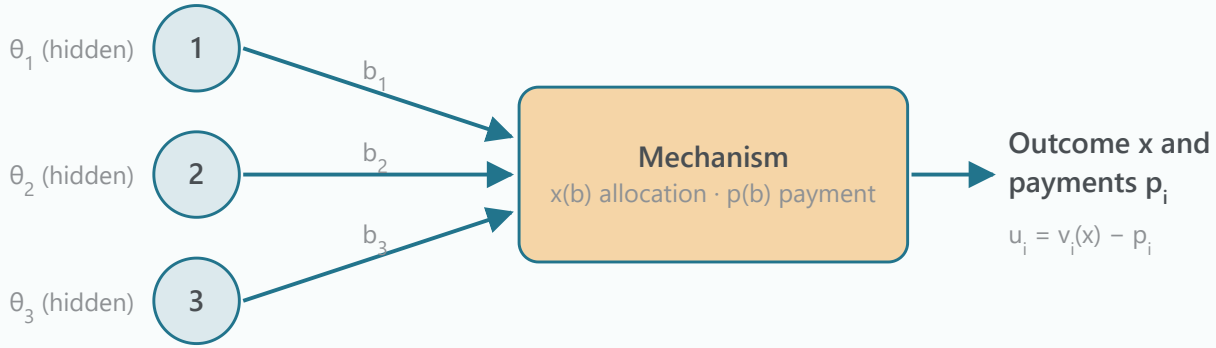
- Each player has a **type** $\theta_i \in \Theta_i$: a value, preference or cost. Types are private.
- A set of possible outcomes O (e.g. who gets the item, which project is built).
- A **social choice function** $f : \Theta_1 \times \ldots \times \Theta_n \to O$ assigns the "desired" outcome to each type profile.
- The problem: the designer does not know θ; to implement f(θ), it must get the information from the players.



Efficient choice: *f(θ)* = the highest-value player → Cem. But what if everyone says "my value is 1000"?

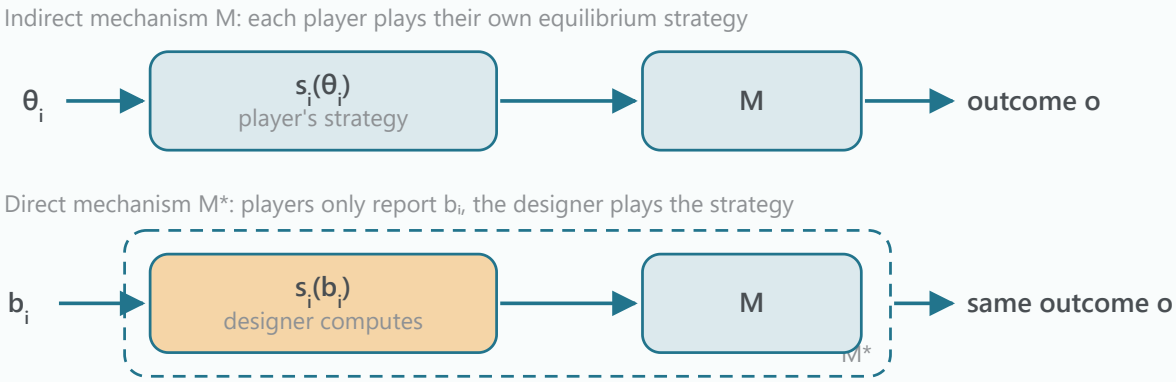
The question of mechanism design: are there rules that implement f even when players pursue **their own interests**?

Direct Mechanisms



- In a direct mechanism, participants **report** their private information (types): b_i (the report; it may be a lie).
- From the reports, the mechanism computes the outcome x(b) and payments p(b). Quasi-linear utility: $u_i = v_i(x) - p_i$ (v_i: the valuation of a player of type θ_i; for a single item $v_i = \theta_i$).
- In an indirect mechanism, a strategy is something else: raising a hand in an auction, bidding in turn…

The Revelation Principle



- If an outcome can be implemented in equilibrium by some indirect mechanism, then under suitable conditions a direct mechanism in which truthful reporting is an equilibrium implements the same outcome.
- Why? Lying amounts to playing an off-equilibrium strategy in M, and that was not profitable in equilibrium.
- Example: in an English auction everyone stays in up to their value. If the designer says “write down your value and I'll stay in for you”, we get a direct mechanism (second price) where writing the truth is best.
- The principle does not always make truth-telling a dominant strategy: the incentive condition depends on the equilibrium concept used (dominant strategy / Bayes–Nash).

Incentive Compatibility (IC) and Individual Rationality (IR)

Dominant-strategy incentive compatible (DSIC)

With true type θ_i, truth is at least as good as any lie b_i, whatever others report:

$$u_i(\theta_i;\ \theta_i,\ b_{-i}) \geq u_i(\theta_i;\ b_i,\ b_{-i})$$

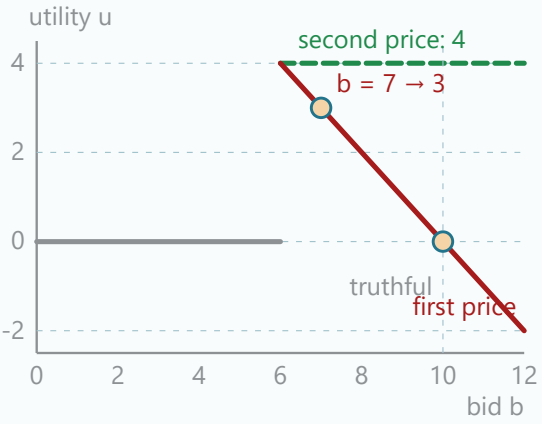
Bayesian incentive compatible (BIC)

If others are truthful, truth maximizes *expected* utility. DSIC ⇒ BIC.

Individual rationality (IR)

Participating is at least as good as staying out:

$$u_i(\theta_i;\ \theta_i,\ b_{-i}) \geq 0 \quad \text{(truthful)}$$



Value 10, rival bid 6. Paying your own bid (first price): truth yields 0, a bid of 7 yields 3: **not IC**. Paying the rival's bid (second price): any bid above 6 yields 4.

Back to the Game: The Ultimatum Game

Theory's answer

A token-only responder gets x by accepting, 0 by rejecting: accepting any $x \geq 0$ is a best response (IR); at $x = 0$ rejecting is too. The proposer offers "take it or leave it". Two pure SPE with integer offers: **(10, 0)** and **(9, 1)** (rejects at 0).

In practice

Since Güth, Schmittberger and Schwarze (1982): the most common offer is 40–50%, and about half of offers below 20% are rejected (Camerer, 2003). People punish unfairness at their own cost.

Responder	Smallest offer accepted	Proposer's payoff
Theory (tokens only)	1 (0 if 0 is also accepted)	9 (10 in that case)
Level 1: lenient	1	9
Level 2: strategic	3	7
Level 3: fairness-minded	5	5

IC design needs the right utility: if players care about fairness, IR becomes "x reaches a fair share", not "$x \geq 0$".

Step by Step: Is Proportional Cost Sharing IC?

	Ayşe	Bora	Cem	Total
True value v	30	25	20	75
Report	—	—	—	—
Payment	—	—	—	—
Utility u = v − payment	—	—	—	—
Park built?	—			

A park costs 60 thousand TL. Rule: if the reports sum to ≥ 60, the park is built and the cost is split **in proportion to the reports**.

Step 1 / 4

The fix: a report should affect its sender's payment **only by changing the chosen outcome** → VCG.

Step by Step: Is Proportional Cost Sharing IC?

	Ayşe	Bora	Cem	Total
True value v	30	25	20	75
Report	30	25	20	75
Payment	24	20	16	60
Utility u = v − payment	**6**	**5**	**4**	15
Park built?	Yes (total ≥ 60)			

Everyone truthful: total 75 ≥ 60, the park is built. Payments 60·30/75 = 24, 20, 16; utilities 6, 5, 4. All ≥ 0 → IR holds.

The fix: a report should affect its sender's payment **only by changing the chosen outcome** → VCG.

Step by Step: Is Proportional Cost Sharing IC?

	Ayşe	Bora	Cem	Total
True value v	30	25	20	75
Report	30	25	5	60
Payment	30	25	5	60
Utility u = v − payment	**0**	**0**	**15**	15
Park built?	Yes (total ≥ 60)			

Cem reports 5: the total is still 60 and the park is still built; Cem now pays 5 and his utility goes **4** → **15**. Lying paid off → not IC.

The fix: a report should affect its sender's payment **only by changing the chosen outcome** → VCG.

Step by Step: Is Proportional Cost Sharing IC?

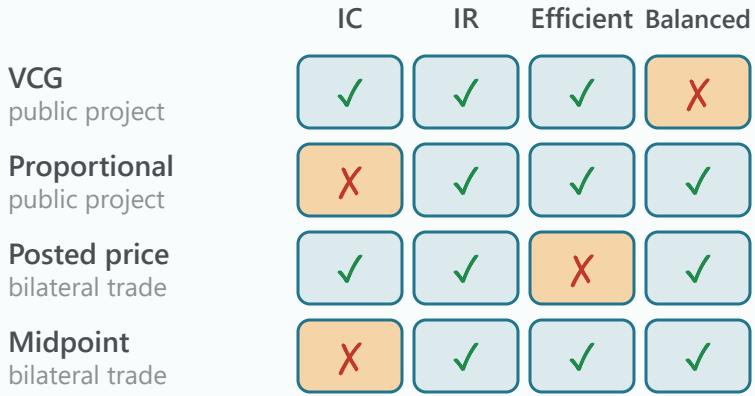
	Ayşe	Bora	Cem	Total
True value v	30	25	20	75
Report	15	12.5	10	37.5
Payment	0	0	0	0
Utility u = v − payment	**0**	**0**	**0**	0
Park built?	No (total < 60)			

If everyone halves their report by the same logic, the total is 37.5 < 60: no park, and total utility drops 15 → 0. Free riding causes inefficiency.

The fix: a report should affect its sender's payment **only by changing the chosen outcome** → VCG.

Criteria for Mechanism Design

- **Incentive compatibility (IC)**: revealing the truth is the best option.
- **Individual rationality (IR)**: participating is no worse than staying out.
- **Efficiency**: the outcome maximizes **social welfare** $SW(x) = \Sigma_i v_i(x)$ (under truthful reports).
- **Budget balance**: total payments ≥ 0 (weak: no outside funding needed) or $= 0$ (strong: no surplus either).



Sample mechanisms: each gives up one criterion.

These criteria cannot always be met together: in bilateral trade with independent values and costs whose ranges overlap, efficiency + (Bayesian) IC + IR + budget balance are jointly impossible (Myerson–Satterthwaite, 1983). Design means choosing the **trade-offs**.

Implementation Theory: Is Truth-Telling the Only Equilibrium?

- The revelation principle only says truthful reporting is *an* equilibrium; the same mechanism may have other, undesirable equilibria.
- Rules can make a good equilibrium possible without guaranteeing that players coordinate on it. Mechanisms whose every equilibrium yields the desired outcome are studied by implementation theory (further reading: Maskin, 1999).

A bad equilibrium in Vickrey

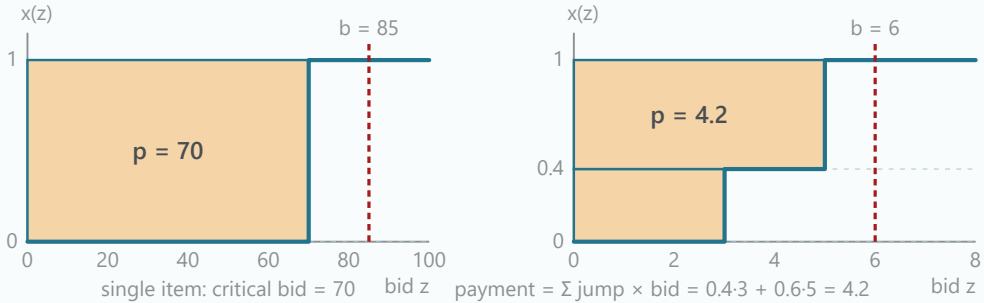
Values: Ayşe 10, Bora 5. Ayşe bids 0, Bora bids 100: Bora wins and pays 0 (utility 5). To win, Ayşe would have to beat 100 and pay 100 (utility −90), so she does not deviate. Bora cannot gain by deviating either, so these bids form a **Nash equilibrium** — yet the item goes to Bora, the lower-value player. Truthful bidding is a dominant strategy; this equilibrium rests on weakly dominated bids.

