

COURSE

How Voting Systems Work

10 Chapters

Philip Osztromok

Generated with Claude

HOW VOTING SYSTEMS WORK

Contents

Chapter 1	Why Voting Is Hard	2
Chapter 2	Plurality Quirks	9
Chapter 3	Majorities and Runoffs	17
Chapter 4	Pairwise Contests	28
Chapter 5	Points and Scores	37
Chapter 6	Paradoxes and Limits	46
Chapter 7	Dividing Seats	57
Chapter 8	Districts and Gerrymandering	65
Chapter 9	Power and Weighted Votes	73
Chapter 10	Design an Election	82

How Voting Systems Work 🏆

Chapter 1 · Why Voting Is Hard

Counting votes sounds like the easiest job in the world: add them up and the biggest number wins. Yet the same pile of ballots can name different winners depending on the rule you use to read it, and which rule is "right" is a question people have argued about for more than two centuries. This course takes the rules apart one at a time, from the simplest to the strangest, and tests each with small Python programs. This first chapter sets up the vocabulary, shows the problem with a nine-voter election you can check by hand, and builds the tally code that later chapters reuse. You need arithmetic and a little algebra; the maths stays light, and every idea is something you can run.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Definitions and history come from the Wikipedia articles on plurality voting and the Condorcet paradox, fetched with a tool that often returns truncated pages or model-written summaries, so treat the wording as a paraphrase, not a quotation. The elections in the code are *assumed* or *illustrative*: invented to show a method, not results from any real vote. Statements marked (check) were not confirmed against a source.

What Is a Vote For?

A vote is a small piece of information from one person, and an election is a rule for turning many such pieces into one decision. The decision might be a single winner (a mayor), several winners (a council), or a yes or no (a referendum). Notice that the word "winner" hides a choice about what the group is trying to find: the option the most people put first, the option most people can live with, or the option that beats every rival one on one. Those are different targets, and a voting system is a bet on which one matters.

Voting systems also differ in what each voter is allowed to say. That is the first thing to look at, because it limits everything that can be done afterwards.

BALLOT TYPE	WHAT THE VOTER WRITES	WHAT THE COUNTER LEARNS
Choose one	a single candidate	only each voter's first choice

BALLOT TYPE	WHAT THE VOTER WRITES	WHAT THE COUNTER LEARNS
Ranked	candidates in order of preference	the whole order, so pairwise comparisons are possible
Approval	a tick for every acceptable candidate	who is acceptable, but not the order
Score	a number (say 0 to 5) for each candidate	strength of feeling, as the voter chooses to scale it

The next chapters cover these ballot types and the rules that go with them: How Voting Systems Work 2 for choose-one ballots, How Voting Systems Work 3 and 4 for ranked ballots, and How Voting Systems Work 5 for approval and score ballots. In this chapter we use ranked ballots, because a ranked ballot contains enough information to simulate nearly every other kind.

Preferences and Ballots

A **preference** is what a voter privately thinks; a **ballot** is what the voter is allowed to write down. For a ranked ballot with three candidates there are 3 × 2 × 1 = 6 possible strict orders, and in Python a ballot is simply a tuple with the favourite first. An electorate is then a list of ballots, and a list can hold repeats, so "four voters share this order" is just four copies.

Our running example has nine voters and three candidates, A, B and C. Four voters rank A, B, C; three rank B, C, A; two rank C, B, A. The first tally is the one most people think of as "the election": count first choices.

```
from collections import Counter

# Nine voters, each with a full ranking of three candidates, best first.
# Illustrative numbers: invented to show a method, not a real election.
ballots = (
    [("A", "B", "C")] * 4
    + [("B", "C", "A")] * 3
    + [("C", "B", "A")] * 2
)

firsts = Counter(ballot[0] for ballot in ballots)
total = len(ballots)
for name, votes in firsts.most_common():
```

```
    print(f"{name}: {votes} first-choice votes ({votes / total:.1%})")
winner, top = firsts.most_common(1)[0]
print("plurality winner:", winner, "| has a majority:", top > total / 2)
# A: 4 first-choice votes (44.4%)
# B: 3 first-choice votes (33.3%)
# C: 2 first-choice votes (22.2%)
# plurality winner: A | has a majority: False
```

Candidate A wins a **plurality** with 4 of 9 first-choice votes (44.4%), but does not reach a **majority**, which means more than half. The Wikipedia article on plurality voting defines plurality voting as a system in which the candidate who polls more than any other (a plurality, or relative majority) is elected, and says that it does not require a majority. In the UK the system is called first past the post for general elections (the article does not give a separate US term, but says it is widely used in most of the United States). Critics, the article says, point to wasted votes, pressure toward tactical voting, and the spoiler effect, where similar candidates split the vote. How Voting Systems Work 2 looks at all three.

TERM	MEANING	IN OUR NINE-VOTER EXAMPLE
Plurality	more votes than any rival	A has 4, ahead of B with 3 and C with 2
Majority	more than half of the votes	needs 5 of 9; nobody has it
Preference	what a voter thinks	never seen directly by the counter
Ballot	what a voter writes	a full ranking of A, B and C

The Same Votes, a Different Winner

Winning with 44% looks fine until you ask who the other 56% would have picked. Look at the ballots from the bottom instead of the top:

```
from collections import Counter

ballots = (
    [("A", "B", "C")] * 4
    + [("B", "C", "A")] * 3
    + [("C", "B", "A")] * 2
```

```
)

# The same ballots, looked at from the bottom instead of the top.
lasts = Counter(ballot[-1] for ballot in ballots)
total = len(ballots)
for name in "ABC":
    print(f"{name}: last choice of {lasts[name]} of {total} voters")
# A: last choice of 5 of 9 voters
# B: last choice of 0 of 9 voters
# C: last choice of 4 of 9 voters
```

The plurality winner is the *last* choice of five of the nine voters, a majority, while B is last for nobody. That is uncomfortable. Now compare the candidates head to head: for every pair, give each voter's vote to whichever of the two they rank higher.

```
from itertools import combinations

ballots = (
    [("A", "B", "C")] * 4
    + [("B", "C", "A")] * 3
    + [("C", "B", "A")] * 2
)

def head_to_head(ballots, x, y):
    """Votes for x and for y when each voter picks whichever they rank higher."""
    x_votes = sum(1 for b in ballots if b.index(x) < b.index(y))
    return x_votes, len(ballots) - x_votes

beats = {}
for x, y in combinations("ABC", 2):
    vx, vy = head_to_head(ballots, x, y)
    print(f"{x} vs {y}: {vx} to {vy}")
    beats[(x, y)] = vx > vy
    beats[(y, x)] = vy > vx

for c in "ABC":
    if all(beats[(c, other)] for other in "ABC" if other != c):
        print("beats every rival head to head:", c)
# A vs B: 4 to 5
# A vs C: 4 to 5
# B vs C: 7 to 2
```

```
# beats every rival head to head: B
```

B beats A by 5 to 4 and beats C by 7 to 2, so B would win a straight contest against either rival. A candidate who beats every other candidate head to head is called a **Condorcet winner**, after the Marquis de Condorcet, whose 1785 *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix* is the source the Wikipedia article on the Condorcet paradox cites. Here the plurality rule picks A, while the head-to-head rule picks B. Neither answer is a mistake in counting; they are two definitions of "best". How Voting Systems Work 4 builds the full pairwise matrix.

RULE	QUESTION IT ASKS	WINNER HERE
Plurality	who has the most first places?	A (4 first places)
Condorcet	who beats every rival one on one?	B (5 to 4 and 7 to 2)
Fewest last places	who is disliked by the fewest voters?	B (0 of 9), a rough guide only

When No One Wins: The Cycle

The Condorcet winner in the example was lucky to exist. The Wikipedia article on the Condorcet paradox gives a three-voter example: one voter ranks A, B, C; one ranks B, C, A; one ranks C, A, B. Every pairwise contest ends 2 to 1, and the majority preferences form a loop, so no candidate beats all the others. The article says the paradox shows that majority rule can be self-contradictory even when every individual's preferences are rational (each voter's own ranking is perfectly consistent).

```
ballots = [("A", "B", "C"), ("B", "C", "A"), ("C", "A", "B")]   # three voters

def head_to_head(ballots, x, y):
    x_votes = sum(1 for b in ballots if b.index(x) < b.index(y))
    return x_votes, len(ballots) - x_votes

for x, y in [("A", "B"), ("B", "C"), ("C", "A")]:
    vx, vy = head_to_head(ballots, x, y)
    print(f"{x} vs {y}: {vx} to {vy}  ->  {x if vx > vy else y} wins")
# A vs B: 2 to 1  ->  A wins
```

```
# B vs C: 2 to 1  ->  B wins
# C vs A: 2 to 1  ->  C wins
```

A beats B, B beats C and C beats A. No candidate is the winner by this definition, and a rule that insists on a Condorcet winner has nothing to say. The same article credits the 13th-century Catalan writer Ramon Llull with describing the paradox first, in a work lost until the 21st century, but the page as fetched gave no title or year for it, and older accounts credit only Condorcet (check). Cycles are the root of the impossibility result in How Voting Systems Work 6.

Why This Is Hard

Put the pieces together and you can see the problem the rest of the course explores. The counter sees only ballots, not preferences. Different rules read the same ballots as asking different questions. Some rules sometimes have no answer, or give answers that depend on details such as who else is standing. Every system you meet later can be judged on what it asks, how easy it is to explain, and how it can go wrong.

Common mistakes

Treating “most votes” as if it meant “a majority” (a plurality is often well under half); assuming that the winner of one rule is the winner of all; assuming a Condorcet winner always exists (cycles can happen); forgetting that the counter sees ballots, not preferences, so a ballot may not match what the voter thinks; and reading invented example numbers, such as the ones in this chapter, as if they were measurements of real elections.

EXERCISE 1

A club of 15 members (assumed numbers) ranks four candidates. Write a program that finds the plurality winner and its share of the vote, says whether it is a majority, plays every head-to-head contest, and reports the Condorcet winner if there is one. Do the two methods agree?

View solution

EXERCISE 2

How common are cycles? List all 216 ways three voters can each pick one of the six rankings of three candidates and count the electorates with a cycle. Then simulate random electorates of 3, 11 and 101 voters (seeded) and compare the rates. Does a bigger electorate remove the problem?

View solution

EXERCISE 3

Start from the nine-voter electorate. Move k of the four A-first voters so that they rank B first instead, for k from 0 to 4, and report the plurality winner and the Condorcet winner each time. How many voters have to change before the plurality winner changes, and why does the Condorcet winner not move?

View solution

QUICK REFERENCE

- **Ballot:** what the voter writes (one choice, a ranking, approvals or scores); the counter never sees the underlying preference
- **Plurality:** the most votes; **majority:** more than half; a plurality winner need not have a majority
- **Condorcet winner:** beats every rival head to head; it may not exist
- **Condorcet cycle:** A beats B, B beats C, C beats A, each by majority
- **Python habit:** a ranked ballot is a tuple, an electorate is a list, and `Counter` tallies first choices
- **Check before trusting a winner:** which rule, what the ballot allowed, and whether the numbers are real or illustrative

COMING NEXT

How Voting Systems Work 2 is about plurality and its quirks: first-past-the-post, spoiler candidates, vote splitting, and simulating three-way races.

How Voting Systems Work

Chapter 2 · Plurality Quirks

How Voting Systems Work 1 ended with a nine-voter election in which the plurality winner was the last choice of a majority. This chapter stays with that rule, the simplest one in the course, and asks what else it does. Plurality is the rule behind the phrase “first past the post”, and it has three famous quirks: it can elect a candidate most voters dislike, it punishes candidates who share voters with a similar rival, and it nudges voters toward tactical choices. We will count each quirk in small Python programs, then simulate thousands of three-way races to see how often the rule disagrees with a head-to-head count. The maths stays at arithmetic and percentages.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Definitions come from the Wikipedia articles on the spoiler effect and on Duverger’s law, fetched with a tool that often returns truncated pages or model-written summaries, so treat the wording as a paraphrase, not a quotation. The elections, districts and candidate positions in the code are *assumed* or *illustrative*: invented to show a method, not results from any real vote. The simulations rest on a simple model (one left-to-right line) that real politics does not follow. Statements marked (check) were not confirmed against a source.

The Rule: Most Votes Wins

Under plurality, each voter marks one candidate, and the candidate with the most marks wins, whether or not that is a majority. Nothing else on the ballot is counted. The Wikipedia article on Duverger’s law describes a tendency, which it is careful to call “a tendency rather than an invariant rule”, for plurality voting in single-member districts to favour two-party competition. It names two mechanisms, summarised in the table. The article attributes the law to Maurice Duverger but, as fetched, gave no date for it (check).

MECHANISM	WHAT HAPPENS	WHO ACTS
Mechanical	small parties win few seats, so they have little reason to form or last	the counting rule itself
Psychological	voters avoid “wasting” a vote on a party unlikely to win and move to a bigger one	the voters, in advance of the count

The same article lists exceptions, so treat the law as a pattern and not a certainty. In Canada, it says, the Green Party took about 5% of the national vote from 2004 to 2011 and won only one of the 308 seats in the House of Commons, which is the mechanical effect in numbers. It also says strong regional parties can win seats even when only two parties compete in each district, as in Canada, the UK and India.

