Chapter 10: Multi-Agent Systems

Algorithmic Game Theory

Agent properties | Cooperation and competition | Centralized or distributed | Contract net | Coalitional games | Shapley value | The core | Power indices | Public goods and free riding

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_10_Multi_Agent_Systems.html

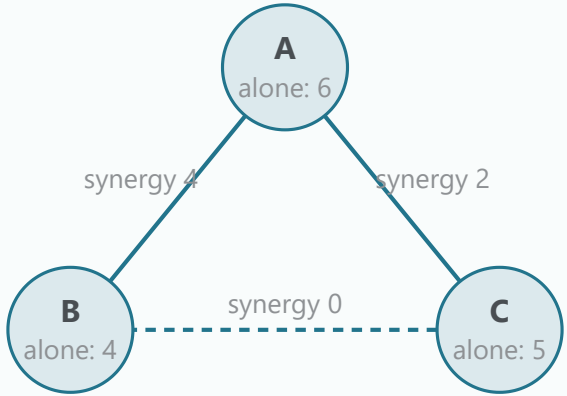
Warm-up: The Shapley Coalition Game

The rules

Each player has a stand-alone value; a coalition's value is its members' stand-alone values plus a synergy bonus for every pair inside it. The grand coalition's value v(N) is split among the players.

Think about it

- In the sample round shown, if the 21 units are split equally (7 each), is A treated unfairly?
- Should the most valuable player on their own always get the largest share?



v(ABC) = 15 + 6 = 21

Learning Outcomes

Agents and MAS

Explain the basic properties of agents and the cooperation, competition and coordination problems of multi-agent systems.

Task allocation

Run the contract net protocol (announce, bid, award) on a small example and compare it with the optimum.

Coalitional games

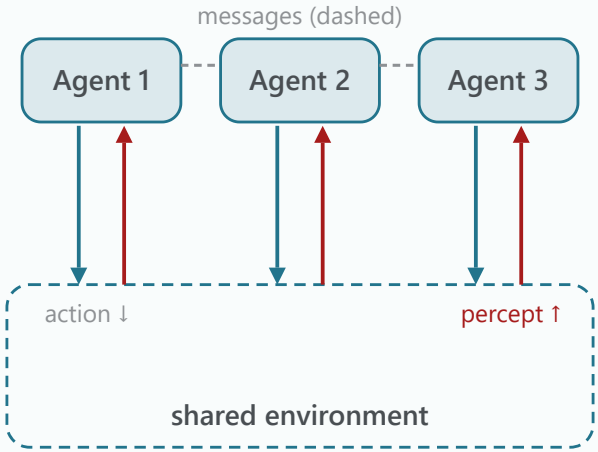
Write a characteristic function; compute the Shapley value from marginal contributions over all orders; test the core; find the power index in weighted voting.

Public goods and free riding

Show with numbers why contributing nothing is a dominant strategy in the public goods game, and recognize its counterparts in multi-agent systems.

What Is a Multi-Agent System (MAS)?

- A system of several autonomous agents that **perceive, decide and act** in a shared environment.
- Agents may share a common goal or **compete** in their own interest.
- Each agent's knowledge is **local**, so coordination and communication are central design problems.
- Examples: robot teams, autonomous vehicle fleets, distributed sensor networks, traffic and economic simulations.



The game theory link: if agents are affected by each other's decisions, the interaction is a **game**; whoever designs the agents' rules is a **mechanism designer**.

The Agent Concept and Its Properties

The four properties in Wooldridge and Jennings's (1995) "weak agent" definition:

Autonomy

Makes its own decisions and manages its own state without human intervention.

`a drone picks its own route`

Reactivity

Perceives changes in its environment and responds in time.

`brakes when it sees an obstacle`

Proactiveness

Does not just react; takes the initiative and plans to reach its goals.

`heads to a charger before the battery dies`

Social ability

Communicates, negotiates and cooperates with other agents.

`bids for a task`

In game theory an agent = a **rational player**: its preferences are captured by a utility function, and it tries to maximize its expected utility.

How Does an Agent Decide? Architectures

Architecture	Decision mechanism	Strength	Example
Reactive	Condition → action rules; no internal model	Fast, robust	Robot vacuum, swarm behavior
Deliberative	World model + planning / search	Long-term goals	Exploration robot, logistics planner
BDI	Belief, desire, intention	Human-like reasoning	Air traffic management, virtual assistant
Learning	Learns a policy from experience	Adapts to unknown environments	Game-playing agents
Hybrid	Reactive lower layer + deliberative upper layer	Speed and foresight together	Autonomous car

An architecture describes the inside of an agent; game theory studies the **interaction** between agents.

Real-World Applications

Autonomous vehicles
Coordination for right of way at intersections, lane changes and platooning.

Drone teams
Dividing up areas in search and rescue: a task allocation problem.

Warehouse robots
Hundreds of robots picking orders; collision-free routes and job distribution.

Online marketplaces
Buyer and seller agents, price bids and automated negotiation.

Smart grid
Energy trading among homes, storage and producers; demand management.

Simulation
Modeling individuals as agents in traffic, epidemic and economic models.

The common question: when agents with their own goals come together, how do we get a good outcome **at the system level**?

Cooperative, Competitive and Mixed Systems

Cooperative MAS

A common goal: agents share information, tasks and resources. The problem is *coordination*.

`search-and-rescue drone team`

Competitive MAS

Different goals: each agent accounts for the others' behavior. The problem is *strategy*.

`bidders in an auction`

Mixed-motive

Some goals are shared, some resources are contested. Most real systems live here.

`homes sharing one grid`

Which is more efficient?	Condition
Cooperation	Goals overlap, communication is cheap, the job is beyond any single agent
Competition	Private information exists and prices/bids can reveal it (markets, auctions)

The answer depends on the system's goal and its resources; mechanism design tries to steer competition toward good outcomes.

The Spectrum of Interests: Three Small Games



Pure coordination

common interest: $u_1 = u_2$

	Right	Left
Right	(2, 2)	(0, 0)
Left	(0, 0)	(1, 1)

two pure eq.: which one?

Prisoner's Dilemma

mixed interests

	Silent	Defect
Silent	(−1, −1)	(−3, 0)
Defect	(0, −3)	(−2, −2)

one eq. (−2, −2), inefficient

Matching Pennies

pure conflict: $u_1 + u_2 = 0$

	Heads	Tails
Heads	(1, −1)	(−1, 1)
Tails	(−1, 1)	(1, −1)

no pure eq.; mixed (½, ½)

- On the left interests are **identical**: communication or a shared convention (e.g. “keep right”) is enough (a coordination problem).
- In the middle interests **partly conflict**: individual incentives work against the common good, so incentives must be designed.
- On the right one's gain is the other's loss (zero-sum): being unpredictable, i.e. a **mixed strategy**, matters.

Orange cells: pure Nash equilibria (both players best-responding).

Individual Rationality, Collective Harm

Prisoner's Dilemma

In the middle game Defect is a dominant strategy; the equilibrium is $(-2, -2)$, while (Silent, Silent) would give both $(-1, -1)$. Individual rationality yields a **collectively bad outcome**.

Bandwidth

When every agent pushes its own flow, the network slows down for everyone.

Common resources

Shared processors, energy or data pools get overused.

Information sharing

Sharing improves the common model; withholding gives an agent a short-term edge.

Its many-player version is the **public goods game**; depleting a common resource is the tragedy of the commons.

Decision Making: Centralized or Distributed?

Agents plan from what they know about the environment. Who decides: a single center, or each agent itself?

	Centralized control	Distributed (full autonomy)
Pros	All information in one place: the **optimum** can be computed; easy coordination, low risk of conflict	Flexible, scalable, quick to react locally; the system keeps running when a part fails
Cons	Single point of failure and a bottleneck; computation explodes as agents are added; agents may misreport private information	Decisions made on local information may clash; selfish agents may reach inefficient equilibria (the Price of Anarchy)
Example	A traffic light control center; a central server in a sensor network	Drone teams, autonomous vehicles, robotic exploration

Most real MAS are **hybrid**: agents decide autonomously while a center sets rules, prices or auctions. The contract net is one example.

Distributed Coordination and Conventions

- Without a center, agents decide using **local information and messages**.
- A protocol must handle communication delay, fault tolerance and scalability together.
- Missing common knowledge leads to clashing plans: two robots meeting in a narrow corridor may both dodge the same way.
- The fix: a **convention / social law** ("keep right"), which picks one equilibrium of the coordination game in advance.

R1 ╲ R2	Right	Left
Right	(1, 1)	(−1, −1)
Left	(−1, −1)	(1, 1)

Corridor game: two pure Nash equilibria (orange)

Both equilibria are equally good; the problem is which one is played. A traffic light gives everyone a common signal that selects one of them (a **correlated equilibrium**).

Game Theory and Incentives in MAS

Analysis: Nash equilibrium

Where do agents' strategic decisions settle? Is the equilibrium efficient?

Design: mechanisms

Rules, payments and auctions steer agents toward truthful reporting and cooperation.

Energy sharing

Balanced incentives (e.g. fair payment for surplus production) → everyone gets enough; wrong incentives → everyone draws at peak time.

Distributed network resources

Bandwidth and compute: without usage fees or quotas, overuse.

Right incentives

+ Increase cooperation, reduce conflict.

Wrong incentives

– Breed selfishness, excessive risk and free riding.

Communication, Learning and Emergent Behavior

Communication and information

Agents know different things; who knows what (the information structure) changes the equilibrium (games of incomplete information). This needs a shared language and messaging protocol.

Learning and adaptation

Agents update strategies from feedback: best-response dynamics, fictitious play (each agent best-responds to the average of its opponents' past moves), reinforcement learning. Since other agents learn too, each agent's environment is *nonstationary*; convergence is not guaranteed.

Emergent behavior

Simple local rules produce unexpected system-level patterns: in Schelling's (1971) model a mild neighbor preference produces stark residential segregation.

System behavior cannot be read off individual agents: equilibrium analysis and simulation are used together. Traffic jams and price swings are also patterns emerging from agent interaction.

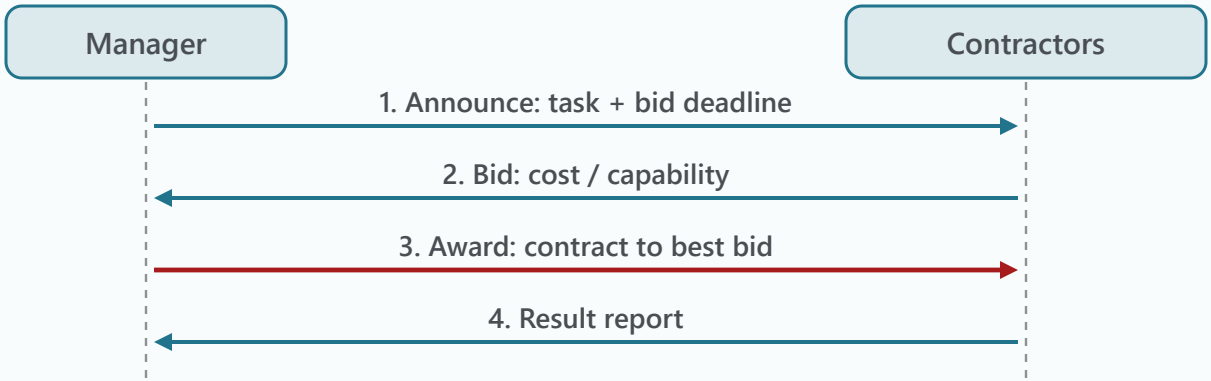
Task and Resource Allocation

- Goal: assign jobs to **suitable agents** by ability and cost (e.g. minimum total time).
- Difficulty: costs are often **private information**; agents are many; tasks arrive over time.
- A good method trades solution quality against communication and computation.

Method	How it works	Pros / cons
Centralized optimization	Collect all costs and solve the assignment problem (Hungarian algorithm: n tasks to n agents at minimum cost in about n^3 steps)	Optimal / needs all information
Contract net	A task is announced, agents bid, the best bid wins	Simple, distributed / can be greedy
Auction	Tasks/resources sold by bidding; bundles via combinatorial auctions	Can be made incentive compatible
Market-based	Prices adjust to supply and demand; agents respond to prices	Scales / may converge slowly

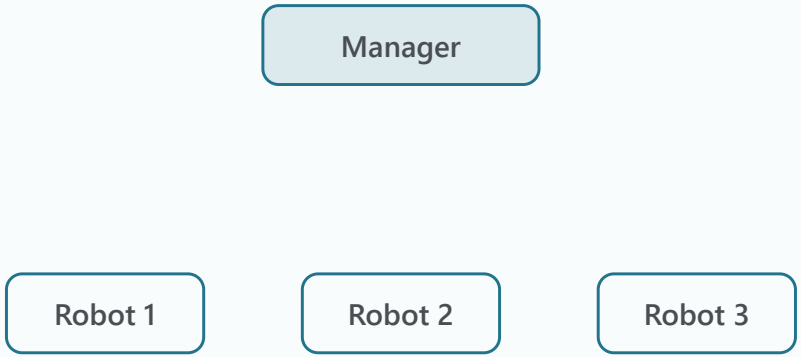
The Contract Net Protocol

Smith (1980): the agent with a task acts as **manager**; agents able to do it act as **contractors**. Roles can change from task to task.