Lesson: under DSIC a player need not predict the opponent, which makes it more robust than designs relying only on Nash equilibrium. Still, it does not guarantee a unique equilibrium.

Single-Parameter Environments: Myerson's Lemma

- Each player has one private number: a per-unit value v_i. The mechanism chooses an allocation $x_i(b)$ and payment $p_i(b)$; utility is $v_i \cdot x_i - p_i$.
- **Myerson's lemma** (1981): x can be used in a DSIC mechanism ⇔ x is **monotone** (x_i does not decrease as the bid rises). The payment is then uniquely determined (with $p_i(0) = 0$):

$$p_i(b) = b_i \cdot x_i(b) - \int_0^{b_i} x_i(z, b_{-i})\, dz$$ = the shaded area **left** of the curve, between 0 and x(b) (the b × x(b) rectangle minus the area under the curve)



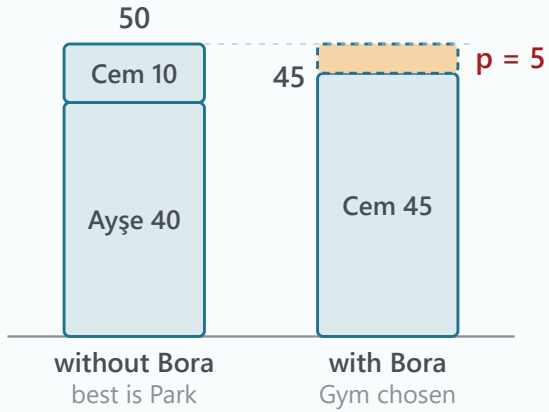
For 0/1 allocations, payment = **critical bid** (the lowest bid that still wins): exactly Vickrey's second price. If the payment is not the critical bid, IC fails: with threshold 50 and a fixed payment of 20, a player valued between 20 and 50 overbids past 50 and wins.

The Vickrey–Clarke–Groves (VCG) Mechanism

- Choose the outcome that maximizes social welfare under the reported values: $x^* = \text{argmax}_x \Sigma_i b_i(x)$ ($b_i(x)$: i's reported value for x).
- Each player's payment is the **externality** their participation imposes on the others (Clarke pivot rule):

$$p_i = \max_x \Sigma_{j \neq i} b_j(x) - \Sigma_{j \neq i} b_j(x^*)$$

= (others' best welfare without i) − (others' welfare with i)



Bora's externality in the public project example.

- A report affects the payment only through x^*; so, under suitable conditions, truthful reporting is a **dominant strategy**.
- A player's utility = social welfare − (a term independent of them): everyone wants to maximize social welfare.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	—	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	—	—	—		
(b) Σ without i	—	—	—		

Three residents, three projects. Each resident reports a value (thousand TL) for each project. Only **one** project will be built.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	—	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without i	—	—	—		

(a) Best choice with everyone: the sum of the three values (social welfare) is Park 50, Library 55, Gym 65. The highest is the Gym: **x* = Gym, welfare 65**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	—	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Ayşe	10	50	65		

(b) Without Ayşe: Bora's and Cem's values sum to Park 10, Library 50, Gym 65. Best is the Gym; the others' welfare is **65**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	—	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Ayşe	10	50	65		

(c) With Ayşe the Gym is chosen. The others' welfare at the Gym: Bora 20 + Cem 45 = **65**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Ayşe	10	50	65		

(d) Payment = (b) − (c) = 65 − 65 = 0. The Gym is chosen with or without Ayşe; she does no harm to the others.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Bora	50	25	45		

(b) Without Bora: Ayşe's and Cem's values sum to Park 50, Library 25, Gym 45. Best is the Park; the others' welfare is **50**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	—	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Bora	50	25	45		

(c) With Bora the Gym is chosen. The others' welfare at the Gym: Ayşe 0 + Cem 45 = **45**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	**5**	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Bora	50	25	45		

(d) Payment = (b) − (c) = 50 − 45 = 5. Without Bora the Park would be chosen and the others would get 50; because of Bora the Gym is chosen and they get 45. Bora's harm to the others is 5.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	**5**	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Cem	40	35	20		

(b) Without Cem: Ayşe's and Bora's values sum to Park 40, Library 35, Gym 20. Best is the Park; the others' welfare is **40**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	**5**	—
Cem	10	20	45	—	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Cem	40	35	20		

(c) With Cem the Gym is chosen. The others' welfare at the Gym: Ayşe 0 + Bora 20 = **20**.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	—
Bora	0	30	20	**5**	—
Cem	10	20	45	**20**	—
(a) Σ everyone	50	55	65 ✓		
(b) Σ without Cem	40	35	20		

(d) Payment = (b) − (c) = 40 − 20 = 20. Without Cem the Park would be chosen and the others would get 40; because of Cem the Gym is chosen and they get 20. Cem's harm to the others is 20.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Step by Step: Choosing a Public Project with VCG

	Park	Library	Gym	Payment p_i	Utility u_i
Ayşe	40	5	0	**0**	**0**
Bora	0	30	20	**5**	**15**
Cem	10	20	45	**20**	**25**
(a) Σ everyone	50	55	65 ✓	25	40
(b) Σ without i	—	—	—		

Utilities (value − payment): Ayşe 0 − 0 = 0, Bora 20 − 5 = 15, Cem 45 − 20 = 25. Everyone's utility is ≥ 0 (IR). The 25 collected is not returned to anyone: a budget surplus.

Payment p_i = (b) − (c) = (others' best welfare without i) − (others' welfare with i): **the harm i does to others**.

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	—	—	—
w(a, D)	—	—	—

Build the player list. `v[i][a]` : i's reported value for project a.

Step 1 / 16

```
1  def vcg(v, A):
▶  N = list(v)              # players
3    def w(a, S):           # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects`

`N = [Ayşe, Bora, Cem]`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D)	—	—	—

Compute `w(a, N)` for each project; the largest is the Gym (65) → **x = G**.

Step 2 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
▶    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects`

`N = [Ayşe, Bora, Cem]` `x = G`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D)	—	—	—

The payment dict starts empty; now ask “what if they were absent?” for each player.

Step 3 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
▶    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G` `p = {}`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Ayşe	10	50	65

i = Ayşe: D holds everyone except Ayşe.

Step 4 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G` `p = {}` `i = Ayşe` `D = [Bora, Cem]`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Ayşe	10	50	65

Without Ayşe, the best project is the Gym: **w_out = 65**.

Step 5 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G` `p = {}` `i = Ayşe` `D = [Bora, Cem]` `w_out = 65`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Ayşe	10	50	65

With Ayşe the Gym is chosen; D's welfare there: **w_in = 65**.

Step 6 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G` `p = {}` `i = Ayşe` `D = [Bora, Cem]` `w_out = 65` `w_in = 65`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Ayşe	10	50	65

p[Ayşe] = 65 − 65 = 0.

Step 7 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0}` `i = Ayşe`

`D = [Bora, Cem]` `w_out = 65` `w_in = 65`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Bora	50	25	45

i = Bora: D holds everyone except Bora.

Step 8 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0}` `i = Bora`

`D = [Ayşe, Cem]`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Bora	50	25	45

Without Bora, the best project is the Park: **w_out = 50**.

Step 9 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G` `p = {Ayşe: 0}` `i = Bora` `D = [Ayşe, Cem]` `w_out = 50`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Bora	50	25	45

With Bora the Gym is chosen; D's welfare there: **w_in = 45**.

Step 10 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                  # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0}` `i = Bora`

`D = [Ayşe, Cem]` `w_out = 50` `w_in = 45`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Bora	50	25	45

p[Bora] = 50 − 45 = 5.

Step 11 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5}` `i = Bora`

`D = [Ayşe, Cem]` `w_out = 50` `w_in = 45`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Cem	40	35	20

i = Cem: D holds everyone except Cem.


```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5}` `i = Cem`

`D = [Ayşe, Bora]`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Cem	40	35	20

Without Cem, the best project is the Park: **w_out = 40**.

Step 13 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5}` `i = Cem`

`D = [Ayşe, Bora]` `w_out = 40`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Cem	40	35	20

With Cem the Gym is chosen; D's welfare there: **w_in = 20**.


```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5}` `i = Cem`

`D = [Ayşe, Bora]` `w_out = 40` `w_in = 20`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D) · D = N − Cem	40	35	20

p[Cem] = 40 − 20 = 20.

Step 15 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5, Cem: 20}` `i = Cem`

`D = [Ayşe, Bora]` `w_out = 40` `w_in = 20`

Code Trace: `vcg(v, A)`

	Park	Library	Gym
w(a, N)	50	55	65
w(a, D)	—	—	—

Result: the Gym is built; payments {Ayşe: 0, Bora: 5, Cem: 20}. Total 25.

Step 16 / 16

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
11     p[i] = w_out - w_in
▶    return x, p
```

VARIABLES

`v = 3 players × 3 projects` `x = G`

`p = {Ayşe: 0, Bora: 5, Cem: 20}`

Does Lying Pay in VCG?

Case	Report (P, L, G)	Chosen	Payment	True utility
Cem truthful	10, 20, 45	Gym	20	**45 − 20 = 25**

Baseline: Cem reports truthfully → the Gym is chosen, he pays 20, true utility 45 − 20 = **25**.

Step 1 / 4

Trying every report (0, 10, …, 100 for each project), no player ever beats the utility of truthful reporting.

Does Lying Pay in VCG?

Case	Report (P, L, G)	Chosen	Payment	True utility
Cem truthful	10, 20, 45	Gym	20	**45 − 20 = 25**
Cem overstates	10, 20, 100	Gym	20	**45 − 20 = 25**

Cem reports 100 for the Gym: the outcome is unchanged and so is the payment (it depends on the others' reports) → still 25.

Trying every report (0, 10, …, 100 for each project), no player ever beats the utility of truthful reporting.

Does Lying Pay in VCG?

Case	Report (P, L, G)	Chosen	Payment	True utility
Cem truthful	10, 20, 45	Gym	20	**45 − 20 = 25**
Cem overstates	10, 20, 100	Gym	20	**45 − 20 = 25**
Cem understates	10, 20, 25	Library	5	**20 − 5 = 15**

Cem reports 25 for the Gym: totals 50, 55, 45 → the Library is chosen. Cem pays 5, but his true value there is 20 → utility **15 < 25**.

Trying every report (0, 10, …, 100 for each project), no player ever beats the utility of truthful reporting.

Does Lying Pay in VCG?

Case	Report (P, L, G)	Chosen	Payment	True utility
Cem truthful	10, 20, 45	Gym	20	**45 − 20 = 25**
Cem overstates	10, 20, 100	Gym	20	**45 − 20 = 25**
Cem understates	10, 20, 25	Library	5	**20 − 5 = 15**
Bora overstates	0, 60, 20	Library	25	**30 − 25 = 5**

Bora reports 60 for the Library: the Library is chosen, but Bora pays 25 → 30 − 25 = **5 < 15**. No lie pays off.

Trying every report (0, 10, …, 100 for each project), no player ever beats the utility of truthful reporting.

Properties and Limits of VCG

Strengths

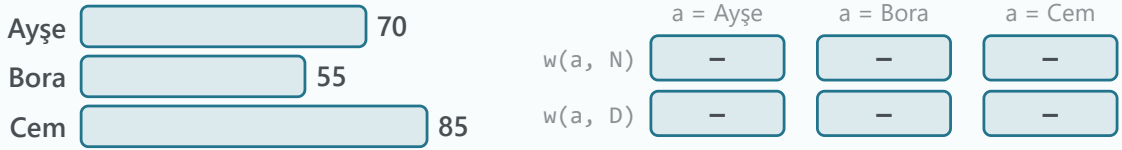
- \+ DSIC: truthful reporting is a dominant strategy.
- \+ Efficient: the welfare-maximizing outcome is chosen.
- \+ IR with the Clarke pivot and nonnegative values; payments ≥ 0.
- \+ For a single item, VCG = the Vickrey (second-price) auction.