Vote Splitting

Suppose Rae and Sam appeal to much the same voters and Tom stands apart. If those voters divide their first choices between Rae and Sam, neither may beat Tom, even though most voters would rather have either of them than Tom. Here are 100 voters (illustrative numbers):

```
from collections import Counter

# 100 voters, three candidates. Rae and Sam are similar; Tom is a rival.
# Illustrative numbers: invented to show a method, not a real election.
ballots = (
    [("Rae", "Sam", "Tom")] * 28
    + [("Sam", "Rae", "Tom")] * 27
    + [("Tom", "Sam", "Rae")] * 45
)

counts = Counter(b[0] for b in ballots)
print("first choices:", dict(counts))
winner, top = counts.most_common(1)[0]
print(f"plurality winner: {winner} with {top / len(ballots):.0%}")

rae_v_tom = sum(1 for b in ballots if b.index("Rae") < b.index("Tom"))
print(f"Rae vs Tom head to head: {rae_v_tom} to {len(ballots) - rae_v_tom}")
sam_v_tom = sum(1 for b in ballots if b.index("Sam") < b.index("Tom"))
print(f"Sam vs Tom head to head: {sam_v_tom} to {len(ballots) - sam_v_tom}")
# first choices: {'Rae': 28, 'Sam': 27, 'Tom': 45}
# plurality winner: Tom with 45%
# Rae vs Tom head to head: 55 to 45
# Sam vs Tom head to head: 55 to 45
```

Tom wins with 45%, but Rae beats Tom 55 to 45 head to head, and so does Sam. In the language of How Voting Systems Work 1, both Rae and Sam would beat Tom in a straight contest; the

plurality count never asks. The next question is who is to blame, and for that we withdraw each candidate in turn and recount.

```
from collections import Counter

ballots = (
    [("Rae", "Sam", "Tom")] * 28
    + [("Sam", "Rae", "Tom")] * 27
    + [("Tom", "Sam", "Rae")] * 45
)

def plurality_winner(ballots):
    """Winner on first choices; ties are listed together, joined by '/'."""
    counts = Counter(b[0] for b in ballots)
    top = max(counts.values())
    return "/".join(sorted(c for c, v in counts.items() if v == top))

def without(ballots, candidate):
    """The same ballots with one candidate struck off every ranking."""
    return [tuple(c for c in b if c != candidate) for b in ballots]

print("all three stand:", plurality_winner(ballots))
for gone in ("Rae", "Sam", "Tom"):
    print(f"{gone} withdraws:", plurality_winner(without(ballots, gone)))
# all three stand: Tom
# Rae withdraws: Sam
# Sam withdraws: Rae
# Tom withdraws: Sam
```

With all three standing Tom wins. If either Rae or Sam withdraws, the other takes the withdrawn candidate's voters and wins. Rae and Sam are each a **spoiler**: the Wikipedia article on the spoiler effect describes it as a losing candidate affecting the result "simply by participating" (as fetched). Tom withdrawing changes the winner too, but Tom is the winner, so this is not a spoiler case. The article cites the 2000 US presidential election, where it says Ralph Nader's campaign is widely cited as a spoiler, but the fetched text gave no vote totals, so none are quoted here (check). It also gives a 2009 mayoral election in Burlington, Vermont, with head-to-head figures of 4,064 to 3,476; that example involves instant-runoff counting, which How Voting Systems Work 3 covers, and I did not verify the figures beyond that one summary (check).

SITUATION	PLURALITY RESULT	HEAD-TO-HEAD RESULT
Rae and Sam both stand	Tom (45 first places)	Rae beats Tom 55 to 45; Sam beats Tom 55 to 45
Only Rae and Tom stand	Rae, 55 to 45, if Sam's voters back Rae (assumed)	Rae
Only Sam and Tom stand	Sam, 55 to 45, if Rae's voters back Sam (assumed)	Sam

Simulating Three-Way Races

One hand-made example proves little, so we need a model that can produce thousands of elections. The simplest is a line from 0 (left) to 1 (right). Voters sit at random points on the line, candidates sit at positions, and each voter ranks the candidates by distance, nearest first. This one-dimensional line is an *assumed* model; real issues have many dimensions. With voters spread along a line, the candidate nearest the middle voter should win head to head (the median-voter idea; I did not confirm it from a source here, so treat it as a property of the model, (check)). The code searches for the Condorcet winner directly, so it does not rely on that idea.

```
import random
from collections import Counter

def run_race(positions, voter_points):
    """One-dimensional race: each voter ranks candidates by distance."""
    ballots = [tuple(sorted(positions, key=lambda c: abs(positions[c] - v))) for v in vo
    counts = Counter(b[0] for b in ballots)
    top = max(counts.values())
    plural = sorted(c for c, n in counts.items() if n == top)
    names = sorted(positions)
    cond = None
    for x in names:
        if all(sum(1 for b in ballots if b.index(x) < b.index(y)) > len(ballots) / 2
               for y in names if y != x):
            cond = x
    return plural, cond

random.seed(2)
voters = [random.random() for _ in range(101)]   # 101 voters spread over 0..1
```

```
positions = {"Rae": 0.30, "Sam": 0.40, "Tom": 0.85}
plural, cond = run_race(positions, voters)
firsts = Counter(min(positions, key=lambda c: abs(positions[c] - v)) for v in voters)
print("first choices:", dict(sorted(firsts.items())))
print("plurality:", plural, "| Condorcet winner:", cond)
# first choices: {'Rae': 30, 'Sam': 28, 'Tom': 43}
# plurality: ['Tom'] | Condorcet winner: Sam
```

Rae and Sam split the left and centre, Tom collects the right with 43 of 101 first places and wins, but Sam, who sits nearest the middle, beats every rival head to head. Now repeat this with random candidate positions and count how often plurality names someone other than the head-to-head winner:

```
import random
from collections import Counter

def run_race(positions, voter_points):
    ballots = [tuple(sorted(positions, key=lambda c: abs(positions[c] - v))) for v in vo
    counts = Counter(b[0] for b in ballots)
    top = max(counts.values())
    plural = sorted(c for c, n in counts.items() if n == top)
    names = sorted(positions)
    cond = None
    for x in names:
        if all(sum(1 for b in ballots if b.index(x) < b.index(y)) > len(ballots) / 2
               for y in names if y != x):
            cond = x
    return plural, cond

random.seed(31)
trials = 2000
for k in (2, 3, 4, 5):
    differ = 0
    for _ in range(trials):
        voters = [random.random() for _ in range(101)]
        positions = {f"C{i}": random.random() for i in range(k)}
        plural, cond = run_race(positions, voters)
        if plural != [cond]:
            differ += 1
    print(f"{k} candidates: plurality differs from the Condorcet winner in {differ / tri
```

```
# 2 candidates: plurality differs from the Condorcet winner in 0.0% of 2000 races
# 3 candidates: plurality differs from the Condorcet winner in 35.8% of 2000 races
# 4 candidates: plurality differs from the Condorcet winner in 54.0% of 2000 races
# 5 candidates: plurality differs from the Condorcet winner in 65.3% of 2000 races
```

With two candidates the rules cannot disagree. With three, they disagree in roughly a third of the simulated races, and the rate climbs as candidates are added: 35.8%, 54.0% and 65.3% for three, four and five. The sample is 2000 seeded races per row, so the exact figures belong to this seed and model; the trend, not the digits, is the finding. This is a simulation of an *assumed* model and says nothing about the frequency in real elections.

Votes Versus Seats

Plurality is usually applied district by district, and a party that is second or third everywhere can win few seats or none. Here are ten districts of 100 voters each with three parties, using invented numbers:

```
# Votes versus seats: ten single-member districts, 100 voters each.
# Illustrative numbers: invented to show a method, not a real election.
districts = [
    {"Red": 40, "Blue": 35, "Green": 25},
    {"Red": 38, "Blue": 36, "Green": 26},
    {"Red": 42, "Blue": 33, "Green": 25},
    {"Red": 37, "Blue": 38, "Green": 25},
    {"Red": 39, "Blue": 34, "Green": 27},
    {"Red": 36, "Blue": 39, "Green": 25},
    {"Red": 41, "Blue": 34, "Green": 25},
    {"Red": 38, "Blue": 37, "Green": 25},
    {"Red": 40, "Blue": 35, "Green": 25},
    {"Red": 35, "Blue": 38, "Green": 27},
]

votes = {p: sum(d[p] for d in districts) for p in ("Red", "Blue", "Green")}
seats = {p: 0 for p in votes}
for d in districts:
    seats[max(d, key=d.get)] += 1

total_votes = sum(votes.values())
```

```
total_seats = sum(seats.values())
for p in votes:
    print(f"{p:5s} votes {votes[p] / total_votes:6.1%}   seats {seats[p]:2d} of {total_s
# Red   votes  38.6%   seats  7 of 10 (70%)
# Blue  votes  35.9%   seats  3 of 10 (30%)
# Green votes  25.5%   seats  0 of 10 (0%)
```

PARTY	SHARE OF VOTES	SHARE OF SEATS	WHAT HAPPENED
Red	38.6%	70%	first in seven districts, mostly by a few votes
Blue	35.9%	30%	close in nearly every district, first in three
Green	25.5%	0%	about a quarter of the electorate, no representation

Green's 255 votes elected nobody, which is what critics mean by *wasted votes*. A Green voter who understands this may vote for Blue or Red instead (the psychological effect in the Duverger table), and Exercise 3 asks what that does to the seats.

> Common mistakes
> Blaming a spoiler candidate for a result when the rule caused it (the same ballots under a different rule would not split); assuming plurality always misses the head-to-head winner (it often agrees, and with two candidates it always does); treating simulation rates from an assumed one-line model as measurements of real elections; reading seat shares as vote shares (they can differ a great deal); and quoting Duverger's law as an invariant rule when the source calls it a tendency.

EXERCISE 1

A club of 21 members ranks four candidates (assumed numbers). Write a program that finds the plurality winner, then withdraws each other candidate in turn and reports whether the winner changes. Which candidates are spoilers?

View solution

EXERCISE 2

Use the one-line model with 101 random voters. For 3, 4 and 5 candidates, simulate 3000 seeded races and count how often removing one losing candidate changes the plurality winner. Does adding more

candidates make spoilers more likely, or does the rate stay roughly flat?

View solution

EXERCISE 3

Take the ten-district electorate from this chapter. Move k Green voters in every district to their second choice, Blue, for k from 0 to 10 in steps of 2, and report the seats each time. At what k does Blue overtake Red, and what does that say about the incentive to vote tactically?

View solution

QUICK REFERENCE

- **Plurality / first past the post:** one mark per voter, most marks wins, no majority needed
- **Vote splitting:** similar candidates share voters and let a rival win with less than half
- **Spoiler:** a losing candidate whose presence changes who wins
- **Duverger's law:** a tendency toward two parties under single-member plurality; mechanical and psychological effects
- **Wasted votes:** votes that elect nobody, which can leave a quarter of voters without a seat
- **Python habit:** write a `without(ballots, candidate)` helper, then rerun the count to test a spoiler
- **Simulations:** state the model, seed the random generator and treat rates as trends, not facts about real elections

COMING NEXT

How Voting Systems Work 3 is about majorities and runoffs: two-round runoffs, instant-runoff (ranked choice) voting, and counting transfers step by step in Python.

How Voting Systems Work

Chapter 3 · Majorities and Runoffs

How Voting Systems Work 2 showed plurality electing a candidate whom most voters ranked below a rival. A natural repair is to insist on a majority: more than half. There are two common ways to get one. The two-round system holds a second election between the leaders; instant-runoff voting, usually called ranked-choice voting in the United States, simulates that second election from ranked ballots in a single trip to the polls. This chapter writes both as short Python functions, prints every transfer so you can follow the count, and then measures how often each one picks the head-to-head winner in the simulated races from the last chapter. The maths is counting and percentages.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Facts about real systems come from web search summaries (Wikipedia pages, a university maths text, advocacy pages), and such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase, not a quotation. The elections, candidates and ballot counts in the code are *assumed* or *illustrative*: invented to show a method, not results from any real vote. The simulation rests on a simple one-line model that real politics does not follow. Statements marked (check) were not confirmed against a source.

The Two-Round System

In the two-round system every voter marks one candidate. If someone has a majority, they win. If not, the top two go to a second round, and the winner of that round has more than half the votes, because only two candidates remain. A summary of the French presidential rules says exactly this: when no candidate has an absolute majority in the first round, a second round is held two weeks later between the two candidates with the most votes, and, as of 2022, every direct presidential election had gone to a second round. The same summary says French legislative races differ, with candidates who reach 12.5% of registered voters advancing, so three or four can be in the second round (check: not confirmed beyond that one summary).

Our function counts first choices, checks for a majority, and otherwise compares the top two finalists using the ballots. Notice the assumption in the docstring: it lets each voter back whichever finalist they ranked higher, which assumes nobody changes their mind between the

rounds. In reality they can, and that is part of the point of holding two rounds. The first profile is the 100-voter split from How Voting Systems Work 2; the second is a new one with a candidate in the middle.

```
from collections import Counter

def two_round(ballots):
    """Round 1: first choices. A majority wins outright; otherwise the top two meet in r
    Assumption: voters keep the same rankings in round 2 and back whichever finalist the
    n = len(ballots)
    first = Counter(b[0] for b in ballots)
    print("  round 1:", dict(sorted(first.items())))
    leader, top = first.most_common(1)[0]
    if top > n / 2:
        print(f"  {leader} has a majority, no second round")
        return leader
    a, b = [c for c, _ in first.most_common(2)]
    a_votes = sum(1 for x in ballots if x.index(a) < x.index(b))
    print(f"  round 2: {a} {a_votes}, {b} {n - a_votes}")
    return a if a_votes > n - a_votes else b

# Illustrative numbers: invented to show a method, not a real election.
split = (
    [("Rae", "Sam", "Tom")] * 28
    + [("Sam", "Rae", "Tom")] * 27
    + [("Tom", "Sam", "Rae")] * 45
)
squeeze = (
    [("Ana", "Bo", "Cy")] * 35
    + [("Cy", "Bo", "Ana")] * 33
    + [("Bo", "Ana", "Cy")] * 16
    + [("Bo", "Cy", "Ana")] * 16
)
for name, profile in (("split vote", split), ("centre squeeze", squeeze)):
    print(name)
    print("  winner:", two_round(profile))
# split vote
#   round 1: {'Rae': 28, 'Sam': 27, 'Tom': 45}
#   round 2: Tom 45, Rae 55
#   winner: Rae
# centre squeeze
#   round 1: {'Ana': 35, 'Bo': 32, 'Cy': 33}
#   round 2: Ana 51, Cy 49
```

```
#   winner: Ana
```

In the split-vote profile the second round does what the plurality count could not: Rae, the other half of the divided pair, meets Tom and wins 55 to 45. In the second profile Bo is first on 32 ballots, last of the three, and is knocked out. Ana and Cy go through, and Ana wins 51 to 49. Look at the ballots again: 65 of the 100 voters (33 + 16 + 16) rank Bo above Ana, so Bo would have beaten Ana head to head, and the same count shows Bo beating Cy by a wide margin too. This is called a **centre squeeze**: the candidate most voters find acceptable is eliminated because few call them first. Remember that for later.

Instant-Runoff Voting

Instant-runoff voting (IRV) asks voters to rank candidates on one ballot. The count then proceeds in rounds: each ballot counts for its highest-ranked candidate still standing; if nobody holds a majority of the ballots in play, the candidate with the fewest votes is eliminated and their ballots move to the next preference on each; repeat until someone does. Each pass is a **transfer**. Australia's House of Representatives is described in the Wikipedia article on optional preferential voting as using this rule, with full preferential voting, which means voters must rank every candidate and ballots never run out. Maine uses ranked-choice voting for some elections (check: I did not confirm which ones).