The manager does not know the costs; bids reveal this **private information**. Bidding is a reverse auction (lowest bid wins). If the winner is paid its own bid, a contractor wants to inflate its bid; if paid the second-lowest bid (Vickrey), bidding truthfully is best for a single task.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

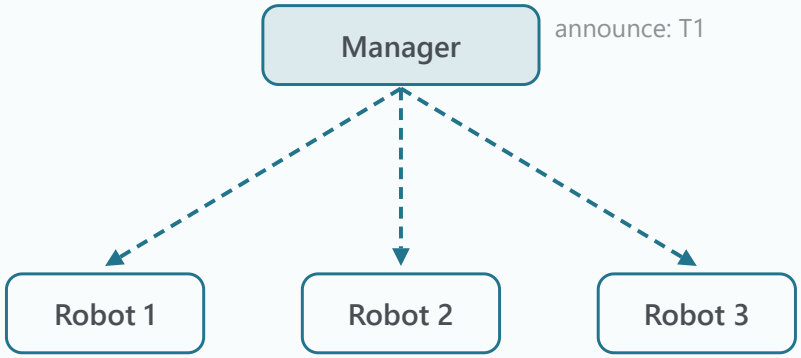
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

The manager auctions three tasks (T1–T3) in order. Each robot takes at most one task; **only the robot knows** its own cost (estimated time); the manager never sees the table.

Step 1 / 11

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

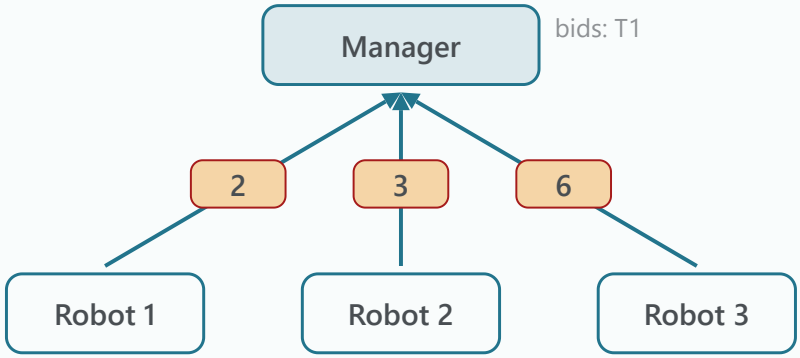
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 0 min

Announce: the manager broadcasts task T1 to the free robots: Robot 1, Robot 2, Robot 3.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

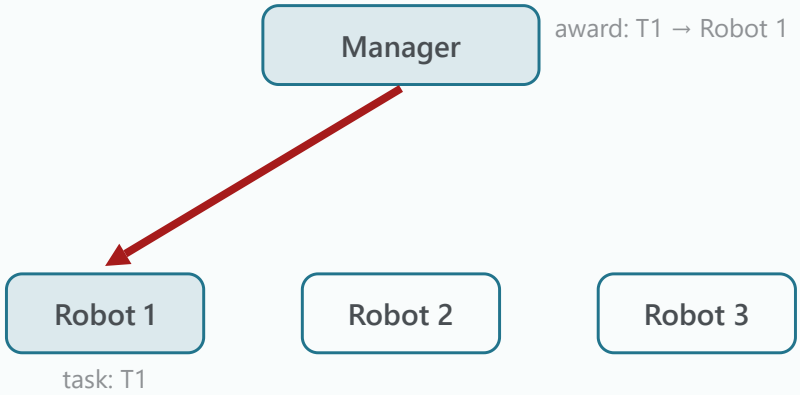
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 0 min

Bids: each free robot reports its cost for T1: R1 → 2, R2 → 3, R3 → 6.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

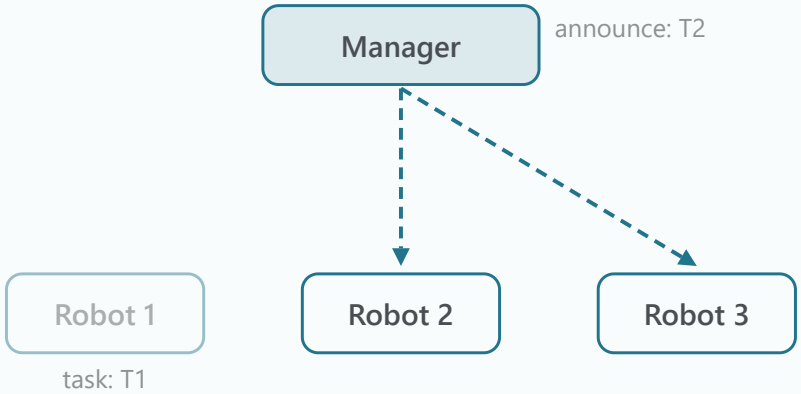
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 2 min

Award: Robot 1 bids lowest (2 min), so T1 goes to it; the robot is now busy.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

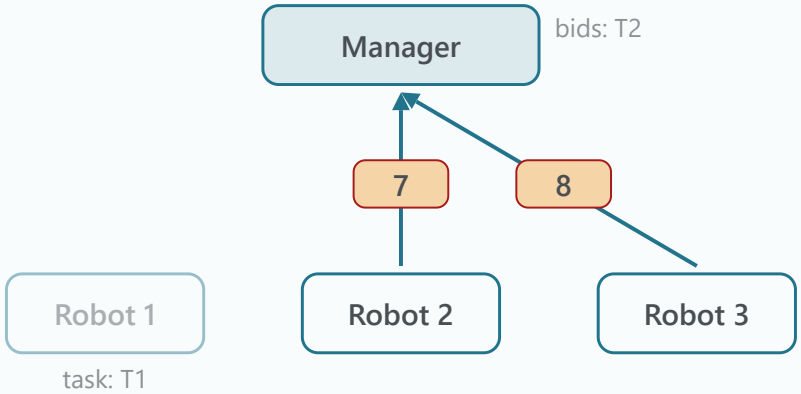
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 2 min

Announce: the manager broadcasts task T2 to the free robots: Robot 2, Robot 3.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

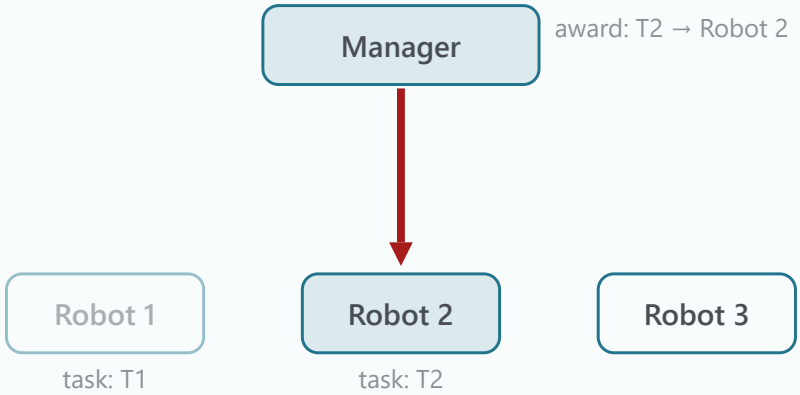
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 2 min

Bids: each free robot reports its cost for T2: R2 → 7, R3 → 8.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

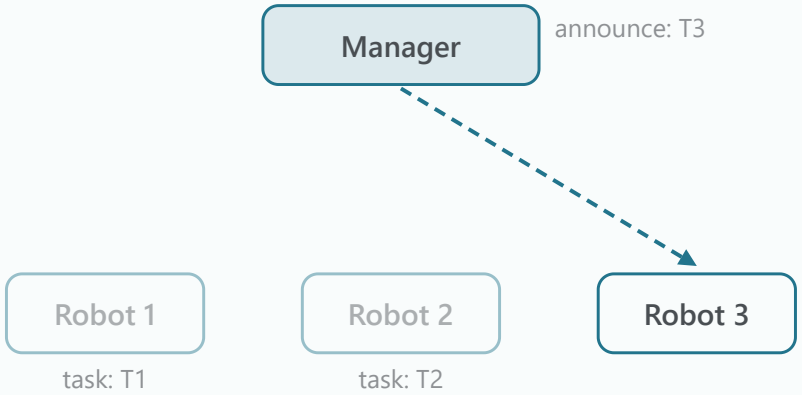
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 9 min

Award: Robot 2 bids lowest (7 min), so T2 goes to it; the robot is now busy.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

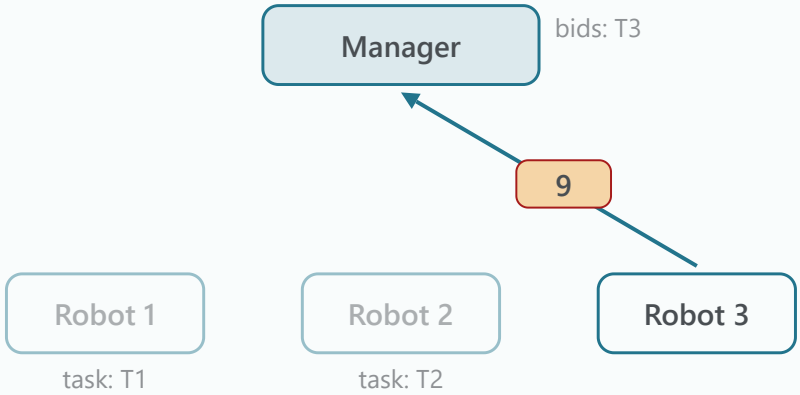
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 9 min

Announce: the manager broadcasts task T3 to the free robots: Robot 3.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

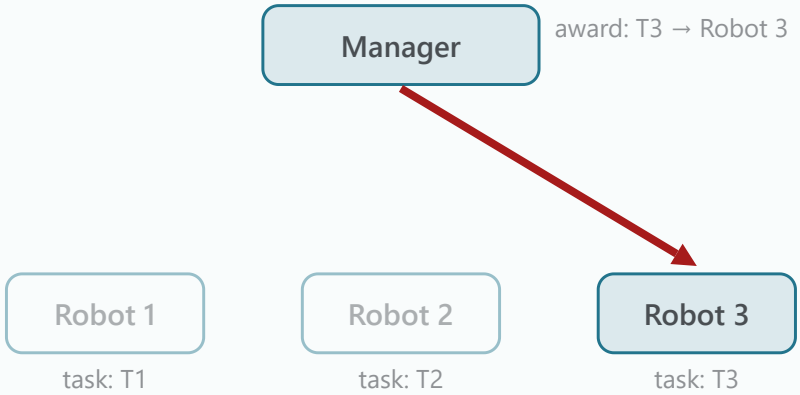
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 9 min

Bids: each free robot reports its cost for T3: R3 → 9.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

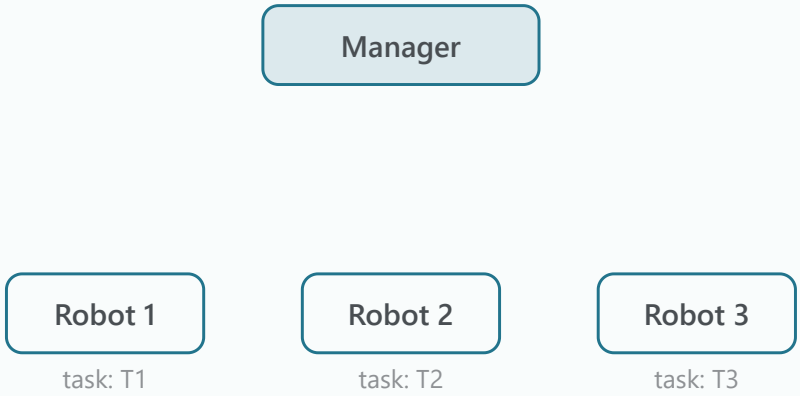
	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

total: 18 min

Award: Robot 3 bids lowest (9 min), so T3 goes to it; the robot is now busy. With only one bid, there is no choice.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Contract Net Step by Step: 3 Tasks, 3 Robots



cost (min) = bid

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

contract net: 18 min

optimum: 13 min

Result: the contract net totals **18 min** (red). The optimum, trying all 6 assignments, is **13 min** (dashed blue): T1→R3, T2→R1, T3→R2. Greedy sequential auctioning costs 38% more.

Legend: dashed arrow = announce · orange = bid · thick red arrow / frame = award · faded robot = busy · dashed blue frame = optimal assignment.

Code Trace: `contract_net(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → ?

T2 → ?

T3 → ?

total = 0 min

tasks = [T1, T2, T3] are auctioned in order. All robots are free; nothing assigned yet.