Limits

- – No strong budget balance: the example leaves a surplus of 25 (never a deficit; weak budget balance holds).
- – Computing the argmax can be NP-hard in combinatorial problems.
- – Vulnerable to collusion and fake identities; revenue can be low.
- – Asking players to reveal all their values raises privacy concerns.

High computational cost and budget-balance issues can limit real deployments. Sponsored search ads mostly use GSP instead of VCG.

Code Trace: `vcg()` on a Single Item = Vickrey



Single item: outcome `a` = who gets the item. `v[i][a]` = i's value if a = i, else 0.

Step 1 / 13

```
1  def vcg(v, A):
▶  N = list(v)              # players
3    def w(a, S):           # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item`

`N = [Ayşe, Bora, Cem]`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	–	–	–

w(a, N) is just the recipient's value: 70, 55, 85 → **x = Cem** (highest value).

Step 2 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
▶    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item`

`N = [Ayşe, Bora, Cem]` `x = Cem`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	–	–	–

For payments, each bidder is asked “what if you weren't here?”

Step 3 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
▶    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem` `p = {}`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	0	55	85

i = **Ayşe**: Ayşe's row is grayed out; D = the others.

Step 4 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem` `p = {}`

`i = Ayşe` `D = [Bora, Cem]`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	0	55	85

Without Ayşe the item would go to Cem: **w_out = 85**.

Step 5 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem` `p = {}`
`i = Ayşe` `D = [Bora, Cem]` `w_out = 85`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	0	55	85

The item is with Cem; D's welfare there is w_in = 85. p[Ayşe] = 85 − 85 = **0** → losers pay nothing.


```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                   # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0}` `i = Ayşe`

`D = [Bora, Cem]` `w_out = 85` `w_in = 85`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	0	85

i = **Bora**: Bora's row is grayed out; D = the others.

Step 7 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem` `p = {Ayşe: 0}` `i = Bora` `D = [Ayşe, Cem]`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	0	85

Without Bora the item would go to Cem: **w_out = 85**.

Step 8 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0}` `i = Bora`

`D = [Ayşe, Cem]` `w_out = 85`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	0	85

The item is with Cem; D's welfare there is w_in = 85. p[Bora] = 85 − 85 = **0** → losers pay nothing.

Step 9 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                  # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`
`p = {Ayşe: 0, Bora: 0}` `i = Bora`
`D = [Ayşe, Cem]` `w_out = 85` `w_in = 85`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	55	0

i = **Cem**: Cem's row is grayed out; D = the others.

Step 10 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
▶    for i in N:
▶      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0, Bora: 0}` `i = Cem`

`D = [Ayşe, Bora]`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	55	0

Without Cem the item would go to Ayşe: **w_out = 70**.

Step 11 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
▶      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                   # i present
11     p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0, Bora: 0}` `i = Cem`

`D = [Ayşe, Bora]` `w_out = 70`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	70	55	0

The item is with Cem; D's welfare there is w_in = 0. p[Cem] = 70 − 0 = **70** = the second-highest value!

Step 12 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
▶      w_in = w(x, D)                  # i present
▶      p[i] = w_out - w_in
12   return x, p
```

VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0, Bora: 0, Cem: 70}` `i = Cem`

`D = [Ayşe, Bora]` `w_out = 70` `w_in = 0`

Code Trace: `vcg()` on a Single Item = Vickrey



	a = Ayşe	a = Bora	a = Cem
w(a, N)	70	55	85
w(a, D)	–	–	–

Result: Cem wins and pays **70**; the losers pay 0. For a single item, VCG = Vickrey (second price).

Step 13 / 13