FEATURE	TWO-ROUND SYSTEM	INSTANT-RUNOFF VOTING
Ballots	one mark per round	a ranking, filled in once
Visits to the polls	one or two	one
Who goes through	the top two on first choices	everyone, until the last-place candidate is dropped each round
Voters can change their mind	yes, between rounds	no, the ranking is fixed (assumed typical rules)
Number of rounds	at most two	up to one fewer than the number of candidates

Here is IRV in Python. The function prints each round, which is the point of the chapter: you can see exactly where each ballot goes. Ties for last place are broken alphabetically here (assumed,

real rules use lots, earlier counts and other devices).

```
from collections import Counter

def irv(ballots, verbose=True):
    """Instant-runoff: drop the candidate with the fewest top choices, pass their ballots
    repeat until someone holds a majority of the ballots still in play.
    Assumption: a tie for last place is broken alphabetically (real rules differ)."""
    remaining = {c for b in ballots for c in b}
    round_no = 1
    while True:
        counts = Counter({c: 0 for c in remaining})
        for b in ballots:
            for c in b:                      # first candidate still standing on this ba
                if c in remaining:
                    counts[c] += 1
                    break
        active = sum(counts.values())
        leader = sorted(counts, key=lambda c: (-counts[c], c))[0]
        if verbose:
            line = ", ".join(f"{c} {counts[c]}" for c in sorted(counts))
            print(f"round {round_no}: {line}  (in play {active}, exhausted {len(ballots)
        if counts[leader] > active / 2:
            if verbose:
                print(f"  {leader} has {counts[leader]} of {active}: winner")
            return leader
        fewest = min(counts.values())
        out = sorted(c for c in remaining if counts[c] == fewest)[0]
        if verbose:
            print(f"  eliminate {out}")
        remaining.discard(out)
        round_no += 1

# Illustrative numbers: invented to show a method, not a real election.
split = (
    [("Rae", "Sam", "Tom")] * 28
    + [("Sam", "Rae", "Tom")] * 27
    + [("Tom", "Sam", "Rae")] * 45
)
print("split vote")
irv(split)
squeeze = (
    [("Ana", "Bo", "Cy")] * 35
```

```
    + [("Cy", "Bo", "Ana")] * 33
    + [("Bo", "Ana", "Cy")] * 16
    + [("Bo", "Cy", "Ana")] * 16
)
print("centre squeeze")
irv(squeeze)
# split vote
# round 1: Rae 28, Sam 27, Tom 45  (in play 100, exhausted 0)
#   eliminate Sam
# round 2: Rae 55, Tom 45  (in play 100, exhausted 0)
#   Rae has 55 of 100: winner
# centre squeeze
# round 1: Ana 35, Bo 32, Cy 33  (in play 100, exhausted 0)
#   eliminate Bo
# round 2: Ana 51, Cy 49  (in play 100, exhausted 0)
#   Ana has 51 of 100: winner
```

On the split profile IRV drops Sam (27), passes those ballots to Rae, and Rae wins 55 to 45: the same result as the two-round count, with no second election. On the centre-squeeze profile it also drops Bo and elects Ana 51 to 49, the same candidate the two-round system elected, and the same candidate who loses head to head to Bo. So fixing the split does not guarantee the head-to-head winner.

Ballots that run out

If voters need not rank everyone, a ballot can run out of candidates: it is **exhausted**, and drops out of the count. The majority is then measured against the ballots still in play, not all those cast. The Maine and Burlington figures I found were mostly from advocacy pages, so the exhaustion rates I saw (for example 10.7% of ballots in Maine's 2018 governor primary, per an MIT analysis of that primary, and 151 of 8,980 in Burlington after the first round, per an advocacy page) are unverified (check).

```
from collections import Counter

def irv(ballots, verbose=True):
    """Instant-runoff: drop the candidate with the fewest top choices, pass their ballot
    repeat until someone holds a majority of the ballots still in play.
    Assumption: a tie for last place is broken alphabetically (real rules differ)."""
    remaining = {c for b in ballots for c in b}
```

```
    round_no = 1
    while True:
        counts = Counter({c: 0 for c in remaining})
        for b in ballots:
            for c in b:                      # first candidate still standing on this ba
                if c in remaining:
                    counts[c] += 1
                    break
        active = sum(counts.values())
        leader = sorted(counts, key=lambda c: (-counts[c], c))[0]
        if verbose:
            line = ", ".join(f"{c} {counts[c]}" for c in sorted(counts))
            print(f"round {round_no}: {line}  (in play {active}, exhausted {len(ballots)
        if counts[leader] > active / 2:
            if verbose:
                print(f"  {leader} has {counts[leader]} of {active}: winner")
            return leader
        fewest = min(counts.values())
        out = sorted(c for c in remaining if counts[c] == fewest)[0]
        if verbose:
            print(f"  eliminate {out}")
        remaining.discard(out)
        round_no += 1

# Illustrative numbers: invented. Voters may rank only some candidates.
ballots = (
    [("Ana",)] * 30
    + [("Bo", "Cy")] * 28
    + [("Cy", "Bo")] * 22
    + [("Dee",)] * 20
)
print("winner:", irv(ballots))
# round 1: Ana 30, Bo 28, Cy 22, Dee 20  (in play 100, exhausted 0)
#   eliminate Dee
# round 2: Ana 30, Bo 28, Cy 22  (in play 80, exhausted 20)
#   eliminate Cy
# round 3: Ana 30, Bo 50  (in play 80, exhausted 20)
#   Bo has 50 of 80: winner
# winner: Bo
```

Bo wins with 50 of the 80 ballots in play, which is a majority of those, but only 50 of the 100 ballots cast, which is not a majority of all of them. Another way to say this is that the headline

"winner has a majority" depends on which total you divide by. Always report both numbers.

Real Elections

A UNL maths text on alternative voting methods describes Burlington, Vermont's 2009 mayoral election, where Andy Montroll was eliminated and his transferred votes lifted Bob Kiss over Kurt Wright. How Voting Systems Work 2 mentioned the head-to-head figures from the Wikipedia summary (4,064 to 3,476); I have not verified them. Sources also differ on timing: one says the city adopted ranked-choice voting for mayoral elections in 2005, while another describes the 2009 contest as the second such election, which suggests an earlier one (check). The lesson is the same as the centre squeeze above: a candidate can be eliminated first and still be the one most voters prefer to the winner.

How Often Do They Pick the Head-to-Head Winner?

We reuse the one-line model from How Voting Systems Work 2: 101 voters at random points on a line, candidates at random positions, voters ranking by distance. For each race we find the Condorcet winner (the candidate who beats every rival head to head; with 101 voters on a line one exists) and ask how often plurality, the two-round system and IRV choose someone else.

```
import random
from collections import Counter

def first_choices(ballots):
    return Counter(b[0] for b in ballots)

def plurality(ballots):
    counts = first_choices(ballots)
    top = max(counts.values())
    return sorted(c for c, v in counts.items() if v == top)

def runoff(ballots):
    n = len(ballots)
    counts = first_choices(ballots)
    ranked = sorted(counts, key=lambda c: (-counts[c], c))
    if counts[ranked[0]] > n / 2:
        return ranked[0]
```

```
    a, b = ranked[:2]
    a_votes = sum(1 for x in ballots if x.index(a) < x.index(b))
    return a if a_votes > n - a_votes else b

def irv(ballots):
    remaining = set(ballots[0])
    while True:
        counts = Counter({c: 0 for c in remaining})
        for b in ballots:
            for c in b:
                if c in remaining:
                    counts[c] += 1
                    break
        leader = sorted(counts, key=lambda c: (-counts[c], c))[0]
        if counts[leader] > len(ballots) / 2:
            return leader
        fewest = min(counts.values())
        remaining.discard(sorted(c for c in remaining if counts[c] == fewest)[0])

def condorcet(ballots):
    names = sorted(ballots[0])
    for x in names:
        if all(sum(1 for b in ballots if b.index(x) < b.index(y)) > len(ballots) / 2
               for y in names if y != x):
            return x

random.seed(31)
trials = 2000
for k in (3, 4, 5):
    miss = {"plurality": 0, "two-round": 0, "instant-runoff": 0}
    for _ in range(trials):
        voters = [random.random() for _ in range(101)]
        pos = {f"C{i}": random.random() for i in range(k)}
        ballots = [tuple(sorted(pos, key=lambda c: abs(pos[c] - v))) for v in voters]
        target = condorcet(ballots)
        if plurality(ballots) != [target]:
            miss["plurality"] += 1
        if runoff(ballots) != target:
            miss["two-round"] += 1
        if irv(ballots) != target:
            miss["instant-runoff"] += 1
    row = "  ".join(f"{m} {n / trials:5.1%}" for m, n in miss.items())
    print(f"{k} candidates: {row}")
# 3 candidates: plurality 35.9%  two-round 17.5%  instant-runoff 17.5%
# 4 candidates: plurality 53.6%  two-round 29.0%  instant-runoff 33.5%
```

```
# 5 candidates: plurality 64.2%  two-round 36.7%  instant-runoff 44.1%
```

Plurality fails in 35.9%, 53.6% and 64.2% of the races, close to the rates seen in the last chapter. Both majority rules do much better: with three candidates the two-round system and IRV each miss in 17.5% of races, but with four and five candidates the gap between them opens and IRV misses more often (33.5% and 44.1%, against 29.0% and 36.7%). The figures depend on this seed and this model, so the trend is what to take away: majority rules help, and neither is free of the problem. Whether the same pattern holds in real elections is unknown from this simulation.

RULE	GUARANTEES A MAJORITY?	CAN ELECT SOMEONE WHO LOSES A HEAD-TO-HEAD CONTEST?
Plurality	no	yes (How Voting Systems Work 2)
Two-round	yes, in round two	yes (centre squeeze)
Instant-runoff	among ballots still in play, not necessarily among all cast	yes (centre squeeze)

There is one more oddity, which Exercise 3 hunts for by machine. IRV is said to fail the *monotonicity* criterion, which means that ranking a candidate higher can sometimes make them lose (check: I took this from memory, not from a source). The exercise finds an explicit example among random profiles and prints it, so you can verify it by hand.

Common mistakes

Assuming a runoff or IRV always picks the head-to-head winner (the centre squeeze shows it need not); calling an IRV winner's share a majority without saying what it is a majority of (ballots in play or ballots cast); assuming voters in a two-round election vote the same way in both rounds; breaking a last-place tie without saying how; and treating simulation rates from an assumed one-line model as measurements of real elections.

EXERCISE 1

Thirty members rank four candidates (assumed numbers). Write a program that prints the first-round counts, announces an outright winner if there is a majority, and otherwise runs the second round between the top two. Then check whether the winner beats every other candidate head to head.

View solution

EXERCISE 2

Run the instant-runoff function on a 40-voter, five-candidate profile in which some ballots rank only one or two candidates (assumed numbers). Print every round, and state how many ballots are exhausted when the winner is declared. Is the winner's count a majority of all 40 ballots?

View solution

EXERCISE 3

Search seeded random 21-voter, three-candidate profiles for an instant-runoff monotonicity failure: a winner who would lose if one group of voters who ranked them below first moved them up a single place. Print the profile and both results.

View solution

QUICK REFERENCE

- **Two-round system:** majority wins outright; otherwise the top two meet in round two
- **Instant-runoff (ranked choice):** drop the last-place candidate, transfer their ballots to the next preference, repeat until someone holds a majority
- **Transfer:** moving an eliminated candidate's ballots to each ballot's next remaining choice
- **Exhausted ballot:** a ballot with no remaining ranked candidate; the majority is then of ballots in play
- **Centre squeeze:** a widely acceptable candidate is eliminated because few rank them first
- **Python habit:** print every round while developing a count, then hide it behind a `verbose` flag for simulations
- **Always state:** the tie-breaking rule, and what a "majority" is a majority of

COMING NEXT

How Voting Systems Work 4 is about pairwise contests: Condorcet winners, cycles, the Copeland and Minimax methods, and building a pairwise matrix.

How Voting Systems Work

Chapter 4 · Pairwise Contests

How Voting Systems Work 3 ended with a centre squeeze: Bo, whom most voters preferred to both rivals, was knocked out first because only 32 of 100 ranked Bo first. Both runoff methods looked only at first choices and then at a shrinking field. A different idea is to hold a miniature election for every pair of candidates and look for someone who wins them all. That idea is usually credited to the Marquis de Condorcet, an eighteenth-century French mathematician, and the candidate it describes is called the **Condorcet winner**. This chapter builds the table of all the pairwise results in Python, shows how the idea can break in a circle, and then looks at two ways of picking a winner anyway: Copeland and Minimax. The maths is counting and comparing two numbers.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Facts about the methods come from one web search whose results were mostly an electoral-methods wiki and a few explainer pages; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase, not a quotation. The sources also disagree in places, and I mark where. The elections, candidates and ballot counts in the code are *assumed* or *illustrative*: invented to show a method, not results from any real vote. The simulations rest on simple models that real politics does not follow. Statements marked (check) were not confirmed against a source.

Head-to-Head Contests

Suppose every voter has ranked all the candidates. For any two candidates, say Ana and Bo, you can count how many ballots put Ana above Bo and how many put Bo above Ana. Whoever has more wins that pair. Do this for every pair and you have a **pairwise matrix**: a grid whose entry in row X and column Y is the number of ballots ranking X above Y. A candidate who wins against every other candidate, by more than half the ballots, is the Condorcet winner. If the ballots are complete and there are no ties, the entry in row X column Y and the entry in row Y column X add up to the number of ballots, so you only ever need to compare one of the two to half the total.