Step 1 / 11

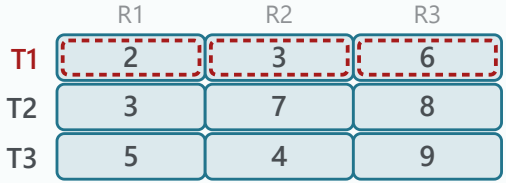
```
▶ def contract_net(tasks, cost):
▶   free = set(cost)
▶   assign, total = {}, 0
4   for g in tasks:
5     bids = {r: cost[r][g] for r in free}
6     winner = min(bids, key=bids.get)
7     assign[g] = winner
8     total += bids[winner]
9     free.remove(winner)
10  return assign, total
```

VARIABLES

`free = {R1, R2, R3}`

`assign = {}` `total = 0`

Code Trace: `contract_net(tasks, cost)`



cost[r][g] (min), known only to the robot

assignment

T1 → ?

T2 → ?

T3 → ?

total = 0 min

Announce: T1. Free robots bid: R1 → 2, R2 → 3, R3 → 6.

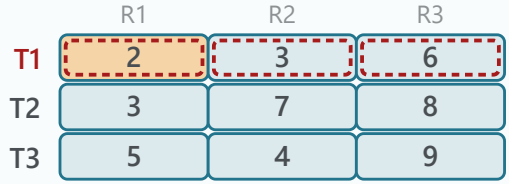
Step 2 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
▶    for g in tasks:
▶      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T1` `free = {R1, R2, R3}` `bids = {R1: 2, R2: 3, R3: 6}` `total = 0`

Code Trace: `contract_net(tasks, cost)`



cost[r][g] (min), known only to the robot

assignment

T1 → ?

T2 → ?

T3 → ?

total = 0 min

Award: lowest bid **R1** (2 min).

Step 3 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
▶      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T1`

`bids = {R1: 2, R2: 3, R3: 6}`

`winner = R1` `total = 0`

Code Trace: `contract_net(tasks, cost)`

	R1 (busy)	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → ?

T3 → ?

total = 2 min

assign[T1] = R1, total = 2; R1 is now busy.


```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
▶      assign[g] = winner
▶      total += bids[winner]
▶      free.remove(winner)
10   return assign, total
```

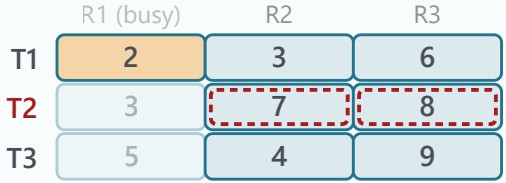
VARIABLES

`g = T1` `winner = R1`

`assign = {T1: R1}` `total = 2`

`free = {R2, R3}`

Code Trace: `contract_net(tasks, cost)`



cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → ?

T3 → ?

total = 2 min

Announce: T2. Free robots bid: R2 → 7, R3 → 8.

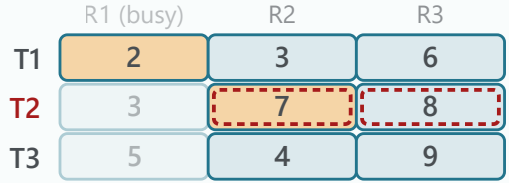
Step 5 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
▶    for g in tasks:
▶      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T2` `free = {R2, R3}` `bids = {R2: 7, R3: 8}` `total = 2`

Code Trace: `contract_net(tasks, cost)`



cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → ?

T3 → ?

total = 2 min

Award: lowest bid **R2** (7 min).

Step 6 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
▶      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T2` `bids = {R2: 7, R3: 8}` `winner = R2` `total = 2`

Code Trace: `contract_net(tasks, cost)`

	R1 (busy)	R2 (busy)	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → R2 (7 min)

T3 → ?

total = 9 min

assign[T2] = R2, total = 9; R2 is now busy. R1, which took T1 cheaply, was also cheapest for T2 (3) but is gone.

Step 7 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
▶      assign[g] = winner
▶      total += bids[winner]
▶      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T2` `winner = R2`

`assign = {T1: R1, T2: R2}`

`total = 9` `free = {R3}`

Code Trace: `contract_net(tasks, cost)`

	R1 (busy)	R2 (busy)	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → R2 (7 min)

T3 → ?

total = 9 min

Announce: T3. Free robots bid: R3 → 9.

Step 8 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
▶    for g in tasks:
▶      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T3` `free = {R3}` `bids = {R3: 9}` `total = 9`

Code Trace: `contract_net(tasks, cost)`

	R1 (busy)	R2 (busy)	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → R2 (7 min)

T3 → ?

total = 9 min

Award: lowest bid **R3** (9 min). A single bid: no choice.

Step 9 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
▶      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T3` `bids = {R3: 9}` `winner = R3` `total = 9`

Code Trace: `contract_net(tasks, cost)`

	R1 (busy)	R2 (busy)	R3 (busy)
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → R2 (7 min)

T3 → R3 (9 min)

total = 18 min

assign[T3] = R3, total = 18; R3 is now busy.

Step 10 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
▶      assign[g] = winner
▶      total += bids[winner]
▶      free.remove(winner)
10   return assign, total
```

VARIABLES

`g = T3` `winner = R3`

`assign = {T1: R1, T2: R2, T3: R3}`

`total = 18` `free = set()`

Code Trace: `contract_net(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

cost[r][g] (min), known only to the robot

assignment

T1 → R1 (2 min)

T2 → R2 (7 min)

T3 → R3 (9 min)

total = 18 min

Result 18 min: the best local decision at each step, but not the optimal total.

Step 11 / 11

```
1  def contract_net(tasks, cost):
2    free = set(cost)
3    assign, total = {}, 0
4    for g in tasks:
5      bids = {r: cost[r][g] for r in free}
6      winner = min(bids, key=bids.get)
7      assign[g] = winner
8      total += bids[winner]
9      free.remove(winner)
▶    return assign, total
```

VARIABLES

`assign = {T1: R1, T2: R2, T3: R3}`

`total = 18`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

(R1, R2, R3) · (R2, R3, R1) ·
(R1, R3, R2) · (R3, R1, R2) ·
(R2, R1, R3) · (R3, R2, R1) ·

p = robots taking tasks (T1, T2, T3); orange = best

The centralized solution: a manager who knows all costs tries all 3! = 6 orderings of the 3 robots.

Step 1 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
▶   best = None
4   for p in permutations(cost):
5     t = sum(cost[r][g]
6             for g, r in zip(tasks, p))
7     if best is None or t < best[0]:
8       best = (t, p)
9   return best
```

VARIABLES

`best = None`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9

(R1, R2, R3) 18 (R2, R3, R1) ·
(R1, R3, R2) · (R3, R1, R2) ·
(R2, R1, R3) · (R3, R2, R1) ·

p = robots taking tasks (T1, T2, T3); orange = best

p = (R1, R2, R3): t = 2 + 7 + 9 = **18** ⇒ new best.

Step 2 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
▶       best = (t, p)
9   return best
```

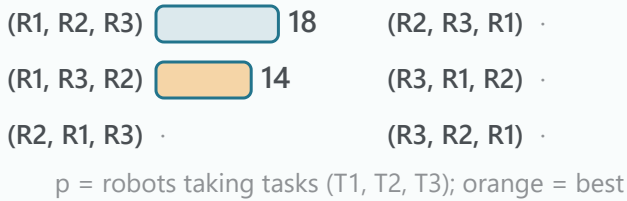
VARIABLES

`p = (R1, R2, R3)` `t = 18`

`best = (18, (R1, R2, R3))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

p = (R1, R3, R2): t = 2 + 8 + 4 = **14** ⇒ new best.

Step 3 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
▶       best = (t, p)
9   return best
```

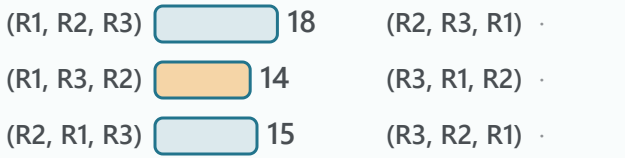
VARIABLES

`p = (R1, R3, R2)` `t = 14`

`best = (14, (R1, R3, R2))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

p = (R2, R1, R3): t = 3 + 3 + 9 = 15 ≥ 14: unchanged.

Step 4 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
8       best = (t, p)
9   return best
```

VARIABLES

`p = (R2, R1, R3)` `t = 15`

`best = (14, (R1, R3, R2))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

p = (R2, R3, R1): t = 3 + 8 + 5 = 16 ≥ 14: unchanged.

Step 5 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
8       best = (t, p)
9   return best
```

VARIABLES

`p = (R2, R3, R1)` `t = 16`

`best = (14, (R1, R3, R2))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

p = (R3, R1, R2): t = 6 + 3 + 4 = **13** ⇒ new best.

Step 6 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
▶       best = (t, p)
9   return best
```

VARIABLES

`p = (R3, R1, R2)` `t = 13`

`best = (13, (R3, R1, R2))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

p = (R3, R2, R1): t = 6 + 7 + 5 = 18 ≥ 13: unchanged.

Step 7 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
▶   for p in permutations(cost):
▶     t = sum(cost[r][g]
▶             for g, r in zip(tasks, p))
▶     if best is None or t < best[0]:
8       best = (t, p)
9   return best
```

VARIABLES

`p = (R3, R2, R1)` `t = 18`

`best = (13, (R3, R1, R2))`

Code Trace: `optimum(tasks, cost)`

	R1	R2	R3
T1	2	3	6
T2	3	7	8
T3	5	4	9



p = robots taking tasks (T1, T2, T3); orange = best

Optimum 13 min: T1→R3, T2→R1, T3→R2; the contract net takes 18 min (38% more). But n robots have n! assignments: 3,628,800 at n = 10 (the Hungarian algorithm solves it in $O(n^3)$).

Step 8 / 8

```
1 from itertools import permutations
2 def optimum(tasks, cost):
3   best = None
4   for p in permutations(cost):
5     t = sum(cost[r][g]
6             for g, r in zip(tasks, p))
7     if best is None or t < best[0]:
8       best = (t, p)
▶   return best
```

VARIABLES

`best = (13, (R3, R1, R2))`

The Contract Net: Strengths and Weaknesses

Strengths

+ Distributed: the manager finds a good assignment without knowing the cost table.
+ O(n) messages per task: one announcement + n bids + one award.
+ Adding robots is easy; a failed robot's task is simply re-auctioned.

Weaknesses

− **Greedy and sequential**: 18 min in the example vs. the optimal 13 (38% more). R1, which took T1 cheaply, was busy although it was the best candidate for T2.
− Robots can strategically inflate their costs.
− Task order changes the outcome.

Improvements: **re-auctioning** (fixing assignments by swaps), **combinatorial auctions** (robots bid on task bundles), and **VCG payments** choosing the whole assignment at once for truthful bids (one-by-one second-price auctions are not enough).

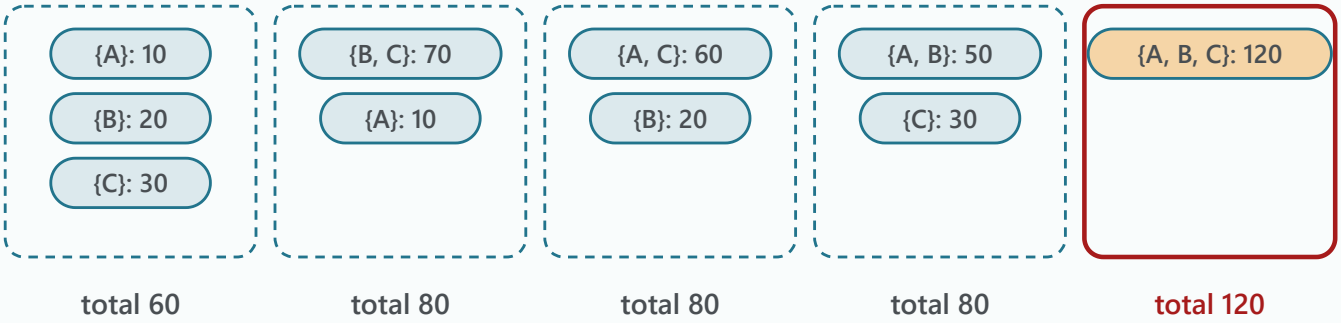
Coalition Formation

- Agents that cannot reach a goal alone may form temporary or lasting **coalitions**: shipping firms share vehicles, drones carry a heavy load together.
- These games are **cooperative**: binding agreements are possible; the question is not "which strategy?" but "who joins whom, and **how is the gain shared**?"
- If the split is unfair, the coalition falls apart: stability depends directly on the sharing rule.

Shapley value
Fairness: everyone gets their average *marginal contribution*.

The core
Stability: no subgroup can get more by breaking away.

Coalition Structures: Who Joins Whom?



- A **coalition structure** partitions the agents into disjoint coalitions. Society's question: which structure has the largest total value?
- The number of structures is the number of ways to split n agents into nonempty disjoint groups (the Bell number): 5 for n = 3, 15 for n = 4, 4,140 for n = 8, 115,975 for n = 10.
- The example game is **superadditive**: merging never hurts, so the best structure is the grand coalition {A, B, C} (120).
- In general, finding the best structure is NP-hard (Sandholm et al., 1999); what remains is **how to split the 120**.