```
1  def vcg(v, A):
2    N = list(v)              # players
3    def w(a, S):             # welfare of S at a
4      return sum(v[i][a] for i in S)
5    x = max(A, key=lambda a: w(a, N))
6    p = {}
7    for i in N:
8      D = [j for j in N if j != i]
9      w_out = max(w(a, D) for a in A)  # i absent
10     w_in = w(x, D)                  # i present
11     p[i] = w_out - w_in
▶    return x, p
```

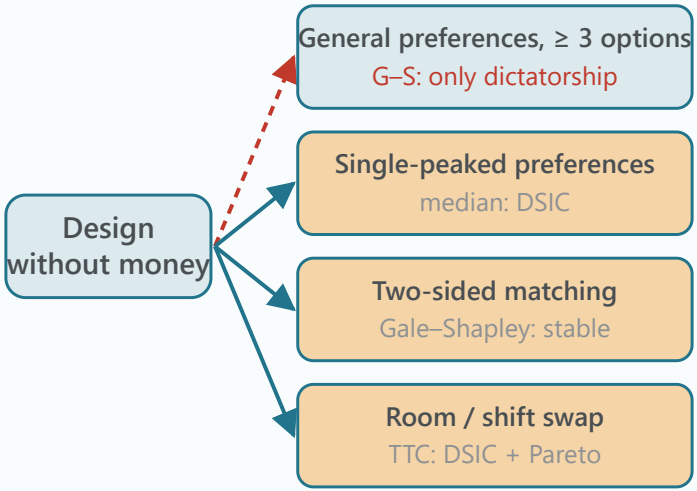
VARIABLES

`v = 3 bidders, 1 item` `x = Cem`

`p = {Ayşe: 0, Bora: 0, Cem: 70}`

Mechanism Design Without Money

- School admissions, dorm room swaps, voting: payments are either banned or inappropriate.
- Without money, incentives cannot be fixed through payments; only the **outcome rule** can be designed.
- Bad news for general preferences: with at least three alternatives, every onto, strategyproof rule (lying never helps) is a dictatorship — the same single person's preference always decides (Gibbard–Satterthwaite).
- **Ways out**: exploit preference structure (single-peaked → median) or change the goal (**stable** instead of "best" → Gale–Shapley; exchange → TTC).

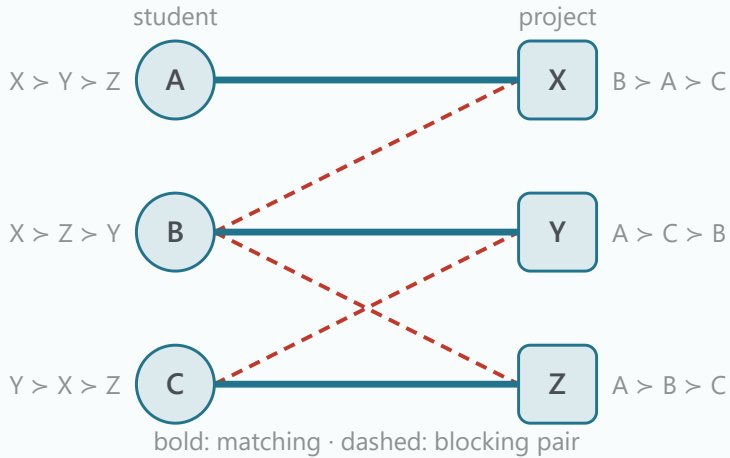


The Stable Matching Problem

- Two sides: students and capstone projects (each project takes one student). Each side ranks the other.
- A **blocking pair**: a student and a project who prefer each other to their current partners.
- A matching with no blocking pair is **stable**: nobody wants to break away.

Student	1.	2.	3.
A	X	Y	Z
B	X	Z	Y
C	Y	X	Z

Project	1.	2.	3.
X	B	A	C
Y	A	C	B
Z	A	B	C



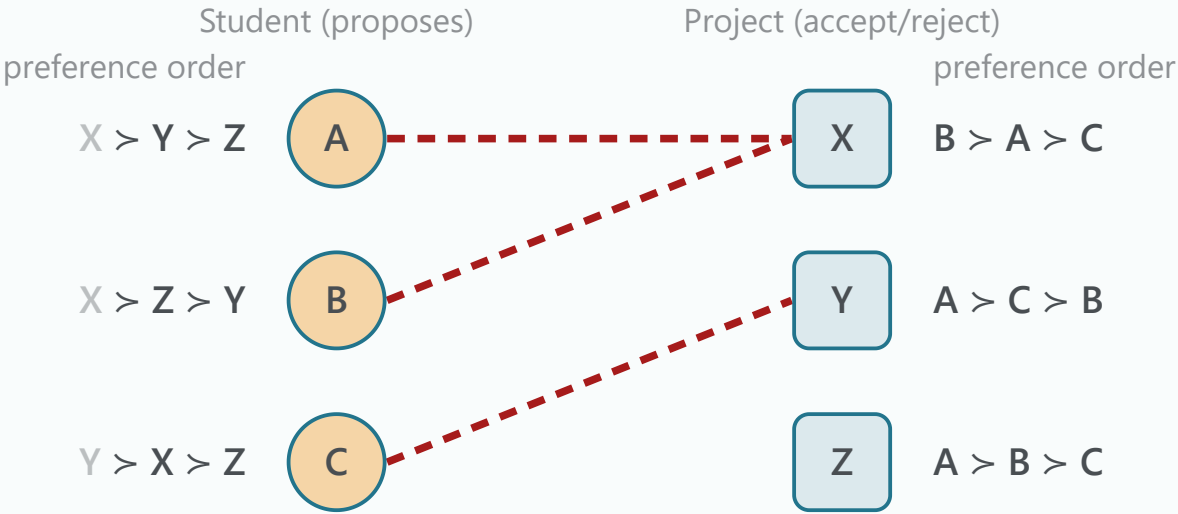
Unstable example {A–X, B–Y, C–Z}: B prefers X to Y, and X prefers B to A → **(B, X) blocks**. By the same logic (B, Z) and (C, Y) block too (checking all 9 pairs finds no others).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



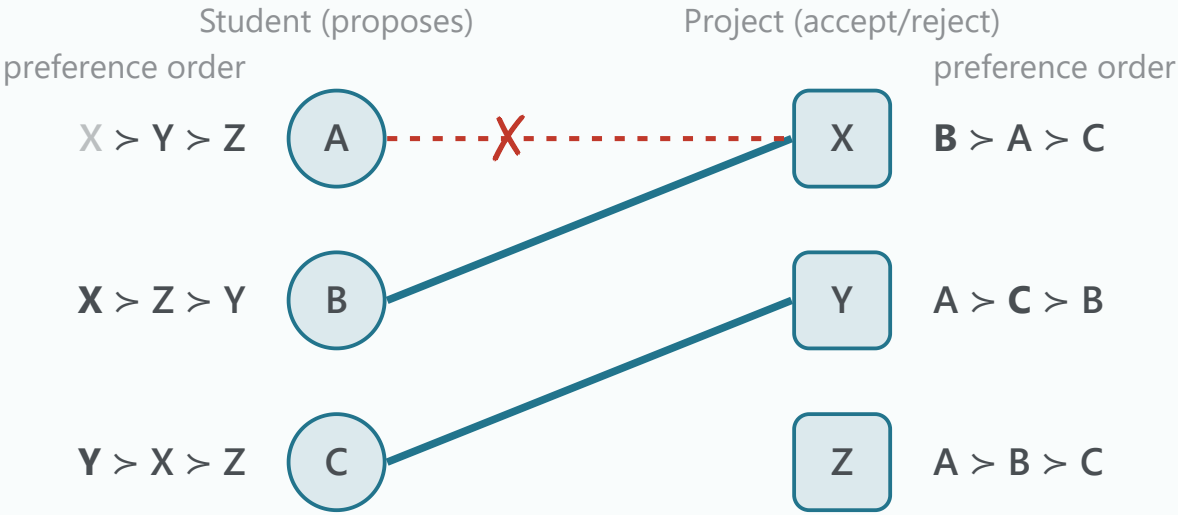
Start: everyone is free. Students propose in preference order; a project **tentatively** holds its best applicant and rejects the rest.

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



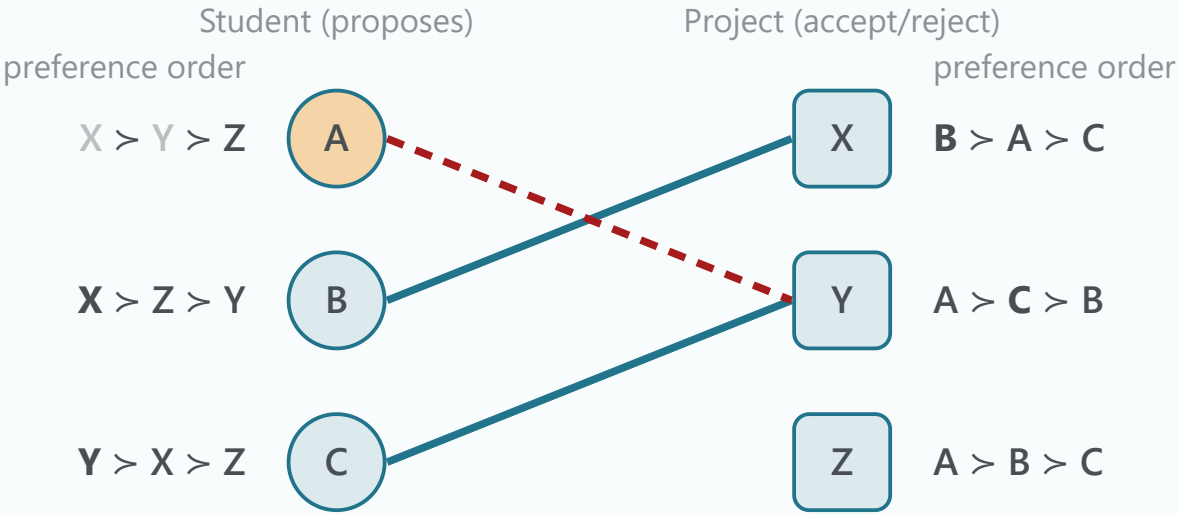
Round 1 proposals: A→X, B→X, C→Y (each free student to their best project that has not yet rejected them).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



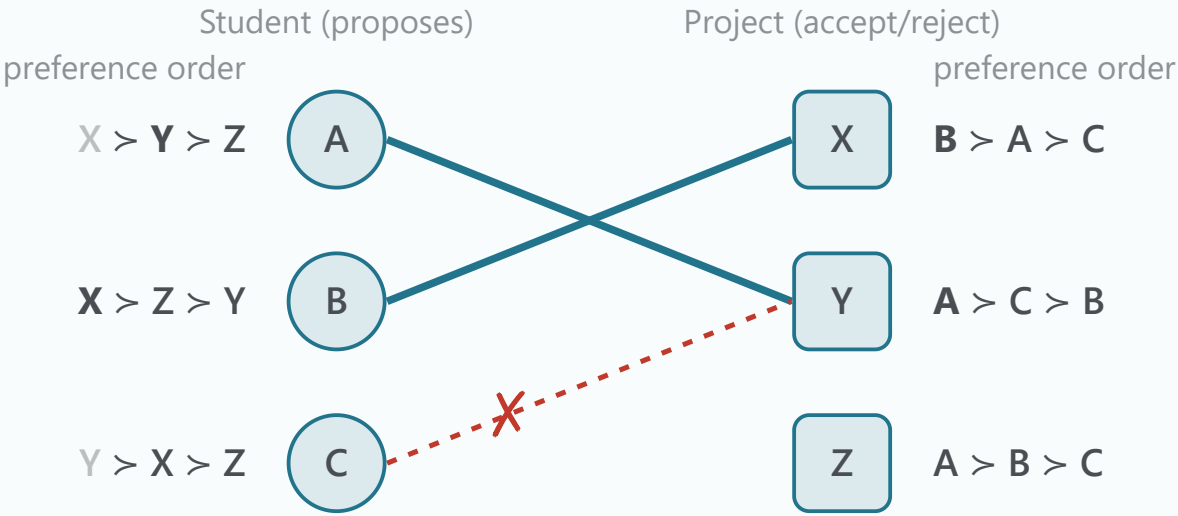
Round 1 decisions — X: holds B, rejects A (B, ranked 1st, is better); Y: accepts C (tentatively).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



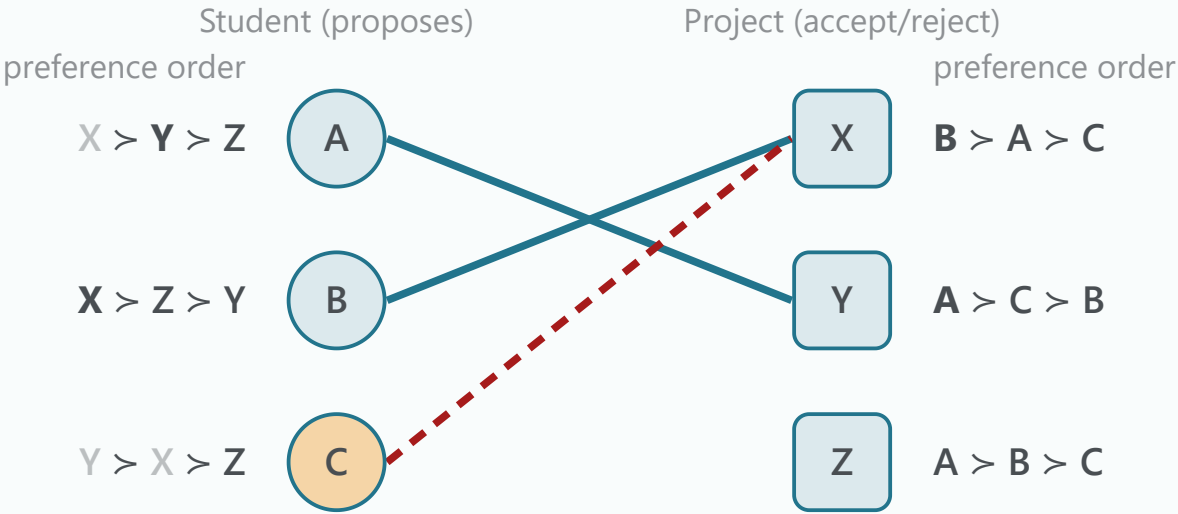
Round 2 proposals: A→Y (each free student to their best project that has not yet rejected them).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



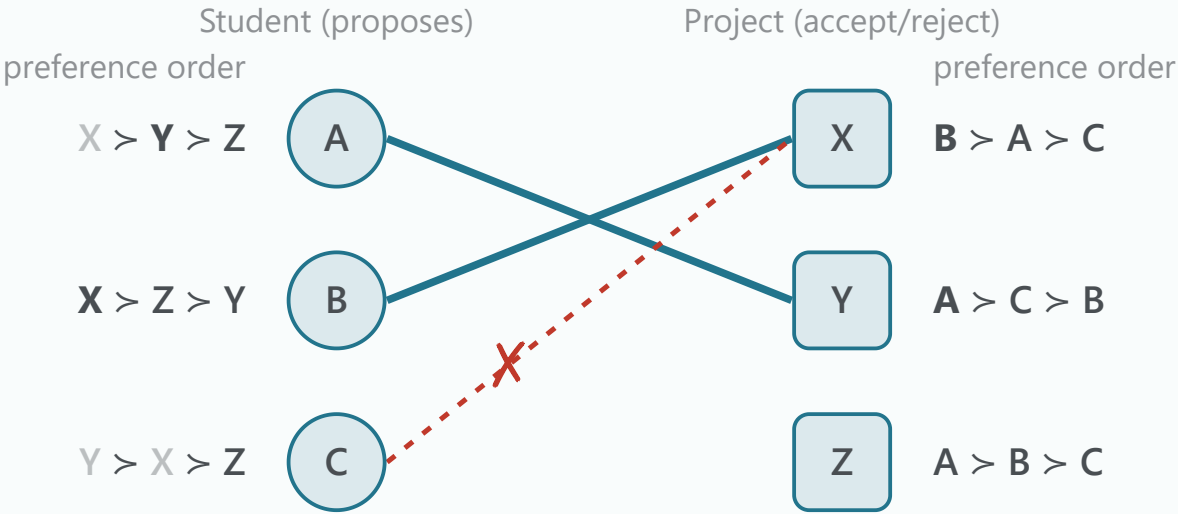
Round 2 decisions — Y: holds A, rejects C (A, ranked 1st, is better).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



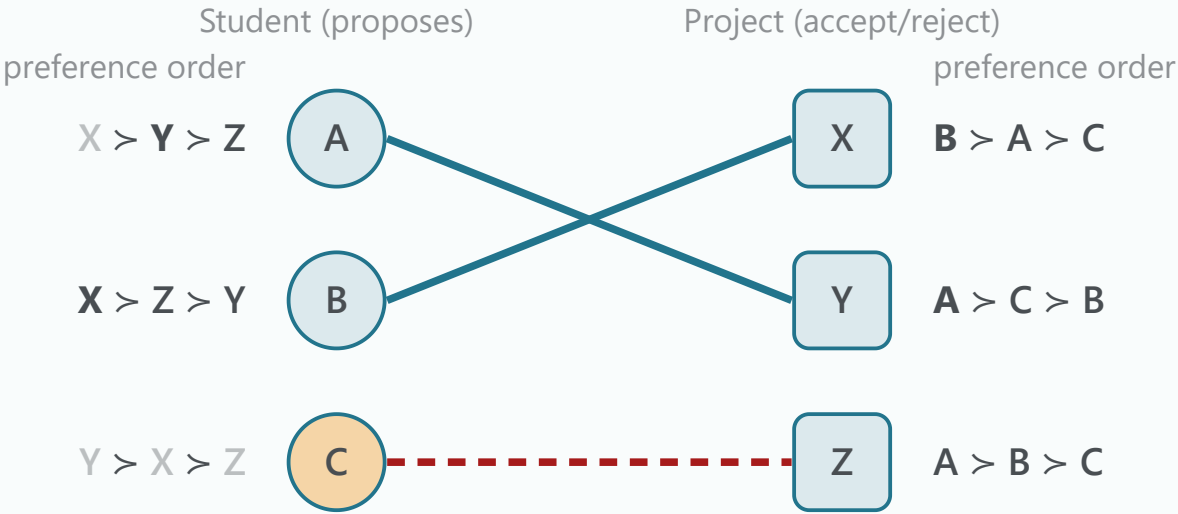
Round 3 proposals: C→X (each free student to their best project that has not yet rejected them).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



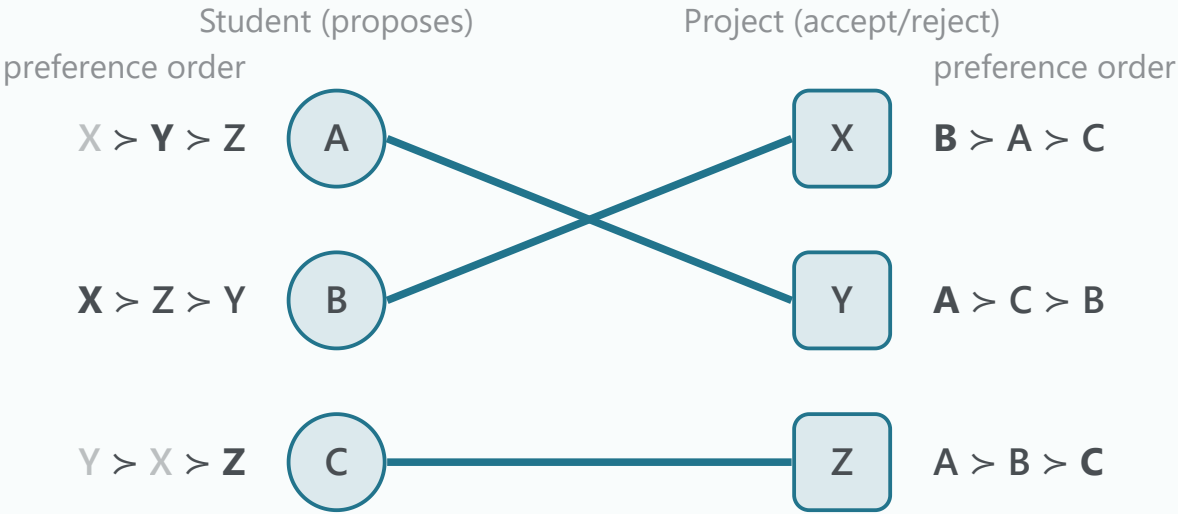
Round 3 decisions — X: holds B, rejects C (B, ranked 1st, is better).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



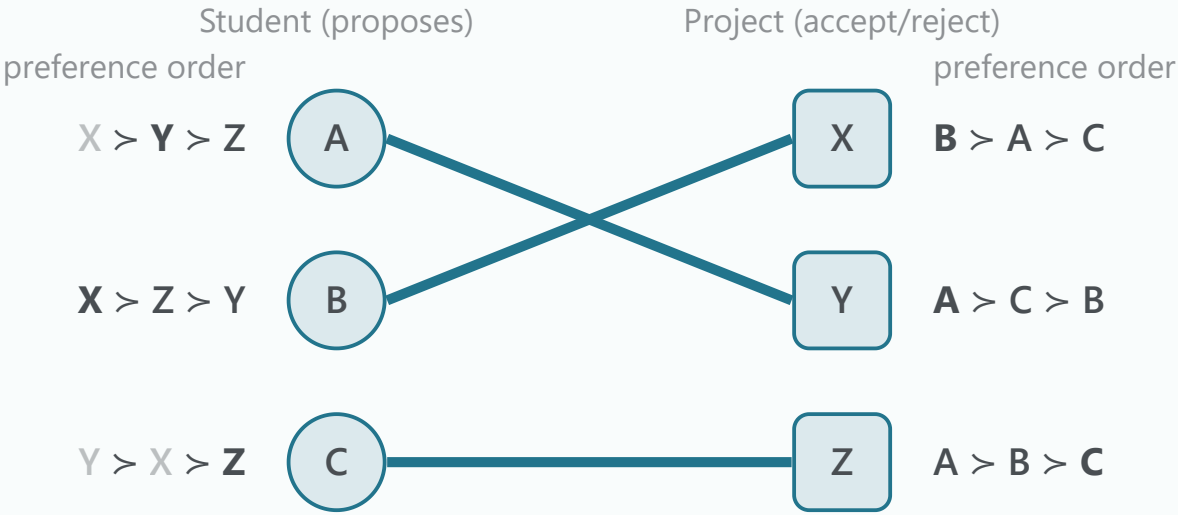
Round 4 proposals: C→Z (each free student to their best project that has not yet rejected them).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



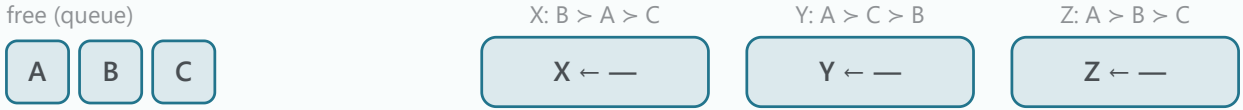
Round 4 decisions — Z: accepts C (tentatively).

Gale–Shapley: Deferred Acceptance (Proposal Rounds)



Everyone is matched → stop. **A–Y, B–X, C–Z**. Check: A wants X, but X prefers B to A; neither X nor Y wants C → no blocking pair, **stable**.

Code Trace: `gale_shapley(stu, proj)`



Everyone free, projects empty. `proj[p][s]` : p's rank for s (0 = best).

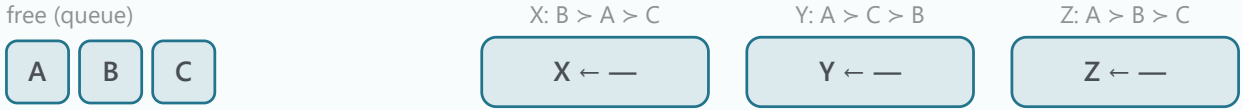
Step 1 / 15