Here is the centre-squeeze profile from the last chapter, with the matrix built from scratch. Each cell is one line of counting, which is the whole method; everything else in this chapter is about what to do with the grid.

```
from collections import Counter

def pairwise(ballots):
    """wins[(x, y)] = number of ballots that rank x above y."""
    names = sorted(ballots[0])
    return {(x, y): sum(1 for b in ballots if b.index(x) < b.index(y))
            for x in names for y in names if x != y}

def condorcet(ballots):
    """The candidate who beats every rival by more than half the ballots, or None."""
    wins = pairwise(ballots)
    n = len(ballots)
    for x in sorted(ballots[0]):
        if all(wins[x, y] > n / 2 for y in ballots[0] if y != x):
            return x
    return None

# Illustrative numbers: the centre-squeeze profile from the last chapter, invented to sh
squeeze = (
    [("Ana", "Bo", "Cy")] * 35
    + [("Cy", "Bo", "Ana")] * 33
    + [("Bo", "Ana", "Cy")] * 16
    + [("Bo", "Cy", "Ana")] * 16
)
wins = pairwise(squeeze)
names = sorted(squeeze[0])
print("row beat column?  (ballots ranking row above column)")
print("      " + "".join(f"{y:>5}" for y in names))
for x in names:
    print(f"{x:>5} " + "".join(f"{'-' if x == y else wins[x, y]:>5}" for y in names))
print("first choices:", dict(sorted(Counter(b[0] for b in squeeze).items())))
print("Condorcet winner:", condorcet(squeeze))
# row beat column?  (ballots ranking row above column)
#         Ana   Bo   Cy
#   Ana     -   35   51
#    Bo    65    -   67
#    Cy    49   33    -
# first choices: {'Ana': 35, 'Bo': 32, 'Cy': 33}
```

```
# Condorcet winner: Bo
```

Read row Bo: 65 ballots rank Bo above Ana and 67 rank Bo above Cy, both more than half of 100. Bo wins both pairs and is the Condorcet winner, though Bo came second on first choices. Neither runoff rule found Bo. A rule that always elects the Condorcet winner when one exists is called a **Condorcet method** (the term appears in the sources I found, which also use "Condorcet-consistent" for the same idea: check the exact wording before relying on either).

When There Is No Winner

Now the awkward part. Majorities between pairs need not line up. Take three equal groups of voters whose rankings are rotations of each other: one group says Ann, then Ben, then Cat; the next says Ben, then Cat, then Ann; the last says Cat, then Ann, then Ben. Every pair has a two-thirds majority in favour of one candidate, and yet the candidates beat one another in a circle.

```
def pairwise(ballots):
    """wins[(x, y)] = number of ballots that rank x above y."""
    names = sorted(ballots[0])
    return {(x, y): sum(1 for b in ballots if b.index(x) < b.index(y))
            for x in names for y in names if x != y}

def condorcet(ballots):
    """The candidate who beats every rival by more than half the ballots, or None."""
    wins = pairwise(ballots)
    n = len(ballots)
    for x in sorted(ballots[0]):
        if all(wins[x, y] > n / 2 for y in ballots[0] if y != x):
            return x
    return None

# Illustrative numbers: three equal groups with rotated rankings, invented to show a meth
ring = (
    [("Ann", "Ben", "Cat")] * 10
    + [("Ben", "Cat", "Ann")] * 10
    + [("Cat", "Ann", "Ben")] * 10
)
wins = pairwise(ring)
```

```
for x, y in (("Ann", "Ben"), ("Ben", "Cat"), ("Cat", "Ann")):
    print(f"{x} vs {y}: {wins[x, y]} to {wins[y, x]}")
print("Condorcet winner:", condorcet(ring))
# Ann vs Ben: 20 to 10
# Ben vs Cat: 20 to 10
# Cat vs Ann: 20 to 10
# Condorcet winner: None
```

Ann beats Ben 20 to 10, Ben beats Cat 20 to 10, and Cat beats Ann 20 to 10. No candidate wins all their contests, so there is no Condorcet winner. This is the **Condorcet paradox**, also called a Condorcet cycle: each individual voter has a sensible ranking, but the majority taken as a whole does not. The sources I found all describe cycles as the reason a Condorcet method needs a fallback rule for when none exists.

FEATURE	PLURALITY OR RUNOFF COUNTS	PAIRWISE MATRIX
Looks at	first choices, then a shrinking field	every pair of candidates, using the whole ranking
Always gives a winner	yes (apart from ties)	no, a cycle can leave nobody undefeated
Finds the centre-squeeze winner	no (How Voting Systems Work 3)	yes, if that candidate beats everyone
Work to count	one pass over the ballots	one pass per pair: with k candidates, k(k-1)/2 pairs

Two Ways to Break a Cycle

The sources I found list many ways to choose a winner when there is a cycle, including Schulze, Ranked Pairs, Kemeny-Young, Copeland and Minimax. We take the two simplest to code.

Copeland scores each candidate by pairwise wins minus pairwise losses and elects the highest score. The summaries I read say it often ends in a tie when there is a cycle, and that critics object that it counts the number of wins and losses without looking at how big the margins were. **Minimax** (the sources also call it Simpson or Simpson-Kramer) looks at each candidate's worst defeat and elects the candidate whose worst defeat is smallest. One page lists several Minimax variants (winning votes, margins, opposition) and says the opposition variant is not a Condorcet

method; the pages disagree on whether “Minimax” as a whole is Condorcet-consistent, so the variant matters (conflict between sources: I did not resolve it). The code below uses the winning-votes version, and I have not proved here that it always elects a Condorcet winner when there is one.

The profile below is invented, found by a seeded random search for a case where the two methods disagree. Ana and Cy tie 12 to 12, and everyone else is in a loop of results.

```
def pairwise(ballots):
    """wins[(x, y)] = number of ballots that rank x above y."""
    names = sorted(ballots[0])
    return {(x, y): sum(1 for b in ballots if b.index(x) < b.index(y))
            for x in names for y in names if x != y}

def condorcet(ballots):
    """The candidate who beats every rival by more than half the ballots, or None."""
    wins = pairwise(ballots)
    n = len(ballots)
    for x in sorted(ballots[0]):
        if all(wins[x, y] > n / 2 for y in ballots[0] if y != x):
            return x
    return None

def copeland(ballots):
    """Score = pairwise wins minus pairwise losses (a tied pair counts for neither)."""
    wins, n = pairwise(ballots), len(ballots)
    return {x: sum((wins[x, y] > n - wins[x, y]) - (wins[x, y] < n - wins[x, y])
                   for y in ballots[0] if y != x)
            for x in sorted(ballots[0])}

def minimax(ballots):
    """Score = size of the candidate's worst defeat, in votes (0 if never beaten).
    Lower is better. This is the 'winning votes' version; other versions exist."""
    wins = pairwise(ballots)
    return {x: max([wins[y, x] for y in ballots[0] if y != x and wins[y, x] > wins[x, y]
            for x in sorted(ballots[0])}

# Illustrative numbers: 24 invented ballots for four candidates, built to give no Condor
groups = [
    (7, ("Di", "Cy", "Ana", "Bo")),
    (8, ("Ana", "Bo", "Cy", "Di")),
    (5, ("Bo", "Cy", "Di", "Ana")),
    (2, ("Bo", "Di", "Ana", "Cy")),
```

```
    (2, ("Ana", "Di", "Bo", "Cy")),
]
ballots = [ranking for count, ranking in groups for _ in range(count)]
wins = pairwise(ballots)
n = len(ballots)
for x in sorted(ballots[0]):
    for y in sorted(ballots[0]):
        if x < y:
            print(f"{x} vs {y}: {wins[x, y]} to {n - wins[x, y]}")
print("Condorcet winner:", condorcet(ballots))
cope, mini = copeland(ballots), minimax(ballots)
print("Copeland scores:", cope)
print("Minimax worst defeats:", mini)
print("Copeland picks:", max(cope, key=cope.get))
print("Minimax picks:", min(mini, key=mini.get))
# Ana vs Bo: 17 to 7
# Ana vs Cy: 12 to 12
# Ana vs Di: 10 to 14
# Bo vs Cy: 17 to 7
# Bo vs Di: 15 to 9
# Cy vs Di: 13 to 11
# Condorcet winner: None
# Copeland scores: {'Ana': 0, 'Bo': 1, 'Cy': 0, 'Di': -1}
# Minimax worst defeats: {'Ana': 14, 'Bo': 17, 'Cy': 17, 'Di': 15}
# Copeland picks: Bo
# Minimax picks: Ana
```

Ana beats Bo, Bo beats Di, Di beats Ana: a cycle, and Cy ties with Ana. Copeland gives Bo +1 (two wins, one loss), Ana 0, Cy 0 and Di -1, so Bo wins. Minimax asks about worst defeats: Bo's worst is the 17 to 7 loss to Ana, which is larger than Ana's worst, a 14 to 10 loss to Di, so Ana wins. Two reasonable pairwise rules, two different winners, from the same twenty-four ballots. The honest summary is that when there is a cycle there is no single correct answer: the rule is a choice about what to value, in this case a count of victories against the size of the worst loss.

How common are cycles?

Cycles sound exotic. Whether they matter depends on how voters are spread. The simulation below uses two assumed models, with 101 voters and 2,000 elections per row. In the "line" model, used in the last two chapters, voters and candidates sit at random points on a line and

rank by distance. In the “random ballots” model every ballot is an independent random ranking, which has no political structure at all.

```
import random

def pairwise(ballots):
    """wins[(x, y)] = number of ballots that rank x above y."""
    names = sorted(ballots[0])
    return {(x, y): sum(1 for b in ballots if b.index(x) < b.index(y))
            for x in names for y in names if x != y}

def condorcet(ballots):
    """The candidate who beats every rival by more than half the ballots, or None."""
    wins = pairwise(ballots)
    n = len(ballots)
    for x in sorted(ballots[0]):
        if all(wins[x, y] > n / 2 for y in ballots[0] if y != x):
            return x
    return None

random.seed(31)
trials = 2000
voters = 101
print("model            k  no Condorcet winner")
for model in ("line", "random ballots"):
    for k in (3, 4, 5):
        names = [f"C{i}" for i in range(k)]
        none = 0
        for _ in range(trials):
            if model == "line":
                pos = {c: random.random() for c in names}
                ballots = []
                for _ in range(voters):
                    v = random.random()
                    ballots.append(tuple(sorted(names, key=lambda c: abs(pos[c] - v))))
            else:
                ballots = [tuple(random.sample(names, k)) for _ in range(voters)]
            if condorcet(ballots) is None:
                none += 1
        print(f"{model:<15} {k}  {none / trials:6.1%}")
# model            k  no Condorcet winner
# line             3    0.0%
# line             4    0.0%
```

```
# line            5    0.0%
# random ballots  3    8.5%
# random ballots  4   17.7%
# random ballots  5   25.9%
```

On the line there are no cycles in these runs. That fits the earlier claim that a Condorcet winner exists with an odd number of voters on a line (I did not look up a proof). With random ballots cycles grow quickly with the field: 8.5%, 17.7% and 25.9% for three, four and five candidates. Neither model is real. The first has too little disagreement and the second too much, so real cycles are somewhere in between, and how often is a question for data, not for this simulation. A commonly quoted limit of about 8.8% for three candidates under random ballots is from memory (check).

Common mistakes

Comparing only the cell for X over Y and forgetting the mirror cell (they must add to the number of ballots when there are no ties); calling the candidate with the most pairwise wins a “Condorcet winner” when they have lost a contest; reading a Copeland tie as a bug; saying “Minimax” without saying which version; and treating the cycle rates of an assumed random-ballot model as the cycle rate in a real election.

EXERCISE 1

Twenty-five members rank five candidates (assumed numbers). Build and print the pairwise matrix, say whether a Condorcet winner exists, and compare that winner with the candidate who has the most first choices.

View solution

EXERCISE 2

Three groups of 12, 9 and 6 voters have rotated rankings of Ann, Ben and Cat (assumed numbers). Print every pairwise result, then compute the Copeland scores and the Minimax worst defeats. What does each method do with the cycle?

View solution

EXERCISE 3

Generate 5,000 seeded random profiles of four candidates (five groups of two to eight voters each). Count how many have no Condorcet winner and, among those, how many give a single Copeland winner that differs from the single Minimax winner.

View solution

QUICK REFERENCE

- **Pairwise matrix:** entry (X, Y) = number of ballots ranking X above Y
- **Condorcet winner:** beats every other candidate head to head by more than half the ballots
- **Condorcet cycle:** A beats B, B beats C, C beats A, so there may be no Condorcet winner
- **Copeland:** score = pairwise wins minus losses; highest score wins; ties are common in cycles
- **Minimax:** smallest worst defeat wins; say which version (winning votes, margins, opposition)
- **Python habit:** build the pairwise dictionary once, then compute every method from it
- **Always state:** how pairwise ties are scored, and which variant of a method you used

COMING NEXT

How Voting Systems Work 5 is about points and scores: the Borda count, approval and score voting, and how scoring rules reward different things.

How Voting Systems Work

Chapter 5 · Points and Scores

How Voting Systems Work 4 compared candidates two at a time. This chapter takes the other obvious route: give every candidate points and add them up. The oldest scheme of that kind is the **Borda count**, usually credited to the French mathematician Jean-Charles de Borda, who put it to the French Academy of Sciences in 1770 as a fairer alternative to plurality voting. Its rival is a family of rules where voters do not rank at all but approve or score. A story often told about Borda is that he said his scheme was meant only for honest men; I have not confirmed that quotation (check). It points at the real issue, though: once points are involved, the voter has a way to shade the points. The maths here is addition and the occasional fraction.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Facts about the methods come from two web searches whose results were mostly encyclopedia-style pages, a maths textbook and a voting-reform group's comparison chart; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase, not a quotation. The sources can differ on details, and I mark where. The elections, candidates, ballots and scores in the code are *assumed* or *illustrative*: invented to show a method, not results from any real vote. The simulation rests on a simple model that real politics does not follow. Statements marked (check) were not confirmed against a source.

Counting Places: the Borda Count

In the Borda count each voter ranks all k candidates, and a ballot gives k-1 points to the first choice, k-2 to the second, and so on down to 0 for last place. Add the points over all ballots; the highest total wins. With three candidates that is 2, 1 and 0. Here it is on the centre-squeeze profile that appeared in the last two chapters.

```
from collections import Counter

def borda(ballots):
    """Borda points: with k candidates, first place earns k-1 points and last place earn
    k = len(ballots[0])
    points = Counter()
```

```
    for ballot in ballots:
        for place, candidate in enumerate(ballot):
            points[candidate] += k - 1 - place
    return dict(sorted(points.items()))

# Illustrative numbers: the centre-squeeze profile from the last two chapters, invented
squeeze = (
    [("Ana", "Bo", "Cy")] * 35
    + [("Cy", "Bo", "Ana")] * 33
    + [("Bo", "Ana", "Cy")] * 16
    + [("Bo", "Cy", "Ana")] * 16
)
points = borda(squeeze)
print("first choices:", dict(sorted(Counter(b[0] for b in squeeze).items())))
print("Borda points: ", points)
print("Borda winner: ", max(points, key=points.get))
# first choices: {'Ana': 35, 'Bo': 32, 'Cy': 33}
# Borda points:  {'Ana': 86, 'Bo': 132, 'Cy': 82}
# Borda winner:  Bo
```

Bo gets 2 points from 32 first places and 1 point from 68 second places (35 ballots with Bo second behind Ana, 33 behind Cy): 64 + 68 = 132. Ana gets 2 x 35 + 1 x 16 = 86, and Cy gets 2 x 33 + 1 x 16 = 82. Bo, the candidate that pairwise comparison found and that neither runoff rule found, wins on the Borda count. A rule that rewards being ranked high by *everyone*, not first by a few, naturally favours the compromise candidate. The same trait is why Borda has a reputation for picking broadly acceptable candidates. It is not a guarantee, as the next section shows.