Coalitional Games and the Characteristic Function

- Player set N = {1, …, n}; for every **coalition** S ⊆ N, **v(S)** is the total value S's members can secure together, independently of the others.
- v assigns each coalition the total value the group can earn together; the empty coalition is worth 0. v is called the **characteristic function**. Because the gain can be shared freely within the group, these are transferable utility (TU) games.
- **Superadditive**: for disjoint S, T, v(S ∪ T) ≥ v(S) + v(T) → merging never hurts; the grand coalition N maximizes total value.
- In the example, three logistics firms A, B, C earn 120 together, at most 70 as the best pair and at most 30 alone.

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S): profit (thousand TL)

How Should 120 Be Split?

An **imputation** $x = (x_A, x_B, x_C)$ sums to $v(N) = 120$ (efficiency) and gives everyone at least their stand-alone value (individual rationality).

Proposal	x_A	x_B	x_C	Rationale / problem
Equal split	40	40	40	Simple, but ignores C's larger contribution
Proportional to stand-alone value (10 : 20 : 30)	20	40	60	Looks only at $v(\{i\})$; misses synergies
Strong C's proposal	10	25	85	A and B can get 50 together; they refuse
Shapley value	30	40	50	Average marginal contribution

Two criteria: is the split **fair** (Shapley) and is it **stable** (the core)?

The Shapley Value

- Players arrive in **a random order** and join the coalition. Each player is credited with the value it adds on arrival (its marginal contribution).
- Shapley value: the **average marginal contribution** over all n! orders.
- Cost: n! orders (or 2^{n-1} subsets); for large n it is approximated by sampling.
- Shapley (1953): the **only** rule satisfying the four axioms shown. It is also used to explain models (SHAP: splitting a model's decision into feature contributions).

Efficiency

$\Sigma \varphi_i = v(N)$.

Symmetry

Two players who contribute equally get equal shares.

Dummy player

If it adds nothing to any coalition, $\varphi_i = 0$.

Additivity

If two games' values are added, so are the Shapley shares.

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	·	·	·
A → C → B	·	·	·
B → A → C	·	·	·
B → C → A	·	·	·
C → A → B	·	·	·
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

In each order agents arrive one at a time; each is credited with **the value it adds to the coalition on arrival** (its marginal contribution).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	·	·	·
B → A → C	·	·	·
B → C → A	·	·	·
C → A → B	·	·	·
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

Order A → B → C. A: 10 − 0 = 10; B: 50 − 10 = 40; C: 120 − 50 = 70. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	·	·	·
B → C → A	·	·	·
C → A → B	·	·	·
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

Order A → C → B. A: 10 − 0 = 10; C: 60 − 10 = 50; B: 120 − 60 = 60. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	30	20	70
B → C → A	·	·	·
C → A → B	·	·	·
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

Order B → A → C. B: 20 − 0 = 20; A: 50 − 20 = 30; C: 120 − 50 = 70. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	30	20	70
B → C → A	50	20	50
C → A → B	·	·	·
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

Order B → C → A. B: 20 − 0 = 20; C: 70 − 20 = 50; A: 120 − 70 = 50. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	30	20	70
B → C → A	50	20	50
C → A → B	30	60	30
C → B → A	·	·	·
total	·	·	·
φ = total / 6	·	·	·

Order C → A → B. C: 30 − 0 = 30; A: 60 − 30 = 30; B: 120 − 60 = 60. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	30	20	70
B → C → A	50	20	50
C → A → B	30	60	30
C → B → A	50	40	30
total	·	·	·
φ = total / 6	·	·	·

Order C → B → A. C: 30 − 0 = 30; B: 70 − 30 = 40; A: 120 − 70 = 50. Row total 120 = v(ABC).

Shapley Value Step by Step: 6 Orders

S	v(S)
{A}	10
{B}	20
{C}	30
{A, B}	50
{A, C}	60
{B, C}	70
{A, B, C}	120

v(S)

order	A	B	C
A → B → C	10	40	70
A → C → B	10	60	50
B → A → C	30	20	70
B → C → A	50	20	50
C → A → B	30	60	30
C → B → A	50	40	30
total	180	240	300
φ = total / 6	**30**	**40**	**50**

Average marginal contribution: **φ = (30, 40, 50)**. Total 120: the efficiency axiom holds.

Code Trace: `shapley(N, v)` (1/2)

order (arrival order)

S = ∅

φA 0 | φB 0 | φC 0

phi (summed, not yet divided)

The phi dict is zeroed; `permutations` (itertools) generates the 3! = 6 orders.

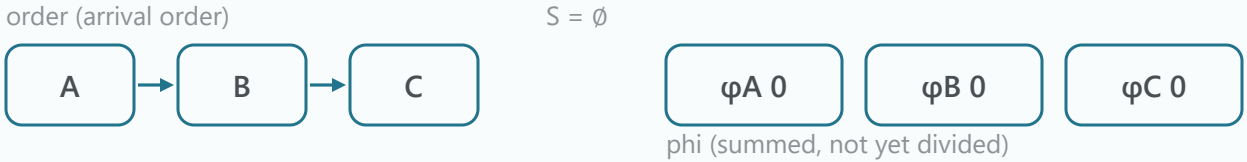
Step 1 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
▶    phi = {i: 0 for i in N}
▶    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `N = [A, B, C]`
- `orders = 6 orders`
- `phi[A] = 0`
- `phi[B] = 0`
- `phi[C] = 0`

Code Trace: `shapley(N, v)` (1/2)



Order 1 (A, B, C): agents arrive in this order. S starts as the empty coalition.

Step 2 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

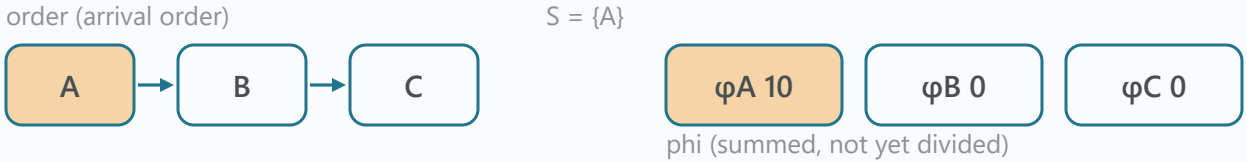
VARIABLES

`order = (A, B, C)` `S = ∅`

`phi[A] = 0` `phi[B] = 0`

`phi[C] = 0`

Code Trace: `shapley(N, v)` (1/2)



A joins: gain = v(A) − v(∅) = 10 − 0 = **10**; phi[A] = 10.

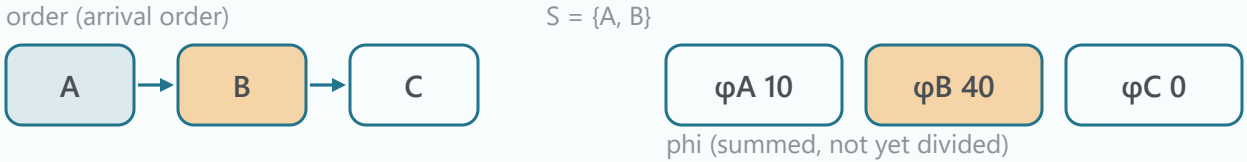
Step 3 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (A, B, C)` `S = {A}` `i = A` `gain = 10` `phi[A] = 10` `phi[B] = 0` `phi[C] = 0`

Code Trace: `shapley(N, v)` (1/2)



B joins: gain = v(AB) − v(A) = 50 − 10 = **40**; phi[B] = 40.

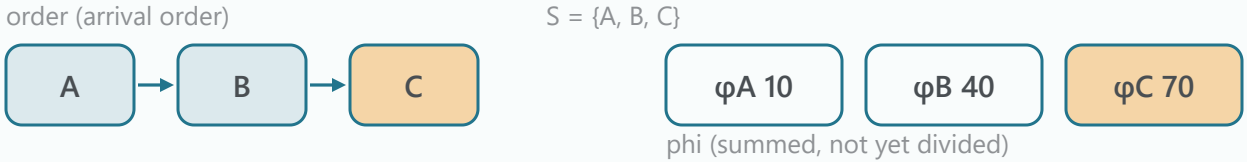
Step 4 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (A, B, C)`
- `S = {A, B}`
- `i = B`
- `gain = 40`
- `phi[A] = 10`
- `phi[B] = 40`
- `phi[C] = 0`

Code Trace: `shapley(N, v)` (1/2)



C joins: gain = v(ABC) − v(AB) = 120 − 50 = **70**; phi[C] = 70.

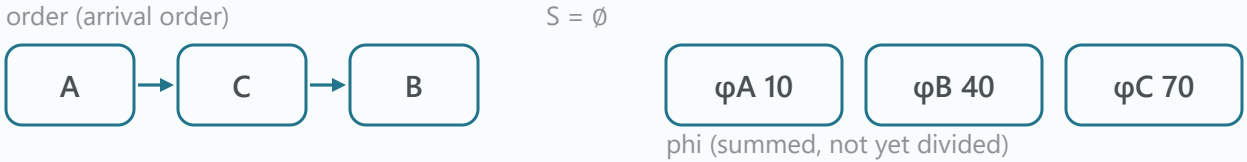
Step 5 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (A, B, C)`
- `S = {A, B, C}`
- `i = C`
- `gain = 70`
- `phi[A] = 10`
- `phi[B] = 40`
- `phi[C] = 70`

Code Trace: `shapley(N, v)` (1/2)



Order 2 (A, C, B): agents arrive in this order. S starts as the empty coalition.

Step 6 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

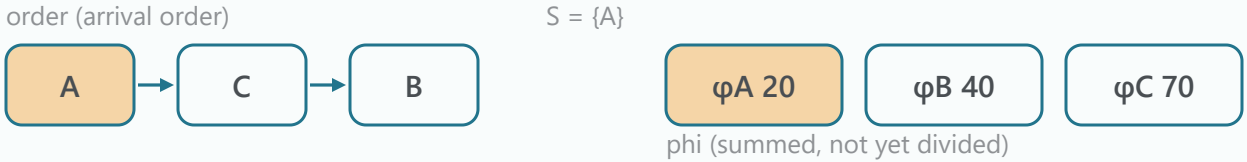
VARIABLES

`order = (A, C, B)` `S = ∅`

`phi[A] = 10` `phi[B] = 40`

`phi[C] = 70`

Code Trace: `shapley(N, v)` (1/2)



A joins: gain = v(A) − v(∅) = 10 − 0 = **10**; phi[A] = 20.

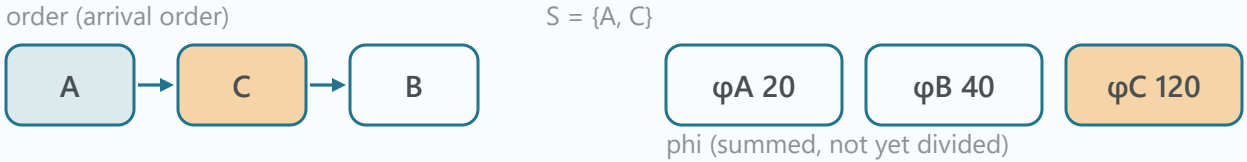
Step 7 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (A, C, B)` `S = {A}` `i = A` `gain = 10` `phi[A] = 20` `phi[B] = 40` `phi[C] = 70`

Code Trace: `shapley(N, v)` (1/2)



C joins: gain = v(AC) − v(A) = 60 − 10 = **50**; phi[C] = 120.

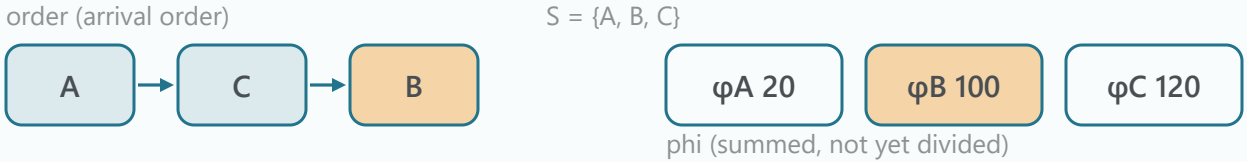
Step 8 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (A, C, B)`
- `S = {A, C}`
- `i = C`
- `gain = 50`
- `phi[A] = 20`
- `phi[B] = 40`
- `phi[C] = 120`

Code Trace: `shapley(N, v)` (1/2)



B joins: gain = v(ABC) − v(AC) = 120 − 60 = **60**; phi[B] = 100.

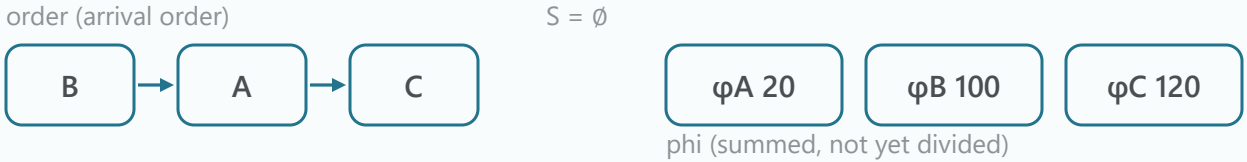
Step 9 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (A, C, B)`
- `S = {A, B, C}`
- `i = B`
- `gain = 60`
- `phi[A] = 20`
- `phi[B] = 100`
- `phi[C] = 120`

Code Trace: `shapley(N, v)` (1/2)



Order 3 (B, A, C): agents arrive in this order. S starts as the empty coalition.