```
1  def gale_shapley(stu, proj):
▶    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = [A, B, C]` `match = {}`

Code Trace: `gale_shapley(stu, proj)`



Each student's next choice is index 0 (their favorite project).

Step 2 / 15

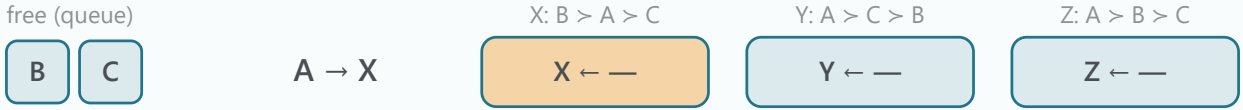
```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
▶    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = [A, B, C]` `match = {}`

`nxt = {A: 0, B: 0, C: 0}`

Code Trace: `gale_shapley(stu, proj)`



Pop **A** from the queue; A proposes to its next choice, **X**.

Step 3 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

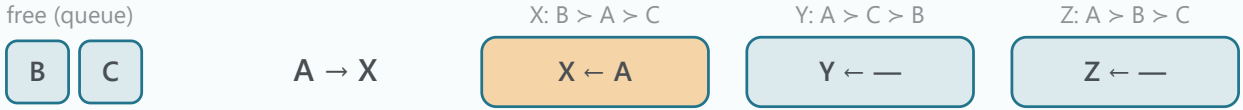
VARIABLES

`free = [B, C]` `match = {}`

`nxt = {A: 1, B: 0, C: 0}` `s = A`

`p = X`

Code Trace: `gale_shapley(stu, proj)`



X is empty (r = None) → A is **tentatively accepted**.

Step 4 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
▶        match[p] = s              # accept
▶        if r is not None: free.append(r)
12     else: free.append(s)       # reject
13   return match
```

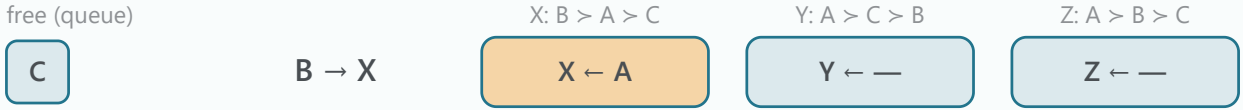
VARIABLES

`free = [B, C]` `match = {X: A}`

`nxt = {A: 1, B: 0, C: 0}` `s = A`

`p = X` `r = None`

Code Trace: `gale_shapley(stu, proj)`



Pop **B** from the queue; B proposes to its next choice, **X**.


```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = [C]` `match = {X: A}`

`nxt = {A: 1, B: 1, C: 0}` `s = B`

`p = X`

Code Trace: `gale_shapley(stu, proj)`

free (queue)	A rejected	X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
C, A	B → X	X ← B	Y ← —	Z ← —

X was holding A; X prefers B (rank 0 < 1) → B accepted, **A rejoins the queue**.

Step 6 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
▶        match[p] = s              # accept
▶        if r is not None: free.append(r)
12     else: free.append(s)       # reject
13   return match
```

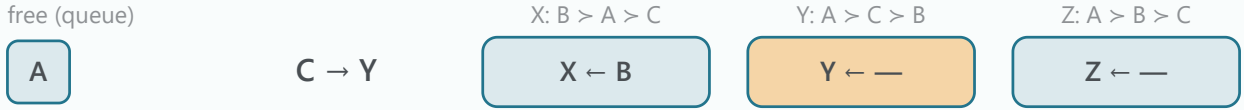
VARIABLES

`free = [C, A]` `match = {X: B}`

`nxt = {A: 1, B: 1, C: 0}` `s = B`

`p = X` `r = A`

Code Trace: `gale_shapley(stu, proj)`



Pop **C** from the queue; C proposes to its next choice, **Y**.

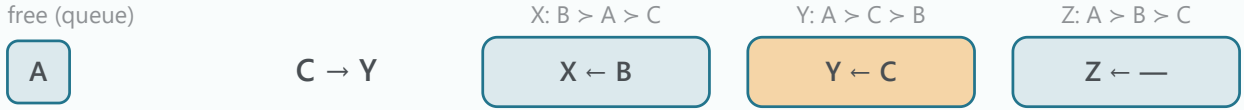
Step 7 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = [A]` `match = {X: B}` `nxt = {A: 1, B: 1, C: 1}` `s = C` `p = Y`

Code Trace: `gale_shapley(stu, proj)`



Y is empty (r = None) → C is **tentatively accepted**.

Step 8 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
▶        match[p] = s              # accept
▶        if r is not None: free.append(r)
12     else: free.append(s)       # reject
13   return match
```

VARIABLES

`free = [A]` `match = {X: B, Y: C}`

`nxt = {A: 1, B: 1, C: 1}` `s = C`

`p = Y` `r = None`

Code Trace: `gale_shapley(stu, proj)`

free (queue)		X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
empty	A → Y	X ← B	Y ← C	Z ← —

Pop **A** from the queue; A proposes to its next choice, **Y**.

Step 9 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = []` `match = {X: B, Y: C}` `nxt = {A: 2, B: 1, C: 1}` `s = A` `p = Y`

Code Trace: `gale_shapley(stu, proj)`

free (queue)	C rejected	X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
C	A → Y	X ← B	Y ← A	Z ← —

Y was holding C; Y prefers A (rank 0 < 1) → A accepted, **C rejoins the queue**.

Step 10 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
▶        match[p] = s              # accept
▶        if r is not None: free.append(r)
12     else: free.append(s)       # reject
13   return match
```

VARIABLES

`free = [C]` `match = {X: B, Y: A}`

`nxt = {A: 2, B: 1, C: 1}` `s = A`

`p = Y` `r = C`

Code Trace: `gale_shapley(stu, proj)`

free (queue)		X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
empty	C → X	X ← B	Y ← A	Z ← —

Pop **C** from the queue; C proposes to its next choice, **X**.

Step 11 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = []` `match = {X: B, Y: A}` `nxt = {A: 2, B: 1, C: 2}` `s = C` `p = X`

Code Trace: `gale_shapley(stu, proj)`

free (queue)	C rejected	X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
C	C → X	X ← B	Y ← A	Z ← —

X prefers the B it holds to C → **C is rejected** and rejoins the queue.

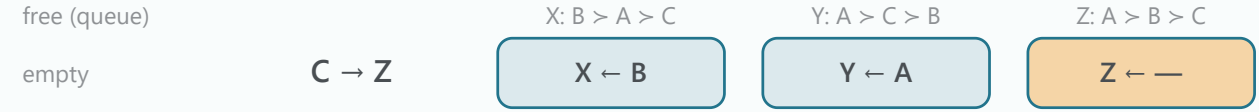
Step 12 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
▶      else: free.append(s)       # reject
13   return match
```

VARIABLES

`free = [C]` `match = {X: B, Y: A}` `nxt = {A: 2, B: 1, C: 2}` `s = C` `p = X` `r = B`

Code Trace: `gale_shapley(stu, proj)`



Pop **C** from the queue; C proposes to its next choice, **Z**.


```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
▶      s = free.pop(0)
▶      p = stu[s][nxt[s]]          # next choice
▶      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
13   return match
```

VARIABLES

`free = []` `match = {X: B, Y: A}` `nxt = {A: 2, B: 1, C: 3}` `s = C` `p = Z`

Code Trace: `gale_shapley(stu, proj)`

free (queue)		X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
empty	C → Z	X ← B	Y ← A	Z ← C

Z is empty (r = None) → C is **tentatively accepted**.

Step 14 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
▶      r = match.get(p)            # p's holder
▶      if r is None or proj[p][s] < proj[p][r]:
▶        match[p] = s              # accept
▶        if r is not None: free.append(r)
12     else: free.append(s)       # reject
13   return match
```

VARIABLES

`free = []`

`match = {X: B, Y: A, Z: C}`

`nxt = {A: 2, B: 1, C: 3}` `s = C`

`p = Z` `r = None`

Code Trace: `gale_shapley(stu, proj)`

free (queue)	X: B ≻ A ≻ C	Y: A ≻ C ≻ B	Z: A ≻ B ≻ C
empty	X ← B	Y ← A	Z ← C

The queue is empty after 6 proposals. Same result as the simultaneous rounds: A–Y, B–X, C–Z.

Step 15 / 15

```
1  def gale_shapley(stu, proj):
2    free, match = list(stu), {}
3    nxt = {s: 0 for s in stu}
4    while free:
5      s = free.pop(0)
6      p = stu[s][nxt[s]]          # next choice
7      nxt[s] += 1
8      r = match.get(p)            # p's holder
9      if r is None or proj[p][s] < proj[p][r]:
10       match[p] = s              # accept
11       if r is not None: free.append(r)
12     else: free.append(s)        # reject
▶    return match
```

VARIABLES

`free = []`

`match = {X: B, Y: A, Z: C}`

Properties of Gale–Shapley

Always stable

The algorithm terminates with a stable matching (Gale and Shapley, 1962). At most n^2 proposals → $O(n^2)$ (with a constant-time queue, e.g. `deque`).