FEATURE	PLURALITY	BORDA COUNT	APPROVAL	SCORE VOTING
Ballot is	one name	a full ranking	a set of approved names	a number for every candidate
Winner is	most first choices	most rank points	most approvals	highest total (or average) score
Rewards	strong first-choice support	being ranked high by many	being acceptable to many	how much voters like a candidate, not just the order
Voter's main decision	who to vote for	the order of all candidates	where to draw the approval line	what number to give each candidate

When Points Mislead

The sources I found agree that the Borda count can fail the majority criterion (a candidate ranked first by more than half the voters should win), and that it can fail to elect a Condorcet winner. One search also gave a Borda failure of independence of irrelevant alternatives. The first of these is easy to build. Ten invented voters: six rank Ana, Bo, Cy; four rank Bo, Cy, Ana. Ana has 60% of the first choices and beats both rivals head to head, six ballots to four. The code also computes the **Dowdall** variant, which gives 1 point for first, 1/2 for second, 1/3 for third. Sources I found say it is used in Nauru, with those weights, to elect members in multi-member districts; here I apply it to a single winner only to show how the weights change the result.

```
from collections import Counter

def borda(ballots):
    """Borda points: with k candidates, first place earns k-1 points and last place earns
    k = len(ballots[0])
    points = Counter()
    for ballot in ballots:
        for place, candidate in enumerate(ballot):
            points[candidate] += k - 1 - place
    return dict(sorted(points.items()))

def dowdall(ballots):
    """Dowdall points: first place earns 1, second 1/2, third 1/3, and so on."""
    points = Counter()
    for ballot in ballots:
        for place, candidate in enumerate(ballot):
            points[candidate] += 1 / (place + 1)
    return {c: round(p, 2) for c, p in sorted(points.items())}

# Illustrative numbers: ten invented ballots in which one candidate has a majority of fi
ballots = [("Ana", "Bo", "Cy")] * 6 + [("Bo", "Cy", "Ana")] * 4
first = Counter(b[0] for b in ballots)
print("first choices:", dict(sorted(first.items())), "(10 ballots)")
for name, rule in (("Borda", borda), ("Dowdall", dowdall)):
    scores = rule(ballots)
    print(f"{name:<8}", scores, "->", max(scores, key=scores.get))
# first choices: {'Ana': 6, 'Bo': 4} (10 ballots)
# Borda    {'Ana': 12, 'Bo': 14, 'Cy': 4} -> Bo
```

```
# Dowdall  {'Ana': 7.33, 'Bo': 7.0, 'Cy': 4.0} -> Ana
```

Borda gives Bo 14 points and Ana 12, so Bo wins even though Ana was the first choice of a majority. Ana gets 2 points from each of the six who rank Ana first and 0 from the four who rank Ana last, 12 in all, while Bo gets 1 point from each of the six and 2 from each of the four, 6 + 8 = 14. The Dowdall weights drop faster after first place (1, 0.5, 0.33) than the evenly spaced Borda weights (2, 1, 0), so a first place counts for relatively more, and Ana wins with 7.33 to 7.00. Neither weighting is the correct one. They are different statements about how much second place is worth.

> Common mistakes
> Calling the Borda winner "the most popular" when it may have fewer first choices than anyone; forgetting that the point weights are a choice (the Dowdall weights are another, and sources describe further variants); mixing up k-1 points for first with k points for first (the winner is the same but the totals are not); and treating one invented example as proof that a method "always" behaves that way.

Adding a near-copy

Because Borda counts places, adding a candidate changes every other candidate's points, even a candidate who cannot win. Imagine nine voters choosing between Ana and Bo: five prefer Ana and four prefer Bo. Now a near-copy of Bo, called Bo2, joins the race. The voters who like Bo rank Bo first and Bo2 second; the Ana voters rank Bo2 last.

```
def borda(ballots):
    """Borda points: with k candidates, first place earns k-1 points and last place earns
    k = len(ballots[0])
    points = {c: 0 for c in ballots[0]}
    for ballot in ballots:
        for place, candidate in enumerate(ballot):
            points[candidate] += k - 1 - place
    return dict(sorted(points.items()))

# Illustrative numbers: nine invented ballots; then a near-copy of Bo ("Bo2") is added.
before = [("Ana", "Bo")] * 5 + [("Bo", "Ana")] * 4
after = [("Ana", "Bo", "Bo2")] * 5 + [("Bo", "Bo2", "Ana")] * 4
for label, ballots in (("two candidates", before), ("with a near-copy of Bo", after)):
    scores = borda(ballots)
```

```
    print(f"{label:<24}", scores, "->", max(scores, key=scores.get))
# two candidates           {'Ana': 5, 'Bo': 4} -> Ana
# with a near-copy of Bo   {'Ana': 10, 'Bo': 13, 'Bo2': 4} -> Bo
```

With two candidates Ana wins 5 to 4. With the near-copy added, nobody changed their mind about Ana versus Bo, but Bo now collects 2 points from each of the four supporters and 1 point (second place) from each of the five Ana voters, 8 + 5 = 13, while Ana stays on 5 x 2 = 10, and Bo wins. Plurality had the opposite weakness in How Voting Systems Work 2: a similar candidate *split* a vote. Here a similar candidate *adds* points. In both cases the number of candidates, not just the voters' opinions, shapes the outcome. This is one reason that anyone able to nominate extra candidates can influence a Borda election, and why How Voting Systems Work 6 looks at strategy more closely.

Approving and Scoring

Two further families drop the ranking. In **approval voting** each voter marks every candidate they approve of, with no limit, and the candidate with the most approvals wins. In **score voting** (also called range voting) each voter gives every candidate a number, for example 0 to 10, and the highest total or average wins. I recall some American cities having adopted approval voting for certain elections, with Fargo, North Dakota and St. Louis, Missouri named, but my searches did not confirm it (check). The sources I found say that, unlike ranked rules, approval and score voting can avoid some of Arrow's trouble with independence of irrelevant alternatives if voters score each candidate on their own fixed scale. They also say a candidate who would lose every one-on-one contest can still win under approval voting. I did not check either claim beyond those summaries.

The key to both is the voter's private decision. Below, seven invented voters give honest scores from 0 to 10. A score-voting count is simply the total. An approval count needs a rule for turning a score into yes or no, so the code tries four cut-offs: "approve anyone I scored at least 4", and so on.

```
# Illustrative numbers: seven invented voters score three candidates from 0 (hate) to 10
scores = {
    "Voter 1": {"Ana": 10, "Bo": 6, "Cy": 0},
```

```
    "Voter 2": {"Ana": 10, "Bo": 5, "Cy": 1},
    "Voter 3": {"Ana": 9, "Bo": 6, "Cy": 2},
    "Voter 4": {"Ana": 0, "Bo": 7, "Cy": 10},
    "Voter 5": {"Ana": 1, "Bo": 6, "Cy": 9},
    "Voter 6": {"Ana": 2, "Bo": 8, "Cy": 8},
    "Voter 7": {"Ana": 3, "Bo": 8, "Cy": 7},
}
names = ["Ana", "Bo", "Cy"]
top = {c: sum(1 for s in scores.values() if s[c] == max(s.values())) for c in names}
print("top-scored by:", top)
totals = {c: sum(s[c] for s in scores.values()) for c in names}
print("score totals: ", totals, "->", max(totals, key=totals.get))
for cutoff in (4, 6, 8, 9):
    approvals = {c: sum(1 for s in scores.values() if s[c] >= cutoff) for c in names}
    best = max(approvals.values())
    winners = [c for c in names if approvals[c] == best]
    print(f"approve at {cutoff}+:", approvals, "->", " and ".join(winners))
# top-scored by: {'Ana': 3, 'Bo': 2, 'Cy': 3}
# score totals:  {'Ana': 35, 'Bo': 46, 'Cy': 37} -> Bo
# approve at 4+: {'Ana': 3, 'Bo': 7, 'Cy': 4} -> Bo
# approve at 6+: {'Ana': 3, 'Bo': 6, 'Cy': 4} -> Bo
# approve at 8+: {'Ana': 3, 'Bo': 2, 'Cy': 3} -> Ana and Cy
# approve at 9+: {'Ana': 3, 'Bo': 0, 'Cy': 2} -> Ana
```

By score total Bo wins with 46 against 37 for Cy and 35 for Ana, though Bo is the top choice of only two voters (Voter 6 scored Bo and Cy equally, 8 each, which is why the top-scored counts add to eight, not seven). Under approval Bo wins when the line is generous, at 4+ or 6+, since nearly everyone finds Bo acceptable. At 8+ the voters approve only the candidates they love, Bo drops to two approvals, and Ana and Cy tie with three. At 9+ Ana wins alone. The ballots did not change; only where each voter draws the line did. That is the point to keep in mind: approval and score voting reward *how much* and *how widely* voters like a candidate, and the result depends on the voters' own sense of scale. Real voters would not all use the same cut-off, and some might exaggerate scores to help a favourite (an idea for How Voting Systems Work 6).

Which Rule Finds the Centre?

A simulation can compare how often each scoring rule picks the Condorcet winner. It uses the "line" model from earlier: four candidates and 101 voters at random points on a line, with voters

ranking by distance. All 2,000 elections had a Condorcet winner, as in How Voting Systems Work 4. The approval rule is that each voter approves candidates within 0.25 of their own position (an arbitrary choice). Score voting gives each candidate a score of minus the distance from the voter, which is an honest and fully consistent scale, a strong assumption.

```
import random
from collections import Counter

def pairwise(ballots):
    names = sorted(ballots[0])
    return {(x, y): sum(1 for b in ballots if b.index(x) < b.index(y))
            for x in names for y in names if x != y}

def condorcet(ballots):
    wins, n = pairwise(ballots), len(ballots)
    for x in ballots[0]:
        if all(wins[x, y] > n / 2 for y in ballots[0] if y != x):
            return x
    return None

random.seed(31)
trials, voters, k = 2000, 101, 4
names = [f"C{i}" for i in range(k)]
agree = Counter()
found = 0
for _ in range(trials):
    pos = {c: random.random() for c in names}
    ups = [random.random() for _ in range(voters)]
    ballots = [tuple(sorted(names, key=lambda c: abs(pos[c] - v))) for v in ups]
    cw = condorcet(ballots)
    if cw is None:
        continue
    found += 1
    plural = Counter(b[0] for b in ballots)
    borda = Counter()
    for b in ballots:
        for place, c in enumerate(b):
            borda[c] += k - 1 - place
    approval = Counter(c for v in ups for c in names if abs(pos[c] - v) <= 0.25)
    score = {c: -sum(abs(pos[c] - v) for v in ups) for c in names}
    for label, tally in (("plurality", plural), ("Borda", borda), ("approval (within 0.2
        if max(tally, key=tally.get) == cw:
            agree[label] += 1
```

```
print(f"elections with a Condorcet winner: {found} of {trials}")
for label in ("plurality", "Borda", "approval (within 0.25)", "score (nearness)"):
    print(f"{label:<24} picks it {agree[label] / found:6.1%}")
# elections with a Condorcet winner: 2000 of 2000
# plurality                picks it  49.6%
# Borda                    picks it  84.4%
# approval (within 0.25)   picks it  57.2%
# score (nearness)         picks it  95.9%
```

Plurality finds the Condorcet winner about half the time in this model (49.6%), Borda about 84%, a fixed-width approval rule about 57%, and score voting by nearness about 96%. Take that with care. The line model is far tidier than any electorate; the gap between score voting and the others depends on every voter using the same honest scale, and the approval figure moves if the 0.25 width changes. The lesson is not “score voting is best”. It is that a scoring rule keeps information a plain first-choice count throws away, and whether that information is reliable depends on how voters behave.

EXERCISE 1

Twenty-five members rank five candidates (assumed numbers). Compute the plurality, Borda and Dowdall scores, name each method’s winner, and say whether all three agree.

View solution

EXERCISE 2

Eight voters score three candidates from 0 to 10 (assumed numbers). Find the score-voting winner by total, then run approval voting at every cut-off from 1 to 10 and list the cut-offs at which approval does not give the score winner alone.

View solution

EXERCISE 3

Run 3,000 seeded elections of five candidates and 51 voters under two assumed models (the line and random ballots). When a Condorcet winner exists, count how often plurality and how often Borda pick it, and print the share of elections that have one.

View solution

QUICK REFERENCE

- **Borda count:** k-1 points for first, down to 0 for last; the most points wins
- **Dowdall:** 1, 1/2, 1/3, ... for first, second, third; a Borda variant with a steeper drop
- **Approval voting:** mark any number of candidates; the most approvals wins
- **Score voting:** give each candidate a number; the highest total or average wins
- **Borda weakness:** can elect someone with no majority of first choices, and an added near-copy can change the winner
- **Approval and score weakness:** the result depends on each voter's own cut-off or scale
- **Always state:** the point weights, the approval cut-off or score range, and how ties are broken

COMING NEXT

How Voting Systems Work 6 is about paradoxes and impossibility: Arrow's theorem, strategic voting, and testing voting methods against fairness criteria by code.

How Voting Systems Work

Chapter 6 · Paradoxes and Limits

Every method in How Voting Systems Work 2 to 5 had a weak spot: vote splitting for plurality, the centre squeeze for runoffs, cycles for pairwise contests, easy point-shading for Borda. It is natural to ask whether a cleverer rule could have none of them. The mathematical answer, for ranked ballots, is a discouraging "not quite". This chapter looks at the two best-known results, **Arrow's theorem** (about fairness) and the **Gibbard–Satterthwaite theorem** (about honesty), and then does something more useful than quoting them: it writes small programs that test each method against named fairness criteria and count how often it fails. The maths stays at counting and comparing.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Facts about the theorems come from two web searches whose results were mostly lecture notes, encyclopedia mirrors and an advocacy group's article; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase, not a quotation. The sources broadly agree on the theorems; one search result stated the list of Arrow's conditions only partly, so that list is from general knowledge (check), and the advocacy source argues that one failure matters little, which I mark below as one side of a dispute. The ballots in the code are *assumed* or *illustrative*, and the random elections use a model (every ballot equally likely) that real politics does not follow. Statements marked (check) were not confirmed against a source.