Step 10 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

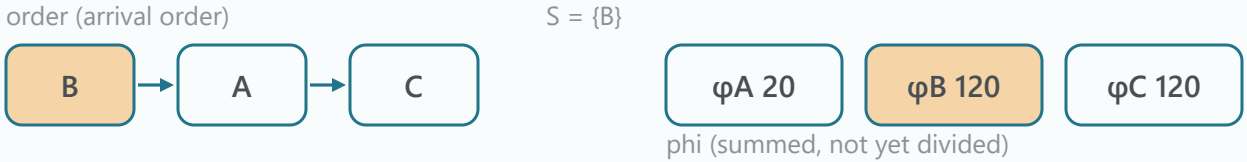
VARIABLES

`order = (B, A, C)` `S = ∅`

`phi[A] = 20` `phi[B] = 100`

`phi[C] = 120`

Code Trace: `shapley(N, v)` (1/2)



B joins: gain = v(B) − v(∅) = 20 − 0 = **20**; phi[B] = 120.

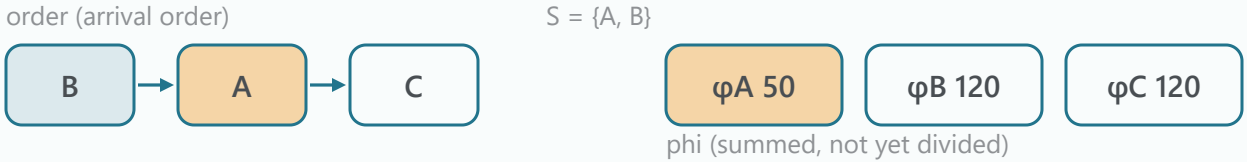
Step 11 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (B, A, C)` `S = {B}` `i = B` `gain = 20` `phi[A] = 20` `phi[B] = 120` `phi[C] = 120`

Code Trace: `shapley(N, v)` (1/2)



A joins: gain = v(AB) − v(B) = 50 − 20 = **30**; phi[A] = 50.

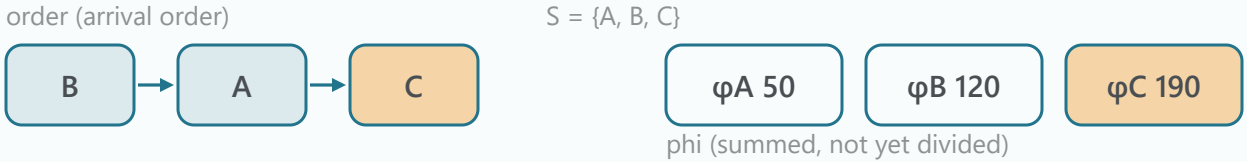
Step 12 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (B, A, C)`
- `S = {A, B}` `i = A`
- `gain = 30` `phi[A] = 50`
- `phi[B] = 120` `phi[C] = 120`

Code Trace: `shapley(N, v)` (1/2)



C joins: gain = v(ABC) − v(AB) = 120 − 50 = **70**; phi[C] = 190.

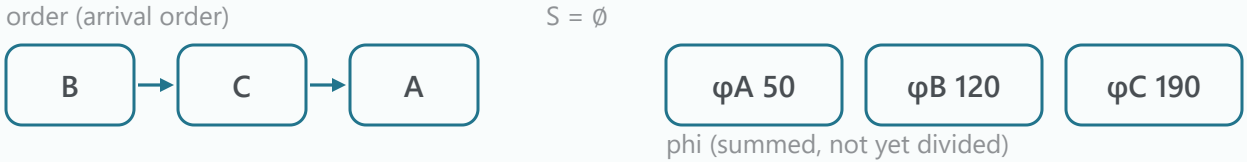
Step 13 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (B, A, C)`
- `S = {A, B, C}`
- `i = C`
- `gain = 70`
- `phi[A] = 50`
- `phi[B] = 120`
- `phi[C] = 190`

Code Trace: `shapley(N, v)` (2/2)



Order 4 (B, C, A): agents arrive in this order. S starts as the empty coalition.

Step 1 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

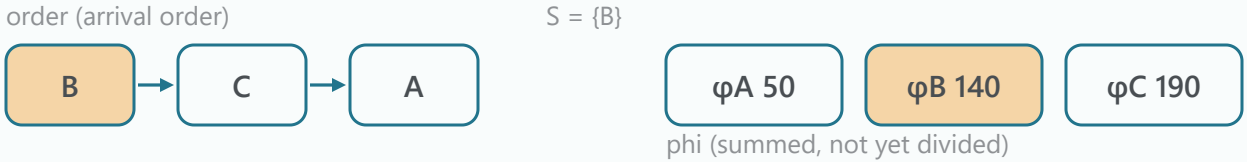
VARIABLES

`order = (B, C, A)` `S = ∅`

`phi[A] = 50` `phi[B] = 120`

`phi[C] = 190`

Code Trace: `shapley(N, v)` (2/2)



B joins: gain = v(B) − v(∅) = 20 − 0 = **20**; phi[B] = 140.

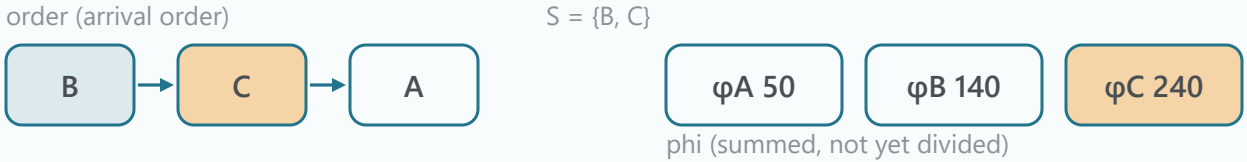
Step 2 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (B, C, A)` `S = {B}` `i = B` `gain = 20` `phi[A] = 50` `phi[B] = 140` `phi[C] = 190`

Code Trace: `shapley(N, v)` (2/2)



C joins: gain = v(BC) − v(B) = 70 − 20 = **50**; phi[C] = 240.

Step 3 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

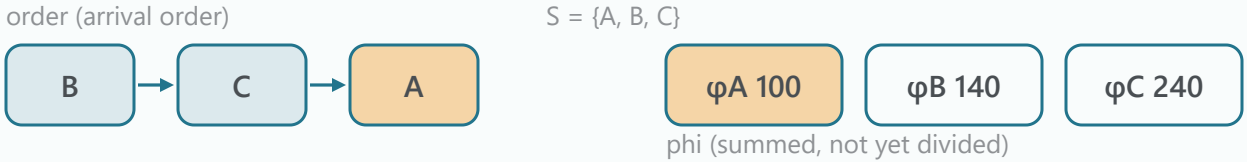
`order = (B, C, A)`

`S = {B, C}` `i = C`

`gain = 50` `phi[A] = 50`

`phi[B] = 140` `phi[C] = 240`

Code Trace: `shapley(N, v)` (2/2)



A joins: gain = v(ABC) − v(BC) = 120 − 70 = **50**; phi[A] = 100.

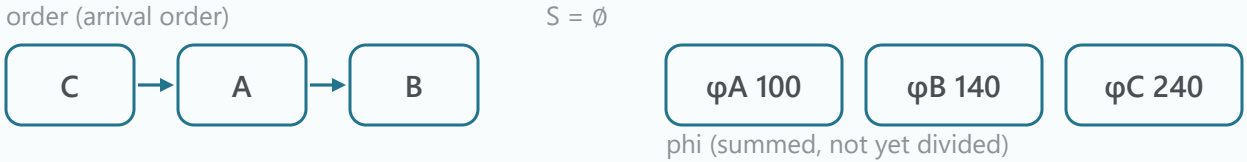
Step 4 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (B, C, A)`
- `S = {A, B, C}`
- `i = A`
- `gain = 50`
- `phi[A] = 100`
- `phi[B] = 140`
- `phi[C] = 240`

Code Trace: `shapley(N, v)` (2/2)



Order 5 (C, A, B): agents arrive in this order. S starts as the empty coalition.

Step 5 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

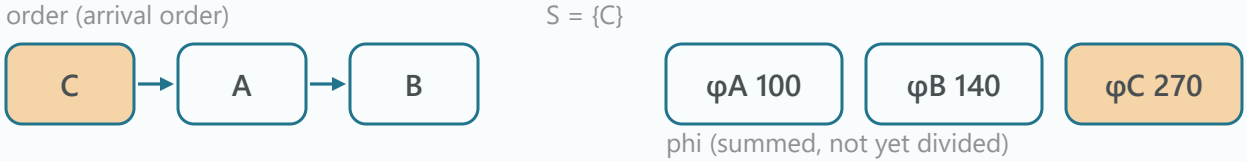
VARIABLES

`order = (C, A, B)` `S = ∅`

`phi[A] = 100` `phi[B] = 140`

`phi[C] = 240`

Code Trace: `shapley(N, v)` (2/2)



C joins: gain = v(C) − v(∅) = 30 − 0 = **30**; phi[C] = 270.

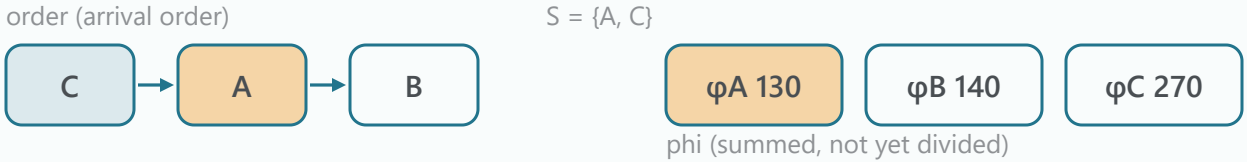
Step 6 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (C, A, B)` `S = {C}` `i = C` `gain = 30` `phi[A] = 100` `phi[B] = 140` `phi[C] = 270`

Code Trace: `shapley(N, v)` (2/2)



A joins: gain = v(AC) − v(C) = 60 − 30 = **30**; phi[A] = 130.

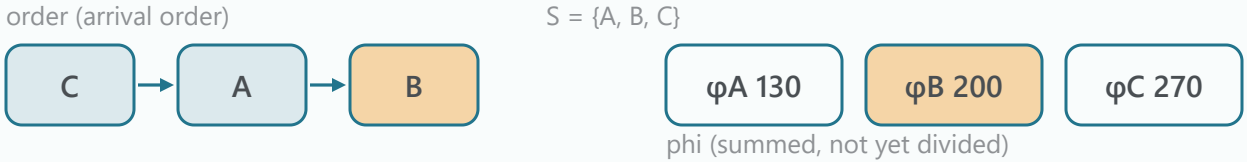
Step 7 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (C, A, B)`
- `S = {A, C}`
- `i = A`
- `gain = 30`
- `phi[A] = 130`
- `phi[B] = 140`
- `phi[C] = 270`

Code Trace: `shapley(N, v)` (2/2)



B joins: gain = v(ABC) − v(AC) = 120 − 60 = **60**; phi[B] = 200.

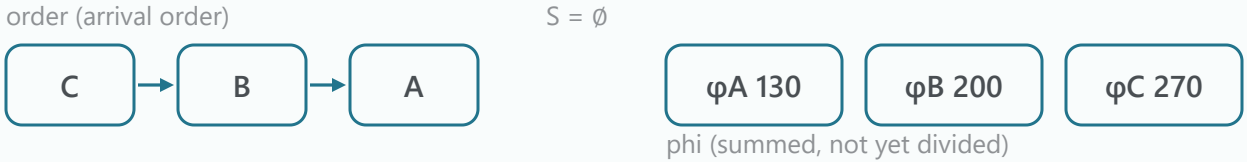
Step 8 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

- `order = (C, A, B)`
- `S = {A, B, C}`
- `i = B`
- `gain = 60`
- `phi[A] = 130`
- `phi[B] = 200`
- `phi[C] = 270`

Code Trace: `shapley(N, v)` (2/2)



Order 6 (C, B, A): agents arrive in this order. S starts as the empty coalition.

Step 9 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

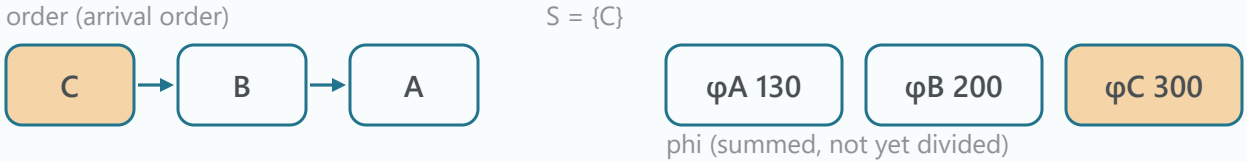
VARIABLES

`order = (C, B, A)` `S = ∅`

`phi[A] = 130` `phi[B] = 200`

`phi[C] = 270`

Code Trace: `shapley(N, v)` (2/2)



C joins: gain = v(C) − v(∅) = 30 − 0 = **30**; phi[C] = 300.