Proposer-optimal

Proposers get their best partner among all stable matchings; the receiving side gets its worst.

DSIC for proposers

For the proposing side, reporting the true ranking is a dominant strategy. The receiving side can sometimes gain by lying.

Applications

The US medical residency match (NRMP), school choice in Boston and New York. 2012 Nobel: Roth and Shapley.

Our example has a single stable matching (checking all 3! = 6 matchings); even if the projects proposed, the result would be A–Y, B–X, C–Z.

The Receiving Side Can Gain by Lying



A new example. Students propose. The projects' true preferences: X: B ≻ A ≻ C; Y and Z: A ≻ B ≻ C.

Step 1 / 4

This example has two stable matchings (checking all 3! = 6): the student-optimal A–X, B–Y, C–Z and the project-optimal A–Y, B–X, C–Z. X's lie moves the outcome to the project-optimal one. No stable mechanism is DSIC for both sides (Roth, 1982).

The Receiving Side Can Gain by Lying



With truthful reports the proposals are A→X, B→Y, C→X, C→Y, C→Z → **A–X, B–Y, C–Z**. X gets A, its 2nd choice.

This example has two stable matchings (checking all 3! = 6): the student-optimal A–X, B–Y, C–Z and the project-optimal A–Y, B–X, C–Z. X's lie moves the outcome to the project-optimal one. No stable mechanism is DSIC for both sides (Roth, 1982).

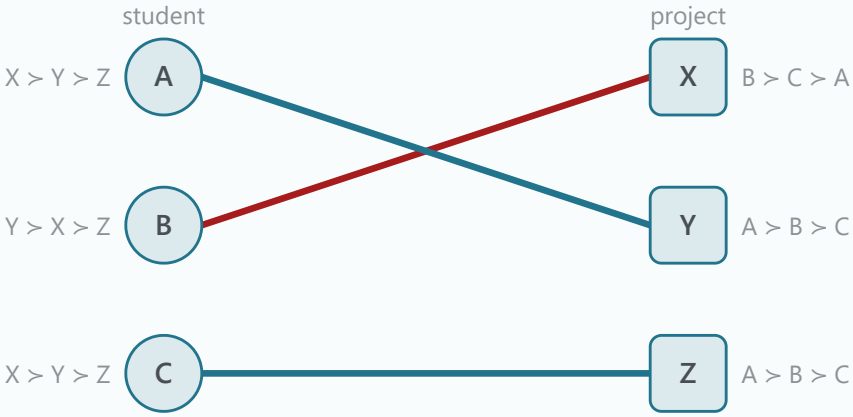
The Receiving Side Can Gain by Lying



X lies: it reports **B ≻ C ≻ A** (pushing A to the bottom).

This example has two stable matchings (checking all 3! = 6): the student-optimal A–X, B–Y, C–Z and the project-optimal A–Y, B–X, C–Z. X's lie moves the outcome to the project-optimal one. No stable mechanism is DSIC for both sides (Roth, 1982).

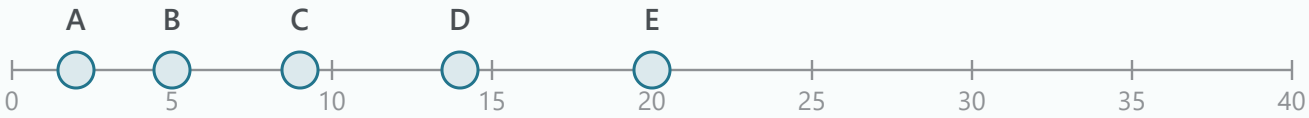
The Receiving Side Can Gain by Lying



Proposals A→X, B→Y, C→X, A→Y, B→X, C→Y, C→Z: rejecting A sets off a chain that brings B to X → **B–X**. X got its true first choice: the lie paid off.

This example has two stable matchings (checking all 3! = 6): the student-optimal A–X, B–Y, C–Z and the project-optimal A–Y, B–X, C–Z. X's lie moves the outcome to the project-optimal one. No stable mechanism is DSIC for both sides (Roth, 1982).

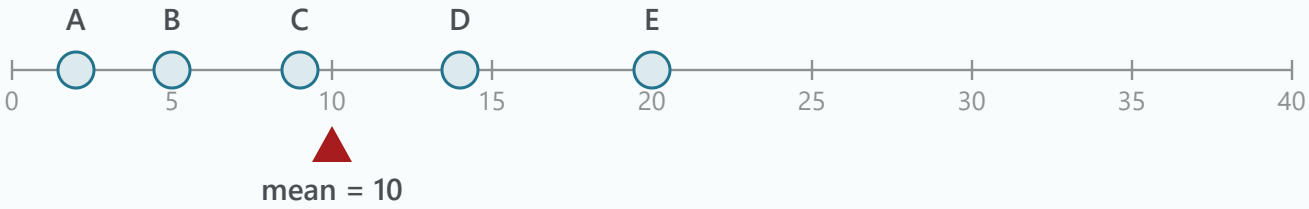
Facility Location: Mean or Median?



Case	Facility	E's distance

Five villages along a road: 2, 5, 9, 14, 20 km. One facility (a hospital) will be built; each village wants it **close to itself** (cost = distance).

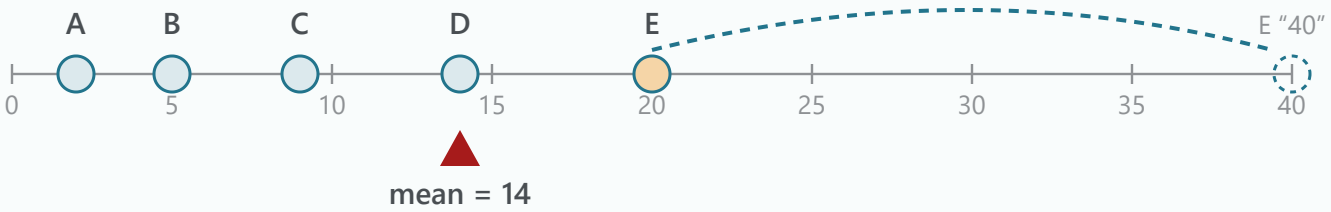
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**

The **mean** mechanism: (2+5+9+14+20)/5 = 10. E is 10 km away.

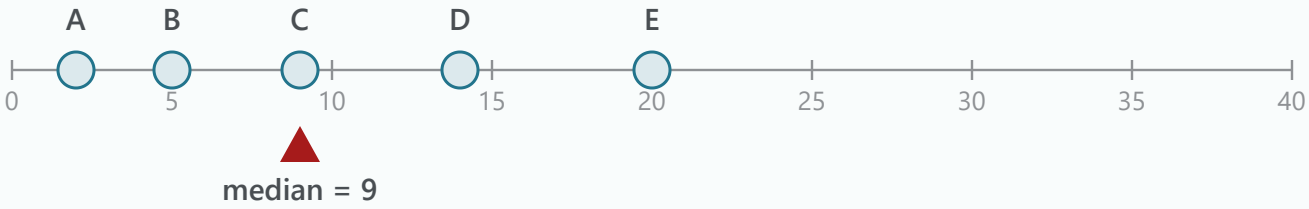
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**
Mean, E says “40”	14	**6**

E lies and reports 40: mean (2+5+9+14+40)/5 = 14. E's distance drops **10** → **6**. Lying paid off: the mean can be manipulated.

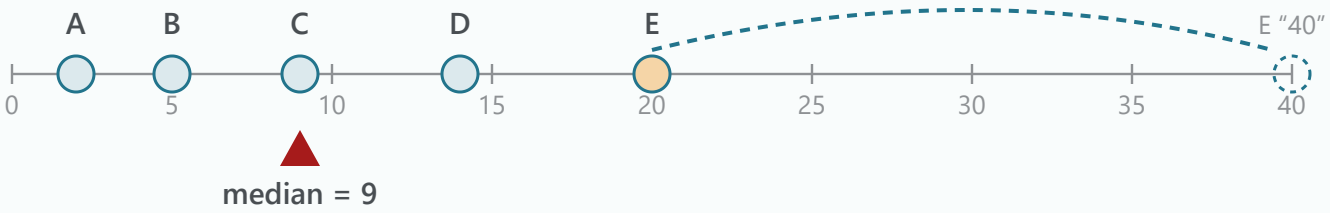
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**
Mean, E says “40”	14	**6**
Median, all truthful	9	**11**

The **median** mechanism: sort the reports and take the middle one → 9. E is 11 km away.

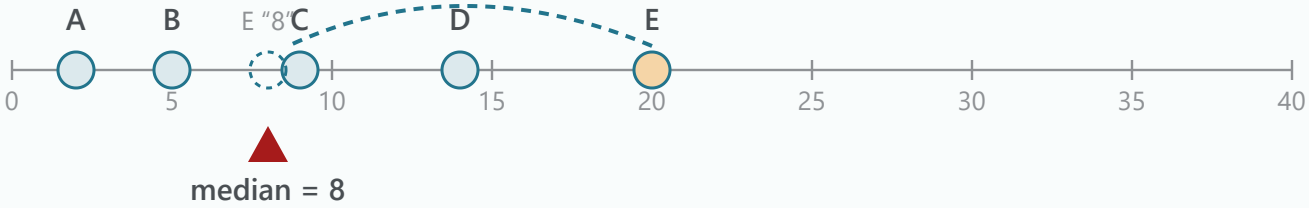
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**
Mean, E says "40"	14	**6**
Median, all truthful	9	**11**
Median, E says "40"	9	**11**

E reports 40: sorted 2, 5, 9, 14, 40 → median **still 9**. An extreme report cannot pull the median.

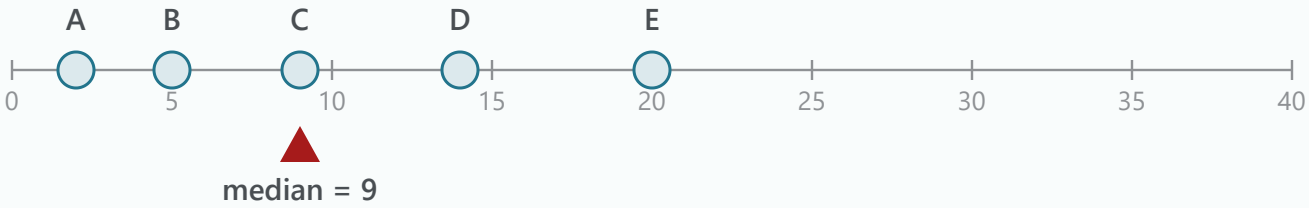
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**
Mean, E says "40"	14	**6**
Median, all truthful	9	**11**
Median, E says "40"	9	**11**
Median, E says "8"	8	**12**

E jumps to the other side of the median and reports 8: the median becomes 8 and E's distance grows **11** → **12**. Worse!

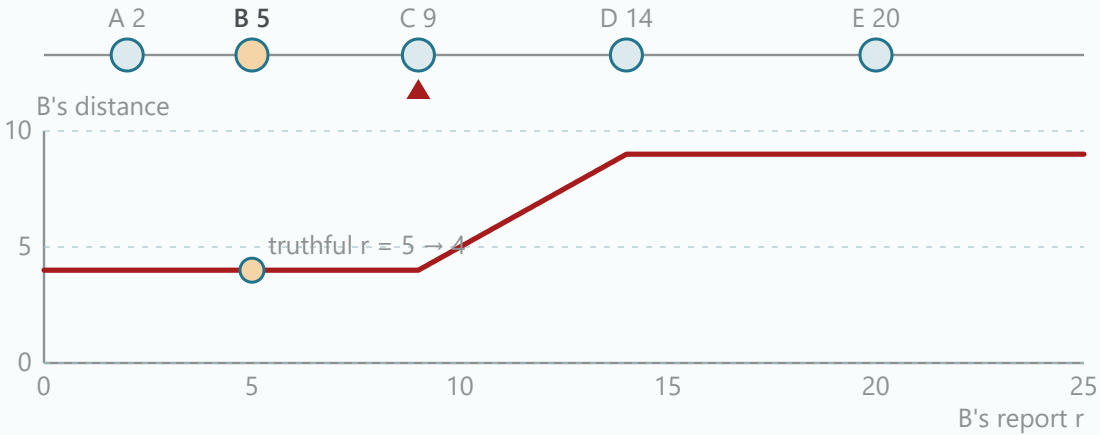
Facility Location: Mean or Median?