The Theorems in Plain Words

Arrow's theorem, published by the economist Kenneth Arrow in his 1951 book *Social Choice and Individual Values*, concerns rules that turn everyone's full rankings of three or more options into one ranking for the group. In the usual statement, no such rule satisfies all of these at once: it works for every possible set of ballots (*unrestricted domain*); if everyone prefers X to Y then the group does (*unanimity*, or Pareto efficiency); the group's order of X and Y depends only on how voters ordered X and Y (*independence of irrelevant alternatives*, IIA); and no single voter always decides (*non-dictatorship*). The sources I found agree on the result and the 1951 date; the exact wording of the four conditions varies between textbooks. They also note the theorem can be avoided by restricting the ballots allowed, for example to preferences that follow a single left-right line.

The **Gibbard–Satterthwaite theorem** (credited to Allan Gibbard in 1973 and Mark Satterthwaite in 1975) is about honesty. Roughly: with three or more possible winners, any rule that can elect any of them and is not a dictatorship will sometimes give some voter a reason to put down a ballot that is not their true ranking. Neither theorem says every method is equally bad, or that failures are common. They say a perfect method cannot exist. How often and how badly each one fails is a question for counting, which is what the rest of this chapter does.

IDEA	PLAIN MEANING	WHERE WE MET IT
Majority criterion	someone ranked first by more than half the voters should win	How Voting Systems Work 5 (Borda fails)
Condorcet criterion	a candidate who beats every rival one-on-one should win	How Voting Systems Work 4
Monotonicity	ranking a winner higher should never make it lose	this chapter
IIA	the group's order of two options depends only on voters' order of those two	How Voting Systems Work 2 and 5 (near-copies)
Strategy-proofness	no voter ever gains by lying	this chapter

Independence: Test It by Code

A criterion becomes testable once you state it as a check on two profiles. For IIA: take a profile, change only where one third candidate (Cy) appears on some ballots, so that no voter's Ana-versus-Bo order moves, and see whether the method's verdict on Ana versus Bo flips. The code below does that for the Borda count with five invented voters. It tries every way of moving Cy on the two ballot types and prints the first one that flips the verdict.

```
from itertools import product

def borda(ballots):
    """Borda points: with k candidates, first place earns k-1 points and last place earns
    k = len(ballots[0])
    points = {c: 0 for c in ballots[0]}
    for ballot in ballots:
        for place, c in enumerate(ballot):
            points[c] += k - 1 - place
```

```
    return points

def move(ballot, who, slot):
    """Return the ballot with `who` taken out and re-inserted at position `slot`."""
    rest = [c for c in ballot if c != who]
    rest.insert(slot, who)
    return tuple(rest)

# Illustrative numbers: five invented voters in two groups. Only Cy moves; Ana versus Bo
groups = [(3, ("Ana", "Bo", "Cy")), (2, ("Bo", "Cy", "Ana"))]
before = [b for n, b in groups for _ in range(n)]
print("before:", borda(before), "-> Ana vs Bo winner:", "Ana" if borda(before)["Ana"] > b
for slots in product(range(3), repeat=len(groups)):
    after = [move(b, "Cy", s) for (n, b), s in zip(groups, slots) for _ in range(n)]
    pts = borda(after)
    if (pts["Ana"] > pts["Bo"]) != (borda(before)["Ana"] > borda(before)["Bo"]):
        print("after: ", pts, "-> Ana vs Bo winner:", "Ana" if pts["Ana"] > pts["Bo"] el
        break
# before: {'Ana': 6, 'Bo': 7, 'Cy': 2} -> Ana vs Bo winner: Bo
# after:  {'Cy': 10, 'Ana': 3, 'Bo': 2} -> Ana vs Bo winner: Ana | Cy slots: (0, 0)
```

Before, Bo wins the Borda count 7 to 6 over Ana. Then Cy is moved to the top of both kinds of ballot, and the order of Ana and Bo in each ballot is unchanged (three voters still prefer Ana to Bo, two still prefer Bo). Yet Ana now beats Bo 3 to 2 on points. The voters said nothing new about Ana or Bo; the points changed because Cy's position changed. That is an IIA failure. The Borda count is not alone: one search result said plurality and ranked runoffs fail IIA as well, which How Voting Systems Work 2 showed in the vote-splitting form (check).

Strategic Voting

The oldest tactic is the **compromise vote** under plurality. Imagine 100 invented voters: 40 prefer Ana, 35 prefer Bo, 25 prefer Cy and would rank Bo second. If everyone is honest Ana wins. If the 25 move to Bo, Bo wins.

```
from collections import Counter

def plurality(ballots):
```

```
    tally = Counter(b[0] for b in ballots)
    return max(sorted(tally), key=tally.get)

# Illustrative numbers: 100 invented voters. Cy cannot win; Ana and Bo are close.
honest = ([("Ana", "Bo", "Cy")] * 40 + [("Bo", "Ana", "Cy")] * 35 + [("Cy", "Bo", "Ana")
print("honest tally:   ", dict(sorted(Counter(b[0] for b in honest).items())), "->", plu
# The 25 Cy voters prefer Bo to Ana. If they vote for Bo instead of Cy ("compromise"):
shifted = ([("Ana", "Bo", "Cy")] * 40 + [("Bo", "Ana", "Cy")] * 35 + [("Bo", "Cy", "Ana"
print("compromise vote:", dict(sorted(Counter(b[0] for b in shifted).items())), "->", pl
# Is that better for those 25 voters by their own honest ranking?
rank = ("Cy", "Bo", "Ana")
print("their view:", rank.index(plurality(honest)), "(place of honest winner) vs", rank.
# honest tally:    {'Ana': 40, 'Bo': 35, 'Cy': 25} -> Ana
# compromise vote: {'Ana': 40, 'Bo': 60} -> Bo
# their view: 2 (place of honest winner) vs 1 (place of new winner); lower is better
```

The numbers are zero-based places: the honest winner, Ana, was last on the 25 Cy voters' true ballots (place 2), and Bo, their second choice, is place 1. By lying, the 25 voters swapped an outcome they liked least for one they liked more. Unless they are sure of the other voters' plans this is a gamble, and a poll can persuade many people to make the same move. Another tactic, **burying**, is to rank a dangerous rival last on purpose; the exercises below try it under the Borda count.

> Common mistakes
> Reading the theorems as "all voting systems are equally bad"; treating one invented example as the rate at which a method fails; forgetting that an alphabetical tie-break (used in the code) is itself a rule that can decide results; and assuming a method that is hard to manipulate by one voter is also hard for a coordinated group.

Counting Failures

Now the whole test bench. The code draws 3,000 random elections of nine voters and three candidates, each ballot chosen uniformly from the six possible orders. When a Condorcet winner exists, it checks whether plurality, Borda and instant-runoff elected that candidate; when some candidate has a majority of first choices it checks the same. Ties are broken alphabetically, which is an assumption of this chapter.

```
import random
from collections import Counter
from itertools import permutations

def plurality(ballots):
    tally = Counter(b[0] for b in ballots)
    return max(sorted(tally), key=tally.get)

def borda(ballots):
    k = len(ballots[0])
    points = {c: 0 for c in sorted(ballots[0])}
    for b in ballots:
        for place, c in enumerate(b):
            points[c] += k - 1 - place
    return max(points, key=points.get)

def irv(ballots):
    alive = set(ballots[0])
    while True:
        tally = Counter({c: 0 for c in alive})
        for b in ballots:
            tally[next(c for c in b if c in alive)] += 1
        top = max(sorted(tally), key=tally.get)
        if tally[top] * 2 > len(ballots) or len(alive) == 1:
            return top
        alive.remove(min(sorted(tally), key=tally.get))

methods = {"plurality": plurality, "Borda": borda, "instant-runoff": irv}
random.seed(6)
names, voters, trials = ("Ana", "Bo", "Cy"), 9, 3000
orders = list(permutations(names))
fails = Counter()
has = Counter()
for _ in range(trials):
    ballots = [random.choice(orders) for _ in range(voters)]
    # Condorcet winner, if any
    cw = next((x for x in names if all(sum(b.index(x) < b.index(y) for b in ballots) * 2
                                       for y in names if y != x)), None)
    first = Counter(b[0] for b in ballots)
    maj = next((c for c, n in first.items() if n * 2 > voters), None)
    has['Condorcet'] += cw is not None
    has['majority'] += maj is not None
    for label, rule in methods.items():
```

```
            w = rule(ballots)
            if cw and w != cw:
                fails[label, "Condorcet"] += 1
            if maj and w != maj:
                fails[label, "majority"] += 1
    for crit in ("Condorcet", "majority"):
        print(f"{crit} criterion: {has[crit]} of {trials} elections have such a winner; failu
    # Condorcet criterion: 2775 of 3000 elections have such a winner; failures: {'plurality'
    # majority criterion: 1298 of 3000 elections have such a winner; failures: {'plurality':
```

A Condorcet winner existed in 2,775 of 3,000 elections. Plurality missed it in 514 of those (about 18.5%), Borda in 227 (about 8.2%) and instant-runoff in 109 (about 3.9%). A first-choice majority winner existed in 1,298 elections; plurality and instant-runoff always chose it, and Borda missed it in 50 (about 3.9%). That matches the textbook picture: plurality and runoffs meet the majority criterion, Borda does not, and none of the three always meets the Condorcet criterion. But the percentages belong to this random-ballot model, in which voters have no shared political shape at all. Real electorates are more structured, and the rates would change.

One Voter Lying

The Gibbard–Satterthwaite theorem says opportunities to gain by lying exist. This simulation measures how often a single voter could use one. For each of 1,000 random elections of seven voters and three candidates, it tries every possible ballot for every voter, keeping the others fixed, and asks whether any lie leads to a winner that voter truly ranks higher.

```
import random
from collections import Counter
from itertools import permutations

def plurality(ballots):
    tally = Counter(b[0] for b in ballots)
    return max(sorted(tally), key=tally.get)

def borda(ballots):
    k = len(ballots[0])
    points = {c: 0 for c in sorted(ballots[0])}
    for b in ballots:
```

```
            for place, c in enumerate(b):
                points[c] += k - 1 - place
        return max(points, key=points.get)

    def irv(ballots):
        alive = set(ballots[0])
        while True:
            tally = Counter({c: 0 for c in alive})
            for b in ballots:
                tally[next(c for c in b if c in alive)] += 1
            top = max(sorted(tally), key=tally.get)
            if tally[top] * 2 > len(ballots) or len(alive) == 1:
                return top
            alive.remove(min(sorted(tally), key=tally.get))

    def can_gain(rule, ballots):
        """True if some single voter can get a better winner (by their true ranking) by lyin
        honest = rule(ballots)
        for i, truth in enumerate(ballots):
            for lie in permutations(truth):
                trial = ballots[:i] + [lie] + ballots[i + 1:]
                if truth.index(rule(trial)) < truth.index(honest):
                    return True
        return False

    random.seed(6)
    names, voters, trials = ("Ana", "Bo", "Cy"), 7, 1000
    orders = list(permutations(names))
    count = Counter()
    for _ in range(trials):
        ballots = [random.choice(orders) for _ in range(voters)]
        for label, rule in (("plurality", plurality), ("Borda", borda), ("instant-runoff", i
            count[label] += can_gain(rule, ballots)
    for label in ("plurality", "Borda", "instant-runoff"):
        print(f"{label:<15} a lone voter can gain by lying in {count[label] / trials:5.1%} o
    # plurality       a lone voter can gain by lying in 25.3% of 1000 elections
    # Borda           a lone voter can gain by lying in 24.3% of 1000 elections
    # instant-runoff  a lone voter can gain by lying in 11.2% of 1000 elections
```

Plurality had a profitable lie in 25.3% of elections, Borda in 24.3%, and instant-runoff in 11.2%. Instant-runoff did better here, in line with a source I found saying it resists some tactics such as burying, but note what the test counts: only a lone voter, only three candidates, only seven

voters. Coordinated groups, more candidates and strategies that need guessing others' votes are outside it. The result is a measurement of one model, not a verdict on any system.

Monotonicity: a stranger failure

Instant-runoff has a quirk that none of the others shows here: it can fail monotonicity, meaning that a candidate can lose *because* some voters ranked them higher. Sources I found agree on the failure. The advocacy group FairVote argues it matters little in practice and that other properties matter more; that is one side of a real dispute, and I did not check how often the failure arises in real elections (check). The code searches random 21-voter elections for one, skipping any profile where a tie for last place would need a tie-break, and then lifts the winner one place on more and more of the ballots that do not rank it first.

```
import random
from collections import Counter
from itertools import permutations

def irv(ballots):
    """Instant-runoff winner, or None if a tie for last place has to be broken."""
    alive = set(ballots[0])
    while True:
        tally = Counter({c: 0 for c in alive})
        for b in ballots:
            tally[next(c for c in b if c in alive)] += 1
        top = max(tally.values())
        if top * 2 > len(ballots) or len(alive) == 1:
            return next(c for c in tally if tally[c] == top)
        low = min(tally.values())
        losers = [c for c in tally if tally[c] == low]
        if len(losers) > 1:
            return None
        alive.remove(losers[0])

def raise_one(ballot, who):
    """Move `who` up one place on this ballot."""
    i = ballot.index(who)
    b = list(ballot)
    b[i - 1], b[i] = b[i], b[i - 1]
    return tuple(b)

# Illustrative: random 3-candidate profiles (an assumed model); ties are skipped, not br
random.seed(6)
```

```
names, voters = ("Ana", "Bo", "Cy"), 21
orders = list(permutations(names))
found = None
for _ in range(50000):
    ballots = [random.choice(orders) for _ in range(voters)]
    w = irv(ballots)
    if w is None:
        continue
    movable = [i for i, b in enumerate(ballots) if b[0] != w]
    for r in range(1, len(movable) + 1):
        trial = [raise_one(b, w) if i in movable[:r] else b for i, b in enumerate(ballot
        result = irv(trial)
        if result is not None and result != w:
            found = (ballots, w, r, trial, result)
            break
    if found:
        break
if found:
    ballots, w, r, trial, result = found
    print("ballots:", dict(sorted(Counter(ballots).items())))
    print("winner before:", w)
    print(f"{r} ballots lift {w} one place -> new winner:", result)
    print("first choices before:", dict(sorted(Counter(b[0] for b in ballots).items())))
    print("first choices after: ", dict(sorted(Counter(b[0] for b in trial).items())))
else:
    print("no failure found")
# ballots: {('Ana', 'Bo', 'Cy'): 1, ('Ana', 'Cy', 'Bo'): 5, ('Bo', 'Ana', 'Cy'): 5, ('Bo
# winner before: Cy
# 2 ballots lift Cy one place -> new winner: Ana
# first choices before: {'Ana': 6, 'Bo': 7, 'Cy': 8}
# first choices after:  {'Ana': 6, 'Bo': 5, 'Cy': 10}
```

Before, Ana is eliminated first (6 first choices against 7 and 8). Ana's five "Ana, Cy, Bo" ballots move to Cy, so Cy wins with 13. After two ballots lift Cy a place, so that Cy has 10 first choices, Bo (5) is eliminated instead, Bo's remaining five voters all rank Ana next, and Ana wins 11 to 10. Cy gained support and lost. The cause is that the candidate who is removed first changes with the first-choice counts, so extra first choices can move who is eliminated and where those votes then go.