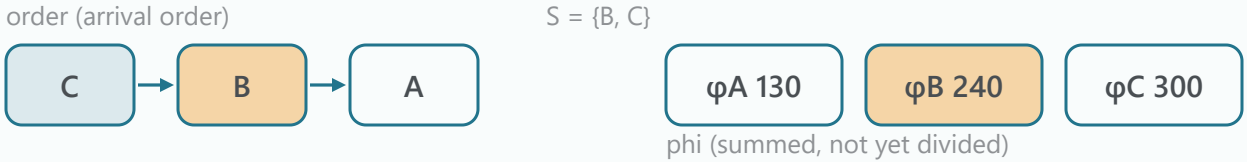
Step 10 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (C, B, A)` `S = {C}` `i = C` `gain = 30` `phi[A] = 130` `phi[B] = 200` `phi[C] = 300`

Code Trace: `shapley(N, v)` (2/2)



B joins: gain = v(BC) − v(C) = 70 − 30 = **40**; phi[B] = 240.

Step 11 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

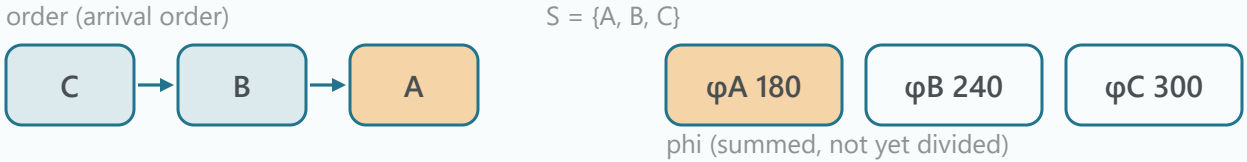
`order = (C, B, A)`

`S = {B, C}` `i = B`

`gain = 40` `phi[A] = 130`

`phi[B] = 240` `phi[C] = 300`

Code Trace: `shapley(N, v)` (2/2)



A joins: gain = v(ABC) − v(BC) = 120 − 70 = **50**; phi[A] = 180.

Step 12 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
▶      for i in order:
▶        gain = v[S | {i}] - v[S]
▶        phi[i] += gain
▶        S = S | {i}
11   return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

`order = (C, B, A)`

`S = {A, B, C}` `i = A`

`gain = 50` `phi[A] = 180`

`phi[B] = 240` `phi[C] = 300`

Code Trace: `shapley(N, v)` (2/2)

order (arrival order) S = ∅

φA 30 | φB 40 | φC 50

φ = total / 6

The totals are divided by 6: φA = 180/6 = 30, φB = 240/6 = 40, φC = 300/6 = 50. Total 120 = v(ABC).

Step 13 / 13

```
1  from itertools import permutations
2  def shapley(N, v):
3    phi = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      S = frozenset()
7      for i in order:
8        gain = v[S | {i}] - v[S]
9        phi[i] += gain
10       S = S | {i}
▶    return {i: phi[i] / len(orders) for i in N}
```

VARIABLES

phi[A] = 180 | phi[B] = 240 | phi[C] = 300

result = {A: 30, B: 40, C: 50}

Back to the Warm-up: The Shapley Coalition Game

The theory's answer

With pairwise synergies, the average marginal contribution is: φ_i = stand-alone value + half of each synergy involving i.

Player	Alone	Synergy share	φ	Equal share
A	6	4/2 + 2/2 = 3	**9**	7
B	4	4/2 + 0 = 2	**6**	7
C	5	2/2 + 0 = 1	**6**	7
Total	15	6	**21**	21

The six arrival orders also give (9, 6, 6). An equal split pays A 2 less.

In practice

Subjects' "fair" splits in three-person games are best fit by a Shapley/equal-split mix (de Clippel and Rozen, 2022). Half-and-half here: (8; 6.5; 6.5).

The game's **equal-split error** measures the same gap: |7 − 9| + |7 − 6| + |7 − 6| = 4.

Sharing by contribution (Shapley) and equal sharing often differ; which is fair depends on context.

The Core

- An imputation x is **in the core** $\Leftrightarrow \Sigma_{i\in N}\, x_i = v(N)$ and $\Sigma_{i\in S}\, x_i \geq v(S)$ for every $S \subseteq N$.
- That is, no coalition can give its members more by **breaking away** (blocking) from the grand coalition.
- The core is a system of linear inequalities (Gillies, 1959): it may be empty, a single point or a region.

Singleton constraints

$x_A \geq 10$, $x_B \geq 20$, $x_C \geq 30$

Pair constraints

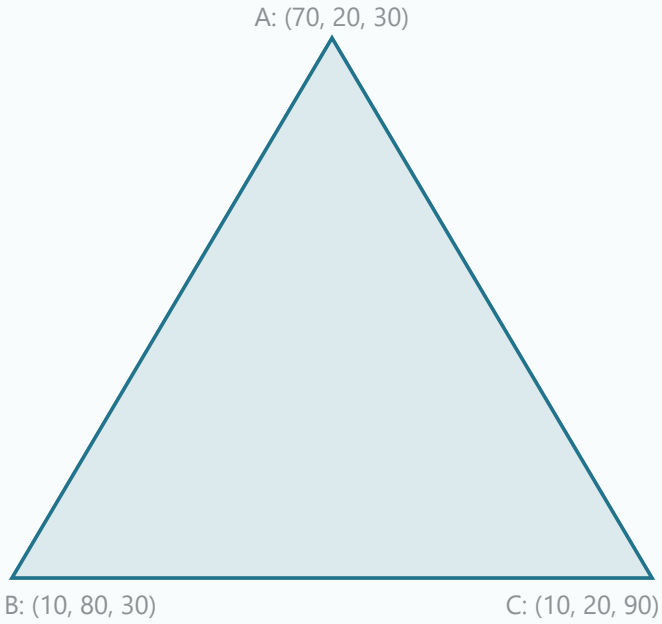
$x_A+x_B \geq 50$, $x_A+x_C \geq 60$, $x_B+x_C \geq 70$

Efficiency

$x_A+x_B+x_C = 120$

Shapley measures fairness, the core stability; they do not always agree. In a **convex** game a player's contribution never shrinks as the coalition it joins grows. Our example is such a game, and in convex games the Shapley value is always in the core (Shapley, 1971).

The Geometry of the Core Step by Step



x = (xA, xB, xC), sum 120

triangle: xA ≥ 10, xB ≥ 20, xC ≥ 30

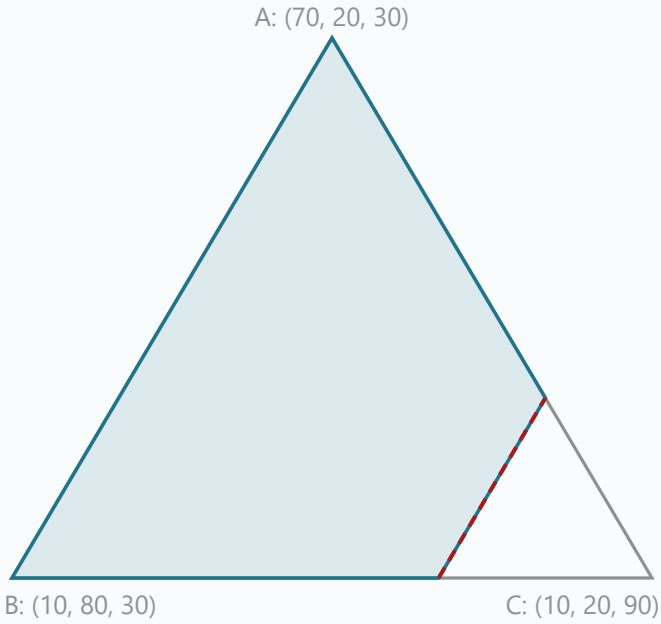
· A+B ≥ 50 ⇔ xC ≤ 70

· A+C ≥ 60 ⇔ xB ≤ 60

· B+C ≥ 70 ⇔ xA ≤ 50

All individually rational, efficient splits form a triangle; at a corner one agent takes all 60 of the surplus.

The Geometry of the Core Step by Step



x = (xA, xB, xC), sum 120

triangle: xA ≥ 10, xB ≥ 20, xC ≥ 30

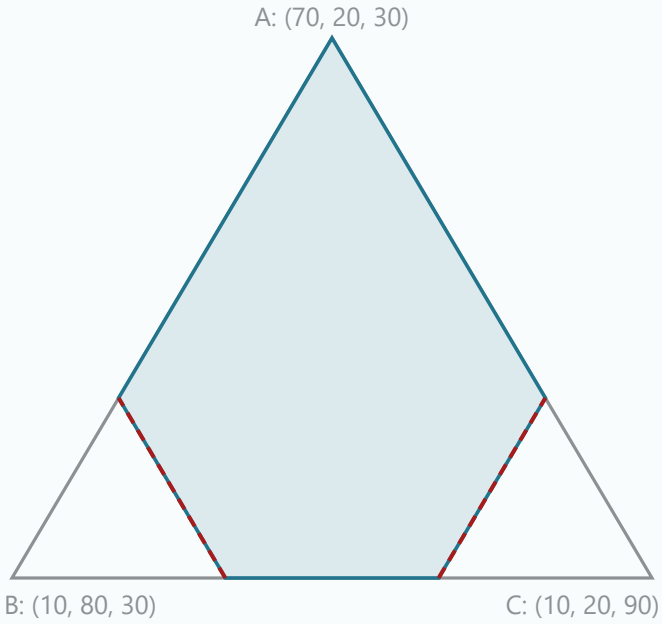
✓ A+B ≥ 50 ⇔ xC ≤ 70

· A+C ≥ 60 ⇔ xB ≤ 60

· B+C ≥ 70 ⇔ xA ≤ 50

{A, B}: $x_A + x_B \geq 50 \Leftrightarrow x_C \leq 70$: if C gets more than 70, A and B break away and take their own 50. The C corner is cut off.

The Geometry of the Core Step by Step



x = (xA, xB, xC), sum 120

triangle: xA ≥ 10, xB ≥ 20, xC ≥ 30

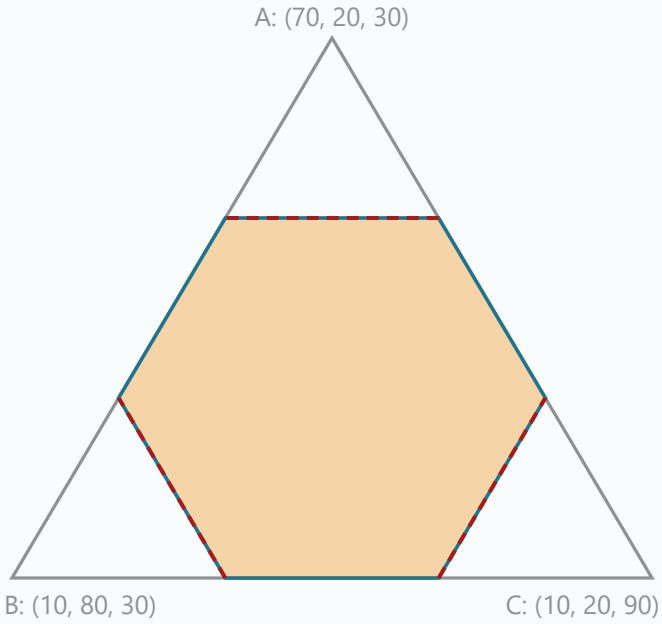
✓ A+B ≥ 50 ⇔ xC ≤ 70

✓ A+C ≥ 60 ⇔ xB ≤ 60

· B+C ≥ 70 ⇔ xA ≤ 50

{A, C}: $x_A + x_C \geq 60 \Leftrightarrow x_B \leq 60$: the B corner is cut off.

The Geometry of the Core Step by Step



x = (xA, xB, xC), sum 120

triangle: xA ≥ 10, xB ≥ 20, xC ≥ 30

✓ A+B ≥ 50 ⇔ xC ≤ 70

✓ A+C ≥ 60 ⇔ xB ≤ 60

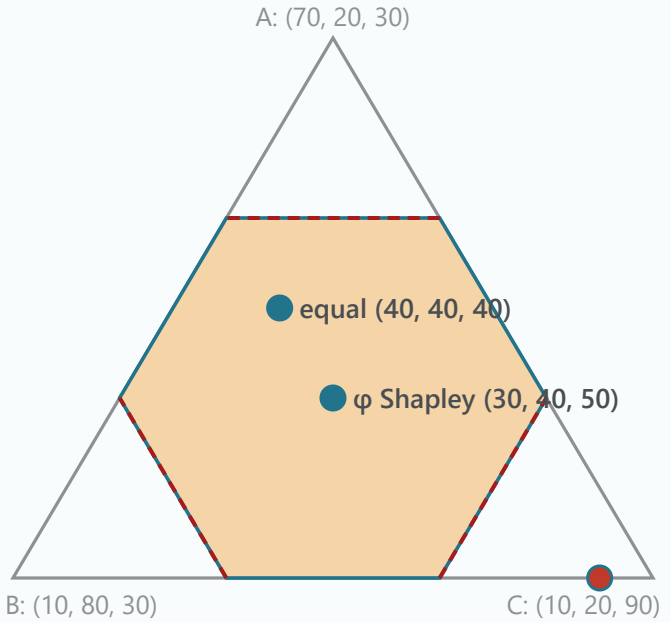
✓ B+C ≥ 70 ⇔ xA ≤ 50

core: a hexagon

corners: (10, 40, 70), (10, 60, 50), (30, 20, 70)
(30, 60, 30), (50, 20, 50), (50, 40, 30)

{B, C}: $x_B + x_C \geq 70 \Leftrightarrow x_A \leq 50$: the A corner is cut off too. The remaining orange hexagon is **the core**.