Case	Facility	E's distance
Mean, all truthful	10	**10**
Mean, E says “40”	14	**6**
Median, all truthful	9	**11**
Median, E says “40”	9	**11**
Median, E says “8”	8	**12**

Trying every report from 0 to 60 for every village, **no lie pays off** under the median. The median also gives total distance 27 versus 28 for the mean: it minimizes total distance too.

Why Is the Median Strategyproof?

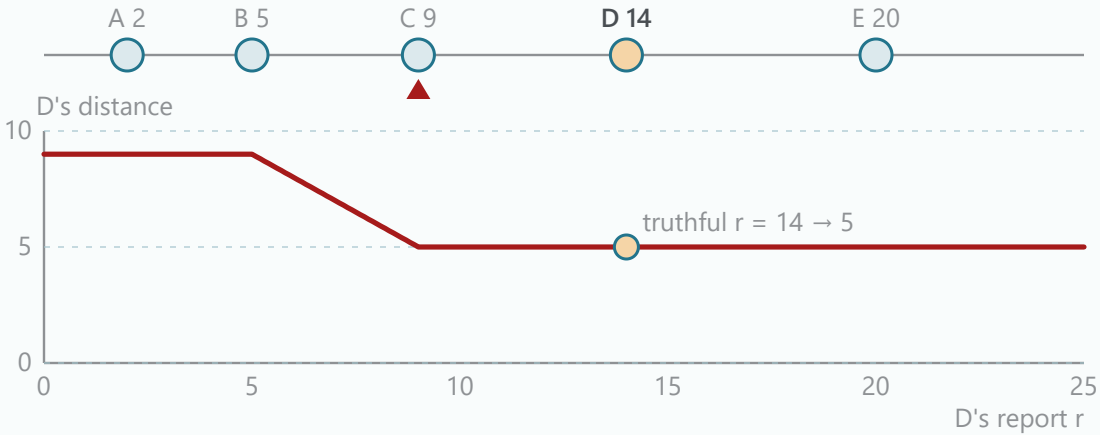


1) $x_i < m$ (B = 5, median 9): reports r ≤ 9 leave the median at 9 (distance 4); r > 9 pushes the median **right**, away from B. No r gets below 4.

Step 1 / 3

Conclusion: in every case truthful reporting is a dominant strategy → the median mechanism is **strategyproof** (Moulin, 1980). The same idea works for voting with single-peaked preferences (Black).

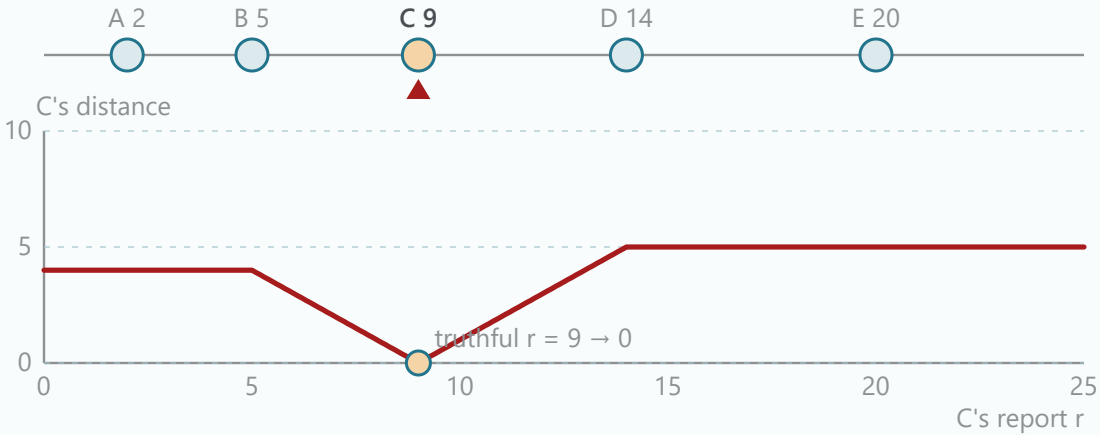
Why Is the Median Strategyproof?



2) $x_i > m$ (D = 14): the symmetric case. r ≥ 9 leaves the median unchanged (distance 5); r < 9 pulls it left, away from D.

Conclusion: in every case truthful reporting is a dominant strategy → the median mechanism is **strategyproof** (Moulin, 1980). The same idea works for voting with single-peaked preferences (Black).

Why Is the Median Strategyproof?



3) $x_i = m$ (C = 9): the facility sits exactly at C, distance 0; any lie can only move the median away.

Conclusion: in every case truthful reporting is a dominant strategy → the median mechanism is **strategyproof** (Moulin, 1980). The same idea works for voting with single-peaked preferences (Black).

Serial Dictatorship and the Boston Mechanism

Serial dictatorship

In turn, each takes their favorite of what remains. No one affects earlier picks → DSIC, Pareto optimal. Random order: **random serial dictatorship** (RSD).

Boston mechanism (immediate acceptance)

Each round, students apply to their next choice; schools fill free seats by priority, and seats are never taken back. Losing a first choice may mean the second is full.

Preferences A: X ≻ Y ≻ Z, B: X ≻ Y ≻ Z, C: Y ≻ X ≻ Z. Priorities at X and Z: A, B, C; at Y: B, A, C. One seat each.

Round	B truthful	B says "Y ≻ X ≻ Z"
1	X: A, B → A; Y: C → C	X: A → A; Y: B, C → B
2	B → Y full	C → X full
3	B → Z	C → Z
Outcome	A–X, **B–Z**, C–Y	A–X, **B–Y**, C–Z

Ranking Y first, B gets Y instead of Z: lying pays.

Truthful Gale–Shapley gives A–X, B–Y, C–Z here. Abdulkadiroğlu and Sönmez (2003) showed truthful ranking can hurt under Boston; in 2005 Boston switched to student-proposing deferred acceptance.

Dorm Room Exchange: Top Trading Cycles (TTC)

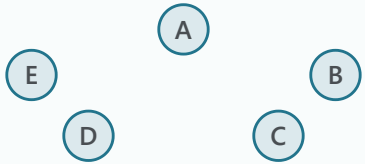
- Each student owns a dorm room, there is no money, and each ranks the others' rooms (Shapley and Scarf, 1974; the algorithm is credited to Gale).
- **TTC**: everyone points to the owner of their favorite *remaining* room. A finite graph has at least one **cycle**; those in it trade and leave. Repeat.
- Reporting true preferences is a dominant strategy (Roth, 1982); the outcome is Pareto optimal and in the core: no group can do better on its own (Roth and Postlewaite, 1977).
- The same rule fits duty or shift swaps; in the real world it is used in kidney exchange (Roth, Sönmez and Ünver, 2004).

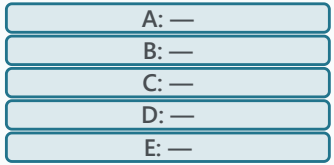
Person	Preference (room owner)
A	C ≻ A ≻ B
B	A ≻ D ≻ B
C	B ≻ A ≻ C
D	C ≻ E ≻ B ≻ D
E	A ≻ D ≻ E

Example: five students swapping dorm rooms. Rooms not listed are worse than the person's own room.

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E





Everyone is still in the list; nobody has traded yet.

Step 1 / 13

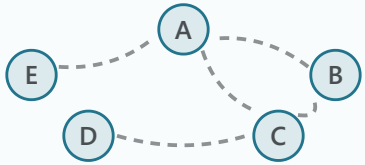
```
1  def ttc(pref):
▶    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

`left = [A, B, C, D, E]` `assign = {}`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A: —
B: —
C: —
D: —
E: —

Round 1: each remaining person points to the owner of their favorite *remaining* room (arrows).

Step 2 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
▶    while left:
▶      g = {i: next(j for j in pref[i]
▶               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

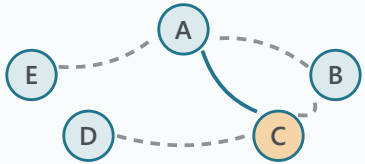
VARIABLES

`left = [A, B, C, D, E]` `assign = {}`

`g = {A→C, B→A, C→B, D→C, E→A}`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A: —
B: —
C: —
D: —
E: —

Add A to the path; its arrow points to C.

Step 3 / 13

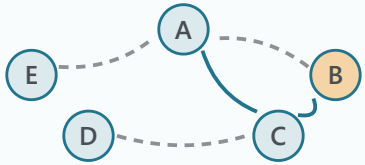
```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
▶        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

`left = [A, B, C, D, E]` `assign = {}`
`path = [A]` `i = C`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A: —
B: —
C: —
D: —
E: —

Add C to the path; its arrow points to B.

Step 4 / 13

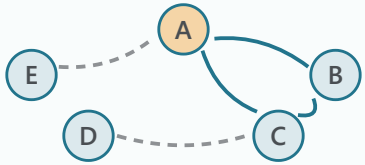
```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
▶        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

`left = [A, B, C, D, E]` `assign = {}`
`path = [A, C]` `i = B`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A: —
B: —
C: —
D: —
E: —

Add B to the path; its arrow points to A, already on the path → **cycle found**.

Step 5 / 13

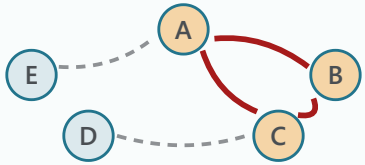
```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
▶        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

`left = [A, B, C, D, E]` `assign = {}`
`path = [A, C, B]` `i = A`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A: —
B: —
C: —
D: —
E: —

Cycle: **A** → **C** → **B** → **A**. Everyone on it gets the room they point to.

Step 6 / 13

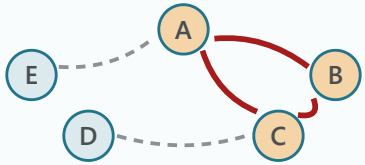
```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
▶      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

`left = [A, B, C, D, E]` `assign = {}`
`path = [A, C, B]` `cycle = [A, C, B]`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D: —
E: —

Trade: A ← C's room, C ← B's room, B ← A's room. They leave.

Step 7 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
▶      for k in cycle:               # trade
▶        assign[k] = g[k]; left.remove(k)
12   return assign
```

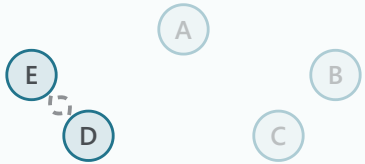
VARIABLES

`left = [D, E]`

`assign = {A→C, C→B, B→A}`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D: —
E: —

Round 2: each remaining person points to the owner of their favorite *remaining* room (arrows).

Step 8 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
▶    while left:
▶      g = {i: next(j for j in pref[i]
▶               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

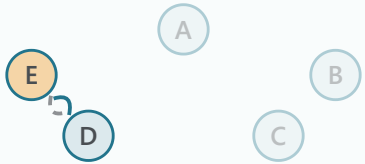
`left = [D, E]`

`assign = {A→C, C→B, B→A}`

`g = {D→E, E→D}`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D: —
E: —

Add D to the path; its arrow points to E.

Step 9 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
▶        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

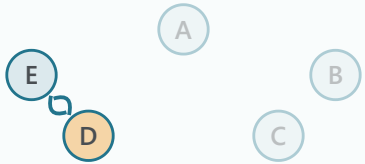
`left = [D, E]`

`assign = {A→C, C→B, B→A}` `path = [D]`

`i = E`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D: —
E: —

Add E to the path; its arrow points to D, already on the path → **cycle found**.

Step 10 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5               if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
▶        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

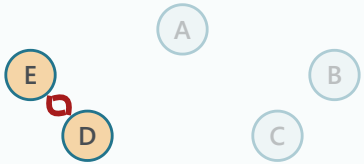
`left = [D, E]`

`assign = {A→C, C→B, B→A}`

`path = [D, E]` `i = D`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D: —
E: —

Cycle: **D** → **E** → **D**. Everyone on it gets the room they point to.