EXERCISE 1

Thirteen voters rank four candidates (assumed numbers). For plurality, Borda and instant-runoff, find the winner, then drop each other candidate in turn and report whether the winner changes. Which methods pass this independence test on this profile?

View solution

EXERCISE 2

Nine voters rank three candidates (assumed numbers). Show that under the Borda count one voter can gain by reordering their own ballot. Print the voter's lie and the points before and after, and say what the tie-break did.

View solution

EXERCISE 3

Run 3,000 seeded elections of nine voters and three candidates with random ballots. In each, lift the winner to first place on more and more of the ballots that do not rank it first, and count the elections where the winner then loses, for plurality, Borda and instant-runoff.

View solution

QUICK REFERENCE

- **Arrow's theorem:** for three or more options, no ranked rule has unrestricted domain, unanimity, IIA and non-dictatorship together (1951)
- **Gibbard–Satterthwaite:** a non-dictatorial rule with three or more possible winners lets some voter gain by lying (1973 and 1975)
- **Majority criterion:** plurality and instant-runoff satisfy it; the Borda count does not
- **Condorcet criterion:** plurality, Borda and instant-runoff all can fail it
- **Monotonicity:** instant-runoff can fail it; plurality did not in the test
- **Compromise vote and burying:** the plurality and Borda tactics shown above
- **Always state:** the ballot model, number of voters and candidates, and the tie-break rule when quoting a failure rate

COMING NEXT

How Voting Systems Work 7 is about dividing seats: apportionment, the Hamilton, Jefferson, Webster and Huntington-Hill methods, and the Alabama paradox.

How Voting Systems Work

Chapter 7 · Dividing Seats

How Voting Systems Work 1 to 6 asked how to pick a winner from ballots. A different question comes up whenever a fixed number of seats must be shared between groups of unequal size: states in a federation, regions in a parliament, parties after a list vote. This is **apportionment**. It sounds like simple division, and it would be if people came in fractions. The trouble is that a state cannot have 5.2 seats, so some rounding rule must decide, and every rounding rule has a character. The maths stays at division and comparison; the Python does the bookkeeping.

> **WHAT WAS RUN, AND WHAT TO TRUST**
>
> Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Historical and theoretical claims come from two web searches whose results were encyclopedia mirrors, a think-tank article, a maths-society column and papers; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. **The sources conflict on dates**: some place the Alabama paradox in 1880, others in 1882; they differ on when each method was in use in the United States (one says Webster's method was adopted in 1842 and replaced by Hamilton's in 1852, others give 1900 to 1901 or the 1910 census), so I give only the dates they agree on. All populations in the code are *invented* or drawn from an *assumed* random model, not real census figures. Statements marked (check) were not confirmed against a source.

The Quota and the Rounding Problem

The fair share of a state is its **quota**: the number of seats times the state's population, divided by the total population. If four invented states hold 5,200, 2,700, 1,300 and 800 people and the house has 10 seats, the total is 10,000 and the quotas are 5.2, 2.7, 1.3 and 0.8. Those add up to 10, but you cannot hand out 0.8 of a seat. Each state is also normally guaranteed at least one seat (true of the United States House; check for other systems), which already pulls against pure proportion.

The methods split into two families. **Quota methods** (Hamilton) start from the whole part of each quota and hand out leftovers. **Divisor methods** (Jefferson, Webster, Huntington-Hill) pick a divisor, divide every population by it, round, and adjust the divisor until the rounded numbers sum to the seat total. The only difference between the divisor methods is how they round.

METHOD	FAMILY	ROUNDING IDEA	TENDS TO FAVOUR
Hamilton	quota (largest remainder)	whole part of quota, then biggest fractions get the spare seats	no steady bias, but paradoxes
Jefferson	divisor	always round down	large states
Webster	divisor	round to the nearest whole number	nearly neutral
Huntington-Hill	divisor	round up if the quotient is above the geometric mean of the two nearest whole numbers	small states (slightly)

One Example, Four Methods

The code below implements all four. The divisor methods are written in the "one seat at a time" form: every state has a priority `p / f(n)`, where `p` is its population and `n` the seats it holds, and the best priority takes the next seat. The function `f` is what differs: `n + 1` gives Jefferson, `n + 0.5` gives Webster, and `sqrt(n(n+1))` gives Huntington-Hill. That square root is the geometric mean of `n` and `n + 1`. It is zero when `n` is zero, so Huntington-Hill must give every state one seat to start with. As far as I know the one-seat-at-a-time form gives the same result as the divisor form (check), apart from tie-breaks; here ties go to the alphabetically first name.

```
from math import floor, sqrt

def quotas(pops, seats):
    """Each state's exact share of the seats: seats * population / total."""
    total = sum(pops.values())
    return {s: seats * p / total for s, p in pops.items()}

def hamilton(pops, seats):
    """Largest remainder: everyone gets the whole part of the quota, leftovers go to the
    q = quotas(pops, seats)
    alloc = {s: floor(v) for s, v in q.items()}
    left = seats - sum(alloc.values())
    for s in sorted(q, key=lambda s: q[s] - alloc[s], reverse=True)[:left]:
        alloc[s] += 1
    return alloc

def divisor(pops, seats, priority, start=0):
```

```
    """Highest-averages form of a divisor method: hand out seats one at a time to the be
    alloc = {s: start for s in pops}
    for _ in range(seats - start * len(pops)):
        best = max(sorted(pops), key=lambda s: priority(pops[s], alloc[s]))
        alloc[best] += 1
    return alloc

def jefferson(pops, seats):
    return divisor(pops, seats, lambda p, n: p / (n + 1))

def webster(pops, seats):
    return divisor(pops, seats, lambda p, n: p / (n + 0.5))

def huntington_hill(pops, seats):
    return divisor(pops, seats, lambda p, n: p / sqrt(n * (n + 1)), start=1)

# Illustrative numbers: four invented states and a 10-seat house.
pops = {"Ardan": 5200, "Brex": 2700, "Cole": 1300, "Dunn": 800}
print("quota        ", {s: round(v, 2) for s, v in quotas(pops, 10).items()})
for name, rule in (("Hamilton", hamilton), ("Jefferson", jefferson), ("Webster", webster
    print(f"{name:<16}", rule(pops, 10))
# quota         {'Ardan': 5.2, 'Brex': 2.7, 'Cole': 1.3, 'Dunn': 0.8}
# Hamilton         {'Ardan': 5, 'Brex': 3, 'Cole': 1, 'Dunn': 1}
# Jefferson        {'Ardan': 6, 'Brex': 3, 'Cole': 1, 'Dunn': 0}
# Webster          {'Ardan': 5, 'Brex': 3, 'Cole': 1, 'Dunn': 1}
# Huntington-Hill  {'Ardan': 5, 'Brex': 3, 'Cole': 1, 'Dunn': 1}
```

Jefferson's round-down habit hands Ardan 6 seats against a quota of 5.2 and leaves Dunn with none, though Dunn holds 8% of the people. Hamilton, Webster and Huntington-Hill all agree on 5, 3, 1, 1 here; one example shows the shape of the differences, not their frequency.

A Little History

The sources I found agree on the following outline. Hamilton's largest-remainder method was passed by Congress but vetoed by Washington (the first presidential veto, if I recall correctly; check), after which Jefferson's method was used for decades. Webster's method and Hamilton's each had periods of use; sources disagree on the years, as the tip above says. Hamilton's method was abandoned for the states, one source says in 1901, after it produced paradoxes. The House was fixed at 435 seats, and Huntington-Hill, proposed by the Census Bureau's Joseph Hill

and refined by the mathematician Edward Huntington, has been used since the 1941 apportionment. Sources note that some critics still prefer Webster's method.

The Alabama Paradox

The paradox is named for a case in which, sources say, increasing the House from 299 to 300 seats would have cut Alabama from 8 seats to 7 (the discovery is dated 1880 by some sources and 1882 by others). More seats and a state loses one: it sounds impossible, but Hamilton's method decides by remainders, and a bigger house changes every remainder. The code below recreates the effect with three invented states, growing the house one seat at a time.

```
from math import floor

def hamilton(pops, seats):
    total = sum(pops.values())
    q = {s: seats * p / total for s, p in pops.items()}
    alloc = {s: floor(v) for s, v in q.items()}
    left = seats - sum(alloc.values())
    for s in sorted(q, key=lambda s: q[s] - alloc[s], reverse=True)[:left]:
        alloc[s] += 1
    return alloc

# Illustrative numbers: three invented states. We grow the house one seat at a time and w
pops = {"Ana": 6000, "Bo": 6000, "Cy": 2000}
previous = hamilton(pops, 2)
for size in range(3, 16):
    now = hamilton(pops, size)
    for s in pops:
        if now[s] < previous[s]:
            print(f"house {size - 1} -> {size}: {s} falls from {previous[s]} to {now[s]}
            print(f"  size {size - 1}:", previous)
            print(f"  size {size}:", now)
    previous = now
# house 3 -> 4: Cy falls from 1 to 0
#   size 3: {'Ana': 1, 'Bo': 1, 'Cy': 1}
#   size 4: {'Ana': 2, 'Bo': 2, 'Cy': 0}
# house 10 -> 11: Cy falls from 2 to 1
#   size 10: {'Ana': 4, 'Bo': 4, 'Cy': 2}
```

```
#   size 11: {'Ana': 5, 'Bo': 5, 'Cy': 1}
```

At 3 seats the quotas are about 1.29, 1.29 and 0.43. The whole parts give 1 seat each to Ana and Bo; the last seat goes to the biggest fraction, which is Cy's 0.43. At 4 seats the quotas are about 1.71, 1.71 and 0.57, the whole parts are 1, 1 and 0, and the two spare seats go to the biggest fractions, 0.71 and 0.71, which belong to Ana and Bo. Cy's quota grew from 0.43 to 0.57 and its seat still vanished, because the other two fractions grew faster.

> Common mistakes
> Treating the example above as typical when it was built to fail; assuming Jefferson's bias means it is "rigged" (it is a rounding rule with a predictable tilt, which is a different thing); forgetting that tie-breaks and the one-seat minimum are rules too; and mixing up the Alabama paradox (a bigger *house* costs a state a seat) with the population paradox (a state that grows *faster* loses a seat to one that grows slower), which Exercise 3 hunts for.

Why There Is No Perfect Method

Hamilton's method always keeps each state within its quota rounded down or up (the **quota rule**), but it can suffer the Alabama paradox and the population paradox, and, sources note, the new-states paradox. The divisor methods avoid those paradoxes but can violate the quota rule; Exercise 2 finds a Jefferson violation. The **Balinski–Young impossibility theorem** (1982, per the sources I found) says that for four or more states no method can always satisfy the quota rule and avoid the population paradox. A recent paper adds a way round it: randomised methods can satisfy both ideas in expectation, which a deterministic rule cannot. Which flaw to tolerate is a value judgement, not something mathematics settles.

Measuring the Tilt

Is Jefferson really biased towards large states? The simulation below draws 15 states with an *assumed* spread of populations (lognormal, which means a few large states and many smaller ones), shares 60 seats, and records how far the largest and smallest state land from their quotas on average, over 2,000 seeded trials.

```python
import random
from math import floor, sqrt

def divisor(pops, seats, priority, start=0):
    alloc = {s: start for s in pops}
    for _ in range(seats - start * len(pops)):
        best = max(sorted(pops), key=lambda s: priority(pops[s], alloc[s]))
        alloc[best] += 1
    return alloc

def hamilton(pops, seats):
    total = sum(pops.values())
    q = {s: seats * p / total for s, p in pops.items()}
    alloc = {s: floor(v) for s, v in q.items()}
    left = seats - sum(alloc.values())
    for s in sorted(q, key=lambda s: q[s] - alloc[s], reverse=True)[:left]:
        alloc[s] += 1
    return alloc

methods = {
    "Hamilton": hamilton,
    "Jefferson": lambda p, k: divisor(p, k, lambda x, n: x / (n + 1)),
    "Webster": lambda p, k: divisor(p, k, lambda x, n: x / (n + 0.5)),
    "Huntington-Hill": lambda p, k: divisor(p, k, lambda x, n: x / sqrt(n * (n + 1)), st
}

# Assumed model: 15 states with populations drawn from a lognormal spread, 60 seats, 2,0
random.seed(7)
seats, trials = 60, 2000
gap = {m: [0.0, 0.0] for m in methods}   # [largest state, smallest state]: average (sea
for _ in range(trials):
    pops = {f"S{i:02d}": int(random.lognormvariate(0, 1) * 1000) + 1 for i in range(15)}
    total = sum(pops.values())
    big, small = max(pops, key=pops.get), min(pops, key=pops.get)
    for m, rule in methods.items():
        a = rule(pops, seats)
        gap[m][0] += a[big] - seats * pops[big] / total
        gap[m][1] += a[small] - seats * pops[small] / total
for m in methods:
    print(f"{m:<16} largest state: {gap[m][0] / trials:+.3f} seats vs quota   smallest: 
# Hamilton         largest state: +0.001 seats vs quota   smallest: -0.078
# Jefferson        largest state: +1.429 seats vs quota   smallest: -0.414
# Webster          largest state: -0.003 seats vs quota   smallest: -0.061
```

```
# Huntington-Hill  largest state: -0.410 seats vs quota   smallest: +0.488
```

On average Jefferson gave the largest state about 1.4 seats more than its quota. Webster and Hamilton sat almost exactly on the quota for the largest state, and Huntington-Hill gave the largest state about 0.4 seats less than its quota and the smallest 0.49 more. That last number is partly the one-seat minimum: with 60 seats and 15 unequal states, the smallest state often has a quota well under one seat but still gets one. The figures belong to this model; a real country's population spread would change them, and the direction of each tilt is the lesson, not the size.