The Geometry of the Core Step by Step



x = (xA, xB, xC), sum 120

triangle: xA ≥ 10, xB ≥ 20, xC ≥ 30

✓ A+B ≥ 50 ⇔ xC ≤ 70

✓ A+C ≥ 60 ⇔ xB ≤ 60

✓ B+C ≥ 70 ⇔ xA ≤ 50

core: a hexagon

corners: (10, 40, 70), (10, 60, 50), (30, 20, 70)
(30, 60, 30), (50, 20, 50), (50, 40, 30)

φ = (30, 40, 50) and the equal split are in the core (blue). y = (10, 25, 85) is outside (red): $x_A + x_B = 35 < 50$.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	Σ ≥ 10	·	·
{B}	Σ ≥ 20	·	·
{C}	Σ ≥ 30	·	·
{A, B}	Σ ≥ 50	·	·
{A, C}	Σ ≥ 60	·	·
{B, C}	Σ ≥ 70	·	·
{A, B, C}	total = 120	·	·

If an imputation is in the core, **no coalition** can do better by breaking away and taking its own value v(S). Let's test two imputations against the 7 constraints.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	Σ ≥ 10	30 ✓	10 ✓
{B}	Σ ≥ 20	·	·
{C}	Σ ≥ 30	·	·
{A, B}	Σ ≥ 50	·	·
{A, C}	Σ ≥ 60	·	·
{B, C}	Σ ≥ 70	·	·
{A, B, C}	total = 120	·	·

{A}: v(A) = 10. For x 30 ≥ 10 ✓; for y 10 ≥ 10 ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	$\Sigma \geq 10$	30 ✓	10 ✓
{B}	$\Sigma \geq 20$	40 ✓	25 ✓
{C}	$\Sigma \geq 30$	·	·
{A, B}	$\Sigma \geq 50$	·	·
{A, C}	$\Sigma \geq 60$	·	·
{B, C}	$\Sigma \geq 70$	·	·
{A, B, C}	total = 120	·	·

{B}: $v(B) = 20$. For x $40 \geq 20$ ✓; for y $25 \geq 20$ ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	$\Sigma \geq 10$	30 ✓	10 ✓
{B}	$\Sigma \geq 20$	40 ✓	25 ✓
{C}	$\Sigma \geq 30$	50 ✓	85 ✓
{A, B}	$\Sigma \geq 50$	·	·
{A, C}	$\Sigma \geq 60$	·	·
{B, C}	$\Sigma \geq 70$	·	·
{A, B, C}	total = 120	·	·

{C}: v(C) = 30. For x 50 ≥ 30 ✓; for y 85 ≥ 30 ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	Σ ≥ 10	30 ✓	10 ✓
{B}	Σ ≥ 20	40 ✓	25 ✓
{C}	Σ ≥ 30	50 ✓	85 ✓
{A, B}	Σ ≥ 50	70 ✓	35 ✗
{A, C}	Σ ≥ 60	·	·
{B, C}	Σ ≥ 70	·	·
{A, B, C}	total = 120	·	·

{A, B}: v(AB) = 50. For x 70 ≥ 50 ✓; for y 35 ≥ 50 ✗. Breaking away together, {A, B} gets 50: it **blocks** y.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	$\Sigma \geq 10$	30 ✓	10 ✓
{B}	$\Sigma \geq 20$	40 ✓	25 ✓
{C}	$\Sigma \geq 30$	50 ✓	85 ✓
{A, B}	$\Sigma \geq 50$	70 ✓	35 ✗
{A, C}	$\Sigma \geq 60$	80 ✓	95 ✓
{B, C}	$\Sigma \geq 70$	·	·
{A, B, C}	total = 120	·	·

{A, C}: v(AC) = 60. For x 80 ≥ 60 ✓; for y 95 ≥ 60 ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	$\Sigma \geq 10$	30 ✓	10 ✓
{B}	$\Sigma \geq 20$	40 ✓	25 ✓
{C}	$\Sigma \geq 30$	50 ✓	85 ✓
{A, B}	$\Sigma \geq 50$	70 ✓	35 ✗
{A, C}	$\Sigma \geq 60$	80 ✓	95 ✓
{B, C}	$\Sigma \geq 70$	90 ✓	110 ✓
{A, B, C}	total = 120	·	·

{B, C}: v(BC) = 70. For x 90 ≥ 70 ✓; for y 110 ≥ 70 ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	$\Sigma \geq 10$	30 ✓	10 ✓
{B}	$\Sigma \geq 20$	40 ✓	25 ✓
{C}	$\Sigma \geq 30$	50 ✓	85 ✓
{A, B}	$\Sigma \geq 50$	70 ✓	35 ✗
{A, C}	$\Sigma \geq 60$	80 ✓	95 ✓
{B, C}	$\Sigma \geq 70$	90 ✓	110 ✓
{A, B, C}	total = 120	120 ✓	120 ✓

{A, B, C}: v(ABC) = 120. For x 120 = 120 ✓; for y 120 = 120 ✓.

Checking the Core Step by Step

coalition S	constraint	x = (30, 40, 50)	y = (10, 25, 85)
{A}	Σ ≥ 10	30 ✓	10 ✓
{B}	Σ ≥ 20	40 ✓	25 ✓
{C}	Σ ≥ 30	50 ✓	85 ✓
{A, B}	Σ ≥ 50	70 ✓	35 ✗
{A, C}	Σ ≥ 60	80 ✓	95 ✓
{B, C}	Σ ≥ 70	90 ✓	110 ✓
{A, B, C}	total = 120	120 ✓	120 ✓

Result: the Shapley imputation x is **in the core**; y is **outside the core** because {A, B} blocks it. (The game is convex, so the Shapley value is in the core.)

A Game with an Empty Core: The Majority Game

- Three agents share a prize of 1; any **two** agents who agree get it: $v(S) = 1$ ($|S| \geq 2$), otherwise 0.
- Core constraints: $x_A+x_B \geq 1$, $x_A+x_C \geq 1$, $x_B+x_C \geq 1$.
- Adding them gives $2(x_A+x_B+x_C) \geq 3$, so the total is $\geq 3/2$. But the total must be 1 $\Rightarrow$ **the core is empty**.
- In every imputation some pair gets $\leq 2/3$ and wants to exclude the third and share the 1: endless bargaining.

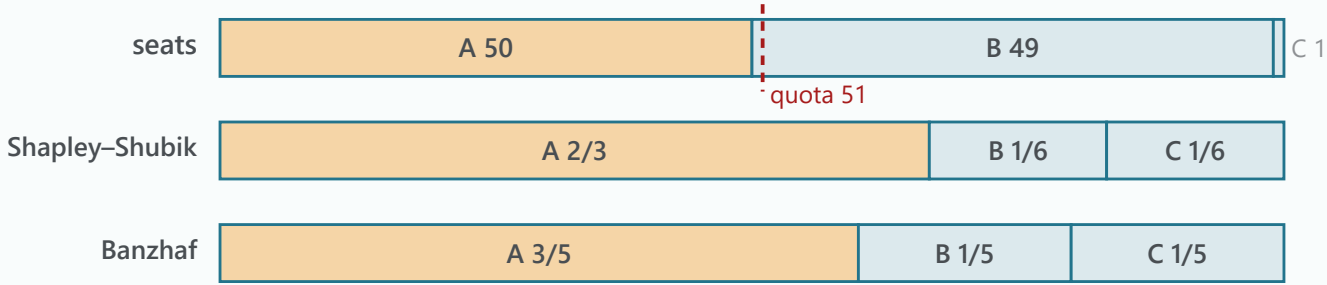
Shapley

$\varphi = (1/3, 1/3, 1/3)$: fair but not stable.

Lesson

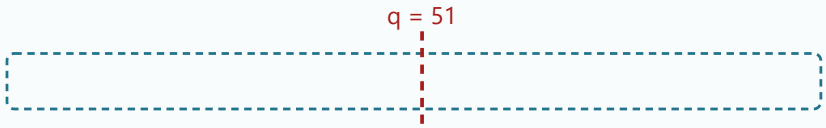
A stable split does not exist in every game. Relaxed rules are then used, such as “no group gains too much by leaving” (e.g. the ε-core: no coalition gains more than ε by breaking away).

Weighted Voting: Weight ≠ Power



- **Weighted voting game** [q; w_1, …, w_n]: a coalition wins if its total weight reaches q, v(S) = 1; otherwise 0.
- **Shapley–Shubik index** (1954) = the Shapley value of this game: the share of orders in which i is **pivotal**.
- Parliament [51; 50, 49, 1]: despite a 48-seat gap, B and C have **equal** power (1/6); the Banzhaf index counts critical memberships: A is critical in three winning coalitions ({A, B}, {A, C}, {A, B, C}), B and C in one each ⇒ (3/5, 1/5, 1/5).
- 1958 EEC Council of Ministers [12; 4, 4, 4, 2, 2, 1]: Luxembourg's vote could never change any coalition's outcome; power = 0.

Code Trace: `power_index(N, w, q)`



pivot counter

A 0

B 0

C 0

w = {A: 50, B: 49, C: 1}, q = 51. 3! = 6 arrival orders; in each, the player who crosses the quota is pivotal.