Step 11 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
▶      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
12   return assign
```

VARIABLES

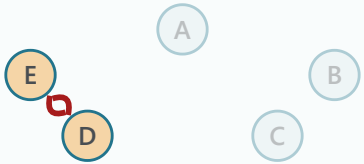
`left = [D, E]`

`assign = {A→C, C→B, B→A}`

`path = [D, E]` `cycle = [D, E]`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D ← E's room
E ← D's room

Trade: D ← E's room, E ← D's room. They leave.

Step 12 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
▶      for k in cycle:               # trade
▶        assign[k] = g[k]; left.remove(k)
12   return assign
```

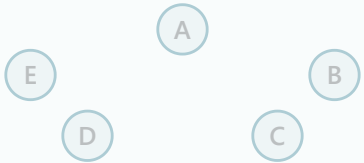
VARIABLES

`left = []`

`assign = {A→C, C→B, B→A, D→E, E→D}`

Code Trace: `ttc(pref)`

A: C ≻ A ≻ B
B: A ≻ D ≻ B
C: B ≻ A ≻ C
D: C ≻ E ≻ B ≻ D
E: A ≻ D ≻ E



A ← C's room
B ← A's room
C ← B's room
D ← E's room
E ← D's room

Done: A, B, C traded in a 3-cycle, D and E in a 2-cycle. Trying every false list for every person never yields a better room (DSIC); the outcome is Pareto optimal.

Step 13 / 13

```
1  def ttc(pref):
2    left, assign = list(pref), {}
3    while left:
4      g = {i: next(j for j in pref[i]
5                   if j in left) for i in left}
6      i, path = left[0], []
7      while i not in path:          # follow arrows
8        path.append(i); i = g[i]
9      cycle = path[path.index(i):]
10     for k in cycle:               # trade
11       assign[k] = g[k]; left.remove(k)
▶    return assign
```

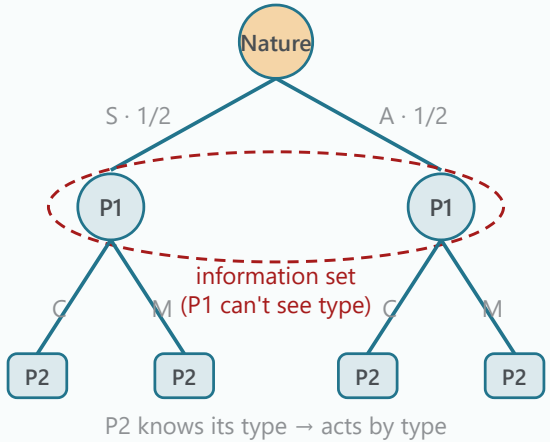
VARIABLES

`left = []`

`assign = {A→C, C→B, B→A, D→E, E→D}`

Incomplete Information: Bayesian Games

- So far the designer did not know the types; often the players do not know **each other's** types either.
- A Bayesian game: players, type sets Θ_i, a **common prior** π over types, payoffs $u_i(a, \theta)$.
- First, chance (Nature) draws each player's type; each player sees only their own (the Harsanyi transformation).
- A strategy is now a **rule**: which action to take for each type.
- **Bayes–Nash equilibrium**: every player, of every type, maximizes their *expected* payoff against the others' strategies.



Example game: P2's type is S (wants to meet) or A (wants to avoid).

Link to mechanisms: the Bayesian revelation principle → any outcome implementable in Bayes–Nash equilibrium can also be implemented by a BIC direct mechanism.

Step by Step: Finding a Bayes–Nash Equilibrium

Type S · probability 1/2	P2: C	P2: M
P1: C	(2, 1)	(0, 0)
P1: M	(0, 0)	(1, 2)

Type A · probability 1/2	P2: C	P2: M
P1: C	(2, 0)	(0, 2)
P1: M	(0, 1)	(1, 0)

Candidate profile	P1: C expected	P1: M expected	Equilibrium?

P1 picks a concert (C) or a match (M) and prefers C. P2 is either type **S** (wants to meet) or type **A** (wants to avoid); P1 does not know which, probabilities 1/2–1/2.

Step 1 / 5

Cells show (P1 payoff, P2 payoff). All 2 × 2 × 2 = 8 pure strategy profiles were checked.

Step by Step: Finding a Bayes–Nash Equilibrium

Type S · probability 1/2	P2: C	P2: M
P1: C	(2, 1)	(0, 0)
P1: M	(0, 0)	(1, 2)

Type A · probability 1/2	P2: C	P2: M
P1: C	(2, 0)	(0, 2)
P1: M	(0, 1)	(1, 0)

Candidate profile	P1: C expected	P1: M expected	Equilibrium?

Candidate: P1 = C. P2's best response depends on its type: type S picks C (1 > 0), type A picks M (2 > 0) → s_2 = (S→C, A→M).

Cells show (P1 payoff, P2 payoff). All 2 × 2 × 2 = 8 pure strategy profiles were checked.

Step by Step: Finding a Bayes–Nash Equilibrium

Type S · probability 1/2	P2: C	P2: M
P1: C	(2, 1)	(0, 0)
P1: M	(0, 0)	(1, 2)

Type A · probability 1/2	P2: C	P2: M
P1: C	(2, 0)	(0, 2)
P1: M	(0, 1)	(1, 0)

Candidate profile	P1: C expected	P1: M expected	Equilibrium?
P1 = C; S→C, A→M	½·2 + ½·0 = 1	½·0 + ½·1 = 0.5	✓ **yes**

P1's expected payoff: C → ½·2 + ½·0 = **1**; M → ½·0 + ½·1 = **0.5**. C is best → **(C; S→C, A→M) is a Bayes–Nash equilibrium**.

Cells show (P1 payoff, P2 payoff). All 2 × 2 × 2 = 8 pure strategy profiles were checked.

Step by Step: Finding a Bayes–Nash Equilibrium

Type S · probability 1/2	P2: C	P2: M
P1: C	(2, 1)	(0, 0)
P1: M	(0, 0)	(1, 2)

Type A · probability 1/2	P2: C	P2: M
P1: C	(2, 0)	(0, 2)
P1: M	(0, 1)	(1, 0)

Candidate profile	P1: C expected	P1: M expected	Equilibrium?
P1 = C; S→C, A→M	½·2 + ½·0 = 1	½·0 + ½·1 = 0.5	✓ yes

The other candidate: P1 = M. Then type S picks M (2 > 0) and type A picks C (1 > 0) → s_2 = (S→M, A→C).

Cells show (P1 payoff, P2 payoff). All 2 × 2 × 2 = 8 pure strategy profiles were checked.

Step by Step: Finding a Bayes–Nash Equilibrium

Type S · probability 1/2	P2: C	P2: M
P1: C	(2, 1)	(0, 0)
P1: M	(0, 0)	(1, 2)

Type A · probability 1/2	P2: C	P2: M
P1: C	(2, 0)	(0, 2)
P1: M	(0, 1)	(1, 0)

Candidate profile	P1: C expected	P1: M expected	Equilibrium?
P1 = C; S→C, A→M	½·2 + ½·0 = 1	½·0 + ½·1 = 0.5	✓ **yes**
P1 = M; S→M, A→C	½·0 + ½·2 = 1	½·1 + ½·0 = 0.5	✗ **deviates to C**

P1: C → ½·0 + ½·2 = 1; M → ½·1 + ½·0 = 0.5. P1 deviates from M to C → not an equilibrium. **Unique pure Bayes–Nash equilibrium: (C; S→C, A→M)**.

Cells show (P1 payoff, P2 payoff). All 2 × 2 × 2 = 8 pure strategy profiles were checked.

Mechanisms Through a Bayesian Lens

Example: first price

A truthful bid leaves zero utility when it wins; in equilibrium everyone bids below their value. This equilibrium is Bayesian: it rests on beliefs about the rivals' values.

DSIC or BIC?

DSIC is robust and independent of others' beliefs. BIC is weaker but allows more goals: e.g. an efficient, strongly budget-balanced public-good mechanism (d'Aspremont and Gérard-Varet, 1979; Arrow, 1979).

Maximum revenue

The mechanism maximizing the seller's expected revenue is also a Bayesian question; when values are i.i.d. from a regular distribution, the answer is a second-price auction with a reserve price (a minimum below which the seller won't sell) (Myerson, 1981).

Poker and bluffing

The opponent's hand (type) is hidden; bluffing is part of a mixed Bayesian strategy.

ASSESSMENT

Discussion: Behavior, Fairness and Ethics

Do people always respond to incentives?

- Theory: players maximize utility.
- In reality, emotions, morals, habit and a sense of fairness matter too (e.g. low offers are rejected in the Ultimatum Game).
- Incentives are powerful but not the only driver; simple, understandable rules matter.

Ethical issues in mechanism design

- Manipulation and withholding information; powerful players gaining an advantage.
- Unfair distribution of income; excluding those who cannot pay.
- Ethical limits should be set by society and institutions: a good mechanism must be fair and ethical, not just efficient.

ASSESSMENT

Summary: This Chapter's Mechanisms

Mechanism	Money	Incentive property	What it achieves
Proportional cost sharing	yes	Not IC (free riding)	—
VCG (Clarke pivot)	yes	DSIC, IR	Efficient outcome; may run a surplus
Vickrey (single-item VCG)	yes	DSIC, IR	Highest value wins
Gale–Shapley	no	DSIC for proposers	Stable matching, $O(n^2)$
Median (facility location)	no	DSIC (strategyproof)	Minimum total distance
TTC (house exchange)	no	DSIC	Pareto optimal, in the core
Bayesian mechanisms	yes	BIC (Bayes–Nash)	Expected goals (e.g. revenue)

Thanks to the revelation principle, it suffices to study direct, truthful mechanisms; but IC + IR + efficiency + budget balance cannot always be achieved together.

ASSESSMENT

Sample Problems

1. Two identical items, three bidders (each wants one), values 10, 8, 5. Under VCG, who wins and who pays what?
2. In the cost-sharing example (60; 30, 25, 20), if the others are truthful, what is Bora's most profitable report?
3. Villages at 3, 7, 8, 12, 30: median and mean? Under the mean, what should the village at 30 report to pull the facility to itself?
4. In the Gale–Shapley example, what would happen if the projects proposed?
5. In the Bayesian example, if type S had probability 1/4, would (C; S→C, A→M) still be an equilibrium?
6. TTC with preferences A: B ≻ C ≻ A, B: A ≻ C ≻ B, C: A ≻ B ≻ C. Who gets which room?

▶ **Show answers**

ASSESSMENT

References

- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007.
- T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, Cambridge University Press, 2016.
- Y. Shoham, K. Leyton-Brown, *Multiagent Systems*, Cambridge University Press, 2009.
- D. Gale, L. S. Shapley, "College Admissions and the Stability of Marriage", *American Mathematical Monthly*, 1962.
- H. Moulin, "On Strategy-Proofness and Single Peakedness", *Public Choice*, 1980 · M. J. Osborne, *An Introduction to Game Theory*, Oxford University Press, 2004 (Bayesian games).
- R. Myerson, "Optimal Auction Design", *Mathematics of Operations Research*, 1981 · L. Shapley, H. Scarf, "On Cores and Indivisibility", *Journal of Mathematical Economics*, 1974.
- A. E. Roth, T. Sönmez, M. U. Ünver, "Kidney Exchange", *Quarterly Journal of Economics*, 2004.
- A. E. Roth, A. Postlewaite, "Weak versus Strong Domination in a Market with Indivisible Goods", *J. Math. Econ.* 4(2), 1977 · C. d'Aspremont, L.-A. Gérard-Varet, "Incentives and Incomplete Information", *J. Public Econ.* 11(1), 1979 · A. Abdulkadiroğlu, T. Sönmez, "School Choice: A Mechanism Design Approach", *AER* 93(3), 2003.
- E. Maskin, "Nash Equilibrium and Welfare Optimality", *Review of Economic Studies* 66(1), 1999.
- A. R. Karlin, Y. Peres, *Game Theory, Alive*, AMS, 2017 (mechanism design, stable matching) · D. Easley, J. Kleinberg, *Networks, Crowds, and Markets*, Cambridge, 2010 (Chapters 10 and 15).
- Course pages: **Roughgarden CS364A** · **Shah CSC304** · **Procaccia 15-896** · **Dinitz AGT**

The End

Chapter 4: Mechanism Design

Previous: Chapter 3: Nash Equilibrium · Next: Chapter 5: Auctions