Step 1 / 14

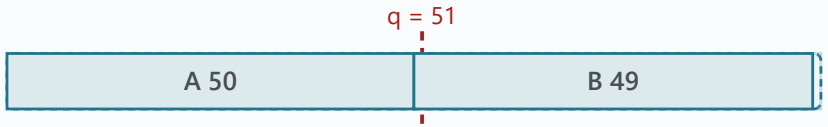
```
1  from itertools import permutations
2  def power_index(N, w, q):
▶    pivots = {i: 0 for i in N}
▶    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

VARIABLES

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

`orders = 6 orders`

Code Trace: `power_index(N, w, q)`



order: A → B → C

pivot counter

A 0
B 0
C 0

Order A → B → C: with the total at 50 < 51, B arrives and the total is 99 ≥ 51.

Step 2 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

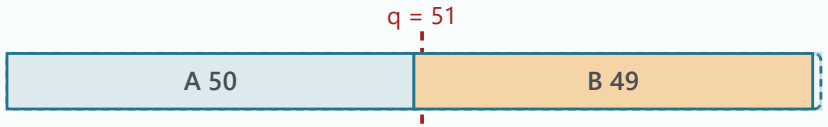
VARIABLES

`order = (A, B, C)` `i = B`

`total = 99`

`pivots = {A: 0, B: 1, C: 0}`

Code Trace: `power_index(N, w, q)`



pivot counter

A 0
B 1
C 0

order: A → B → C pivot: B

B is pivotal: pivots[B] = 1; `break` moves on to the next order.

Step 3 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

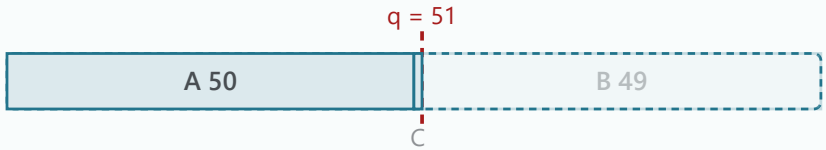
VARIABLES

`order = (A, B, C)` `i = B`

`total = 99`

`pivots = {A: 0, B: 1, C: 0}`

Code Trace: `power_index(N, w, q)`



order: A → C → B

pivot counter

A 0

B 1

C 0

Order A → C → B: with the total at 50 < 51, C arrives and the total is 51 ≥ 51.

Step 4 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

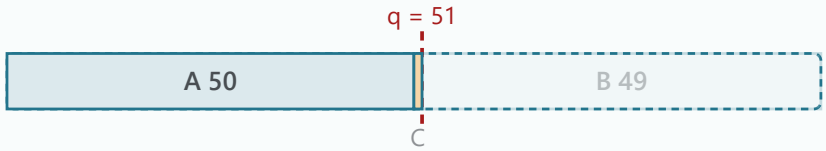
VARIABLES

`order = (A, C, B)` `i = C`

`total = 51`

`pivots = {A: 0, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



order: A → C → B pivot: C

pivot counter

A 0

B 1

C 1

C is pivotal: pivots[C] = 1; `break` moves on to the next order.

Step 5 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

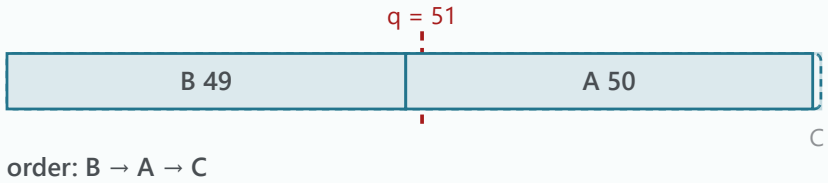
VARIABLES

`order = (A, C, B)` `i = C`

`total = 51`

`pivots = {A: 0, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`





Order B → A → C: with the total at 49 < 51, A arrives and the total is 99 ≥ 51.

Step 6 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

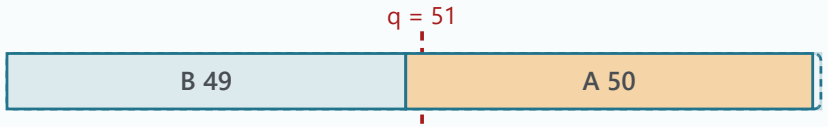
VARIABLES

`order = (B, A, C)` `i = A`

`total = 99`

`pivots = {A: 1, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



pivot counter

A	1
B	1
C	1

order: B → A → C pivot: A

A is pivotal: pivots[A] = 1; `break` moves on to the next order.

Step 7 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

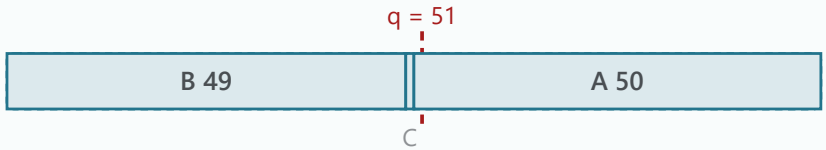
VARIABLES

`order = (B, A, C)` `i = A`

`total = 99`

`pivots = {A: 1, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



order: B → C → A

pivot counter

A 1

B 1

C 1

Order B → C → A: with the total at 50 < 51, A arrives and the total is 100 ≥ 51.

Step 8 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

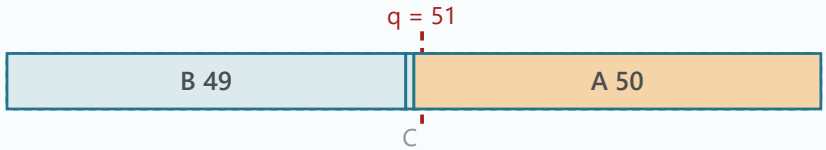
VARIABLES

`order = (B, C, A)` `i = A`

`total = 100`

`pivots = {A: 2, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



order: B → C → A pivot: A

pivot counter

A	2
B	1
C	1

A is pivotal: pivots[A] = 2; `break` moves on to the next order.

Step 9 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

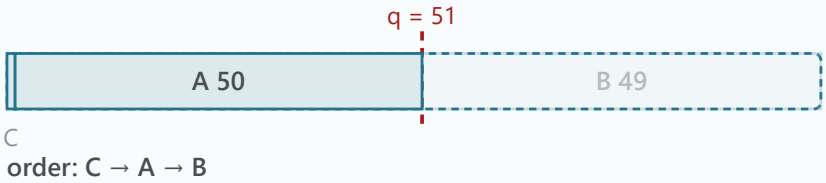
VARIABLES

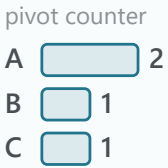
`order = (B, C, A)` `i = A`

`total = 100`

`pivots = {A: 2, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`





Order C → A → B: with the total at 1 < 51, A arrives and the total is 51 ≥ 51.

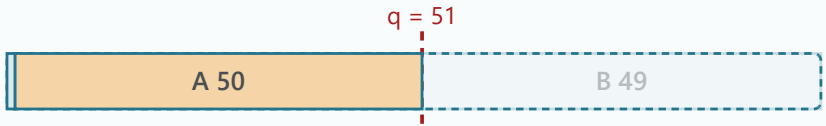
Step 10 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

VARIABLES

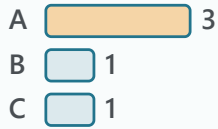
`order = (C, A, B)` `i = A` `total = 51` `pivots = {A: 3, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



order: C → A → B pivot: A

pivot counter



A is pivotal: pivots[A] = 3; `break` moves on to the next order.

Step 11 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

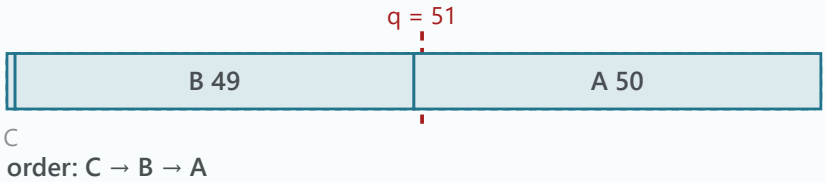
VARIABLES

`order = (C, A, B)` `i = A`

`total = 51`

`pivots = {A: 3, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`





Order C → B → A: with the total at 50 < 51, A arrives and the total is 100 ≥ 51.

Step 12 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
▶    for order in orders:
▶      total = 0
▶      for i in order:
▶        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
12   return {i: pivots[i] / len(orders) for i in N}
```

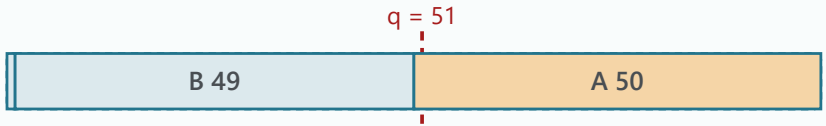
VARIABLES

`order = (C, B, A)` `i = A`

`total = 100`

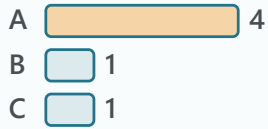
`pivots = {A: 4, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`



order: C → B → A pivot: A

pivot counter



A is pivotal: pivots[A] = 4; `break` moves on to the next order.

Step 13 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
▶        if total >= q:
▶          pivots[i] += 1
▶          break
12   return {i: pivots[i] / len(orders) for i in N}
```

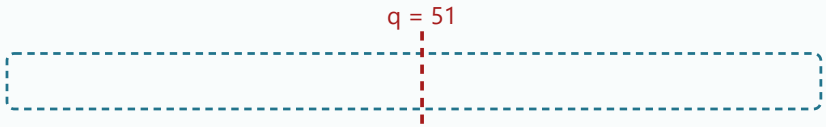
VARIABLES

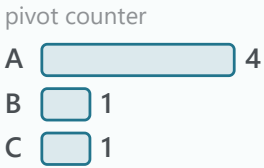
`order = (C, B, A)` `i = A`

`total = 100`

`pivots = {A: 4, B: 1, C: 1}`

Code Trace: `power_index(N, w, q)`





Power = pivots / 6 ⇒ (2/3, 1/6, 1/6). B's 49 seats bring no more power than C's single seat.

Step 14 / 14

```
1  from itertools import permutations
2  def power_index(N, w, q):
3    pivots = {i: 0 for i in N}
4    orders = list(permutations(N))
5    for order in orders:
6      total = 0
7      for i in order:
8        total += w[i]
9        if total >= q:
10         pivots[i] += 1
11         break
▶    return {i: pivots[i] / len(orders) for i in N}
```

VARIABLES

`pivots = {A: 4, B: 1, C: 1}`

`result = {A: 0.667, B: 0.167, C: 0.167}`

The Public Goods Game

- A **public good**: nobody can be excluded from it, and one person's use does not reduce another's (e.g. open-source software, clean air, a shared dataset).
- The game: n players with e tokens each. Player i puts k_i tokens, between 0 and e, into a common pool. The pool is multiplied by r and split **equally among everyone**.
- Payoff: $u_i = e - k_i + (r/n) \cdot \Sigma_j k_j$ ($\Sigma_j k_j$: total tokens in the pool)
- r/n: the part of a token that returns to its contributor (MPCR: marginal per capita return). There is a dilemma if **1/n < r/n < 1**.

Our example

n = 4, e = 10, r = 1.6 ⇒ r/n = 0.4

Individual

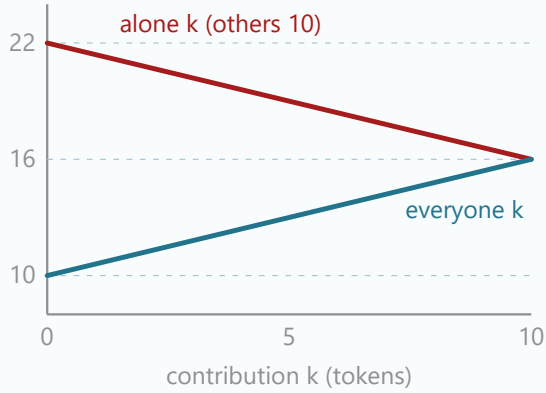
Contributing 1 token: 0.4 returns to the contributor ⇒ net −0.6

Social

Contributing 1 token: 1.6 returns to the group ⇒ net +0.6

Why Free Riding?

- $u_1 = 10 - k_1 + 0.4 \cdot (k_1 + \text{others})$. One more token from P1: −1 from the pocket, +0.4 from the pool, net **−0.6**.
- Whatever others do, each token costs P1 0.6: **k = 0 is dominant**.
- Unique equilibrium: all give 0 → 10 each. All giving 10 yields 16 each: **not Pareto optimal**.
- Red: others give fully, P1's payoff falls from 22 to 16. Blue: all give the same k, 10 up to 16.
- Multi-agent versions: a file-sharing peer that never uploads; a federated-learning client that never sends updates.



The n-player Prisoner's Dilemma: the individual incentive (−0.6) and social welfare (+0.6) point in opposite directions.

Summary

Agents and MAS

Autonomous, reactive, proactive and social agents; cooperative, competitive or mixed systems; most real systems use hybrid control.

Contract net

Announce → bid → award. Distributed and simple, but greedy: 18 min in the example vs. the optimal 13.

Coalitional games

v(S) gives a coalition's value. Shapley = average marginal contribution: (30, 40, 50). In weighted voting, power ≠ weight.

The core

Imputations no coalition wants to leave; it can be empty (the majority game).

Public goods

If r/n < 1, contributing nothing is dominant: equilibrium 10, optimum 16 (per person).

Free riding

An agent that gives nothing to a shared resource (a download-only peer) gains individually; the system loses.

SUMMARY

Sample Problems

1. Contract net example: assignment and total time if auctioned in order T3, T2, T1?
2. Glove game: A has a left glove, B and C each a right glove; a pair is worth 1. Write v; find the Shapley value.
3. The core of this glove game?
4. In the main example, is (40, 40, 40) in the core? Is (20, 25, 75)?
5. Public goods: n = 5, e = 10, r = 2. Payoffs if all contribute fully, and if one free rides?
6. Public goods with r ≥ n: dominant strategy?

▶ **Show answers**

References

- M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed., Wiley, 2009.
- Y. Shoham, K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge University Press, 2009 (Chapter 12).
- G. Chalkiadakis, E. Elkind, M. Wooldridge, *Computational Aspects of Cooperative Game Theory*, Morgan & Claypool, 2011.
- R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver", *IEEE Trans. Computers* C-29(12), 1980.
- L. S. Shapley, "A value for n-person games", *Contributions to the Theory of Games II*, Princeton, 1953; "Cores of convex games", *Int. J. Game Theory* 1, 1971; L. S. Shapley, M. Shubik, *APSR* 48(3), 1954.
- G. de Clippel, K. Rozen, "Fairness through the lens of cooperative game theory: An experimental approach", *American Economic Journal: Microeconomics* 14(3), 2022.
- T. Sandholm et al., "Coalition structure generation with worst case guarantees", *Artificial Intelligence* 111, 1999.

SUMMARY

References: Books, Courses and Tools

- A. R. Karlin, Y. Peres, *Game Theory, Alive*, American Mathematical Society, 2017 (chapter on coalitions and the Shapley value).
- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007 (Chapter 15: cost sharing, the core and the Shapley value).
- C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model", *SIGGRAPH '87*, 1987; T. C. Schelling, "Dynamic models of segregation", *J. Mathematical Sociology* 1(2), 1971.
- FIPA, *Contract Net Interaction Protocol Specification* (SC00029H), 2002.
- M. Wooldridge, N. R. Jennings, "Intelligent agents: Theory and practice", *Knowledge Engineering Review* 10(2), 1995; D. B. Gillies, "Solutions to general non-zero-sum games", *Contributions to the Theory of Games IV*, Princeton, 1959.
- O. N. Bondareva, "Some applications of linear programming methods to the theory of cooperative games", *Problemy Kibernetiki* 10, 1963; L. S. Shapley, "On balanced sets and cores", *Naval Research Logistics Quarterly* 14(4), 1967.
- Course pages: **N. Shah, CSC304 (Toronto)** · **A. Procaccia, 15-896 (CMU)** · **T. Roughgarden, CS364A (Stanford)** · **M. Hoefer, AGT (Frankfurt)**
- Tools: **Nashpy** (Nash equilibria of two-player games, e.g. the three games on the spectrum of interests) · **Gambit** (open-source game theory tools).

The End

Chapter 10: Multi-Agent Systems

Previous: Chapter 9: The Price of Anarchy · Next: Chapter 11: Algorithmic Fairness and Ethics