EXERCISE 1

Five invented states share 12 seats under Huntington-Hill. Start each state with one seat, then print each remaining seat as it is awarded with the winning priority and the runner-up's. Which seat was closest?

View solution

EXERCISE 2

Search seeded random profiles of five states and 10 seats for a case where Jefferson's method gives a state more than its quota rounded up (a quota-rule violation). Print the populations, the quota and the allocation.

View solution

EXERCISE 3

Search random four-state profiles with 20 seats for a population paradox under Hamilton's method: state X grows by a larger percentage than state Y, yet X loses a seat while Y gains one.

View solution

QUICK REFERENCE

- **Quota:** seats × state population ÷ total population

- **Hamilton:** whole parts first, spare seats to the largest fractions; can show the Alabama and population paradoxes
- **Jefferson:** priority p / (n + 1); rounds down; favours large states
- **Webster:** priority p / (n + 0.5); rounds to nearest
- **Huntington-Hill:** priority p / sqrt(n(n + 1)); every state starts with one seat; used for the US House since 1941
- **Alabama paradox:** a larger house costs a state a seat
- **Balinski–Young:** no deterministic method always meets the quota rule and avoids the population paradox (four or more states)
- **Always state:** the populations, seat total, tie-break and minimum-seat rule when quoting a result

COMING NEXT

How Voting Systems Work 8 is about districts and gerrymandering: drawing boundaries, packing and cracking, and measuring compactness and the efficiency gap.

How Voting Systems Work

Chapter 8 · Districts and Gerrymandering

How Voting Systems Work 7 shared a fixed number of seats between regions. This chapter asks the follow-up question: once a region is entitled to several seats and elects them from single-member **districts**, who decides where the district lines go? The answer matters more than most people expect, because the same voters can produce very different results depending on how they are grouped. Drawing lines to tilt the result on purpose is called **gerrymandering**. The maths stays at counting, percentages and one area-over-perimeter formula; the Python does the bookkeeping and, in the last example, tries every possible map.

> **WHAT WAS RUN, AND WHAT TO TRUST**
>
> Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Historical and technical claims come from two web searches whose results were encyclopedia mirrors, a law-school summary, news and blog pieces, and an arXiv paper; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. **The sources conflict** on who published the famous 1812 cartoon (one names the Boston Gazette, another the Boston Centinel and the artist Elkanah Tisdale), and they normalise the efficiency gap slightly differently (by the district's votes or by all votes in the election). All vote counts and shapes in the code are *invented*, not real elections or real maps. Statements marked (check) were not confirmed against a source.

Why Lines Matter

Under plurality voting in single-member districts (How Voting Systems Work 2), each district is a separate little election, and only the winner counts. Votes for a loser, and votes for a winner beyond what was needed, change nothing. A mapmaker who knows roughly where each party's voters live can therefore choose lines that make one party's votes count for more than the other's. Two techniques do almost all of the work:

- **Packing:** cram the opponent's voters into a few districts they win by huge margins, wasting their surplus.
- **Cracking:** split the opponent's voters across many districts so that they are a narrow minority in each and win none.

The word itself dates from 1812 in Massachusetts. The sources I found agree it was coined in connection with state senate districts redrawn under Governor Elbridge Gerry, and that one district was said to look like a salamander, giving "Gerry-mander". They also note the coiner may never be firmly established. The practice is older than the word and has not gone away.

Counting Wasted Votes

A **wasted vote** is any vote that did not help elect anyone: every vote for the loser in a district, plus the winner's votes beyond a bare majority. The **efficiency gap**, proposed by Stephanopoulos and McGhee (per the sources I found), compares the two parties' wasted votes: take party A's wasted total minus party B's, and divide by the total votes cast. If A wastes more than B, the map favours B. The same sources report that its authors suggested a gap of about 8% or more as a flag for possible partisan gerrymandering; I could not confirm the exact threshold in a primary source (check).

The experiment below uses fifty invented voters in five districts of ten, with party A holding 26 votes (52%) in every plan. Only the lines change.

```
# Illustrative numbers: 50 voters in 5 equal districts of 10; party A has 26 votes in to
# Each plan lists A's votes in each district; B gets the rest.
def wasted(a, b):
    """Votes wasted in one district: all of the loser's votes, and the winner's votes ab
    needed = (a + b) // 2 + 1
    return (a - needed, b) if a > b else (a, b - needed)

def efficiency_gap(districts):
    """(wasted A - wasted B) / total votes. Positive means A wastes more, so the map fav
    wa = wb = total = 0
    for a, b in districts:
        x, y = wasted(a, b)
        wa, wb, total = wa + x, wb + y, total + a + b
    return (wa - wb) / total

plans = {
    "even spread": [6, 6, 6, 4, 4],
    "B packed":    [6, 6, 6, 6, 2],
    "A packed":    [9, 9, 3, 3, 2],
}
for name, a_votes in plans.items():
```

```
    districts = [(a, 10 - a) for a in a_votes]
    seats = sum(1 for a, b in districts if a > b)
    print(f"{name:<12} A votes {sum(a_votes)}/50, A seats {seats}/5, efficiency gap {eff
# even spread  A votes 26/50, A seats 3/5, efficiency gap -0.08
# B packed     A votes 26/50, A seats 4/5, efficiency gap -0.32
# A packed     A votes 26/50, A seats 2/5, efficiency gap +0.16
```

Look at the second row of the output: A wins 4 of 5 seats with 52% of the votes, because B's 24 voters are packed into one district (8 B to 2 A) and A holds a comfortable margin elsewhere. In the third row the roles flip: A's voters are packed into two districts (9 to 1 each) and cracked across the other three (3, 3 and 2 out of 10), so A wins only 2 of 5 seats with the majority of votes. The efficiency gap moves with the outcome: negative when the map favours A, positive when it favours B.

PLAN (A HAS 26 OF 50 VOTES)	A VOTES PER DISTRICT	A SEATS	EFFICIENCY GAP
Even spread	6, 6, 6, 4, 4	3 of 5	−0.08 (mild tilt to A)
B packed	6, 6, 6, 6, 2	4 of 5	−0.32 (strong tilt to A)
A packed and cracked	9, 9, 3, 3, 2	2 of 5	+0.16 (tilt to B)

Measuring Shape: Compactness

A second family of tests ignores votes and looks at the shape of each district. A fair-looking

introduced in 1991 according to the sources I found, is $4\pi \times \text{area} \div \text{perimeter}^2$. A perfect circle scores 1 and thin or ragged shapes score near 0. Real maps measure area and perimeter from boundary data; the code below uses shapes built from unit squares, where the perimeter is just the number of cell edges that face the outside.

```
from math import pi

def area_perimeter(cells):
    """Area and perimeter of a shape made of unit squares given as (row, col) cells."""
    cells = set(cells)
    perimeter = 0
```

```
    for r, c in cells:
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            if (r + dr, c + dc) not in cells:
                perimeter += 1
    return len(cells), perimeter

def polsby_popper(cells):
    """4 * pi * area / perimeter**2: 1 for a perfect circle, near 0 for a thin or ragged
    area, perimeter = area_perimeter(cells)
    return 4 * pi * area / perimeter ** 2

# Illustrative shapes, all 16 cells in area.
shapes = {
    "4x4 square": [(r, c) for r in range(4) for c in range(4)],
    "2x8 block":  [(r, c) for r in range(2) for c in range(8)],
    "1x16 strip": [(0, c) for c in range(16)],
    "plus + arms": [(r, 3) for r in range(7)] + [(3, c) for c in range(7) if c != 3] + [
}
for name, cells in shapes.items():
    area, perimeter = area_perimeter(cells)
    print(f"{name:<12} area {area:>2}  perimeter {perimeter:>2}  Polsby-Popper {polsby_p
# 4x4 square   area 16  perimeter 16  Polsby-Popper 0.785
# 2x8 block    area 16  perimeter 20  Polsby-Popper 0.503
# 1x16 strip   area 16  perimeter 34  Polsby-Popper 0.174
# plus + arms  area 16  perimeter 40  Polsby-Popper 0.126
```

The 4x4 square scores 0.785 (the best a grid of squares can do for this area, since a square is not a circle). The same 16 cells stretched into a 1x16 strip drop to 0.174, and the tentacled plus-shape to 0.126. Area is identical in all four, so the score is measuring perimeter, which is why compactness is easy to compute and easy to argue about: it says nothing about who the voters are, and a real coastline or river border can make a perfectly innocent district look ragged.

MEASURE	LOOKS AT	STRENGTH	BLIND SPOT
Polsby–Popper	area and perimeter of each district	simple, needs no election data	natural borders; a compact map can still be biased
Efficiency gap	wasted votes in an actual election	measures the outcome, not the look	depends on that election's turnout and swing; sources differ on its normalisation

MEASURE	LOOKS AT	STRENGTH	BLIND SPOT
Seats vs votes	share of seats against share of votes	easy to explain	some mismatch is normal under plurality

How Much Power Does the Mapmaker Have?

The final experiment removes all judgement: it takes a 4x4 grid of voting blocks, with party A holding 9 of the 16 (56%) clustered on the left, and tries *every* way to cut the grid into four connected districts of four blocks each. A district counts for A only with at least 3 of its 4 blocks; a 2-2 split goes to neither party in this simplified model. The function builds each connected piece by growing it one neighbouring cell at a time, then fills in the rest recursively.

```
from itertools import product

SIZE, PER = 4, 4   # a 4x4 grid of voting blocks, drawn into 4 connected districts of 4 b

def neighbours(cell):
    r, c = cell
    return [(r + dr, c + dc) for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1))
            if 0 <= r + dr < SIZE and 0 <= c + dc < SIZE]

def pieces_from(start, free):
    """Every connected set of PER free cells that contains start."""
    found, frontier = set(), {frozenset([start])}
    for _ in range(PER - 1):
        frontier = {piece | {n} for piece in frontier for cell in piece
                    for n in neighbours(cell) if n in free and n not in piece}
    found |= frontier
    return found

def all_maps(free=None):
    """Every way to cut the grid into connected districts of equal size."""
    free = frozenset(product(range(SIZE), repeat=2)) if free is None else free
    if not free:
        yield []
        return
    start = min(free)
    for piece in pieces_from(start, free):
        for rest in all_maps(free - piece):
```

```
            yield [piece] + rest

# Illustrative layout: 'A' blocks (9 of 16, 56%) are packed on the left, 'B' blocks (7) c
layout = ["AAAB",
          "AAAB",
          "AABB",
          "AABB"]
vote = {(r, c): layout[r][c] for r in range(SIZE) for c in range(SIZE)}

tally = {}
maps = list(all_maps())
for plan in maps:
    a_seats = sum(1 for d in plan if sum(vote[c] == "A" for c in d) > PER // 2)
    tally[a_seats] = tally.get(a_seats, 0) + 1
print("valid maps:", len(maps))
for seats in sorted(tally):
    print(f"  A wins {seats} of 4 seats in {tally[seats]} maps")
# valid maps: 117
#   A wins 1 of 4 seats in 7 maps
#   A wins 2 of 4 seats in 84 maps
#   A wins 3 of 4 seats in 26 maps
```

There are 117 valid maps. Proportional shares would give A about 2.25 seats, and 84 of the maps give exactly 2. But 7 maps hand A only 1 seat and 26 give it 3: the mapmaker's choice spans a range, and a mapmaker who wants a particular outcome simply picks from the end of the list that suits them. This is the heart of the problem. Nothing in the layout of voters forces one answer; the lines are a free choice, and free choices can be made with an agenda. Exercise 3 repeats the experiment on random layouts.

What the Law and the Maths Say

In the United States, the sources I found report that the Supreme Court held in *Rucho v. Common Cause* (2019), by a 5-4 majority, that federal courts cannot decide partisan gerrymandering claims because they are political questions, leaving the matter to states; racial gerrymandering claims remain actionable. One result I found was a blog post describing a later decision, which I could not confirm against a second source, so I leave it out. Other countries handle the problem differently: some give boundary drawing to independent commissions (check which, and when), and systems with proportional representation (How Voting Systems

Work 7 covered the seat-sharing step) largely avoid the problem because seats follow votes across whole regions rather than single districts (check).

A paper on arXiv titled "An impossibility theorem for gerrymandering" argues, as the search summary put it, that districting cannot always satisfy all three of one person one vote, Polsby–Popper compactness and a low efficiency gap at once. I did not read the paper's full argument, so take that as a pointer rather than a result I verified (check). It echoes the theme of How Voting Systems Work 6: every fairness measure is a value choice, and no one number settles the question.

Common mistakes

Reading a high efficiency gap as proof of intent (geography can cause it: voters who cluster in cities get packed with no mapmaker needed); trusting compactness alone (an ugly shape can be harmless and a tidy one biased); forgetting that the gap depends on one election's votes, not a map's whole life; and mixing up the sign convention. Here a positive gap means party A wasted more, so the map favours B; some authors use the opposite sign, so always check which way a quoted figure points.

EXERCISE 1

Six invented districts have the vote tallies A:B of 5200:4800, 4100:5900, 3300:6700, 6100:3900, 4600:5400 and 4400:5600. Print each district's wasted votes for both parties, then the efficiency gap, which party it favours, and whether it passes an 8% flag.

View solution

EXERCISE 2

Build three 12-cell shapes (a 3x4 block, an L shape and a staircase) and rank them by Polsby–Popper score. Do any two tie? What does that tell you about the score?

View solution

EXERCISE 3

Place 9 A blocks at random (seeded) on the 4x4 grid for eight layouts, enumerate all 117 maps for each, and report the range of seats A can win. Print the layout where the mapmaker has the widest range.

View solution

QUICK REFERENCE

- **Packing:** concentrate the opponent's voters in a few districts
- **Cracking:** split the opponent's voters so they lose everywhere narrowly
- **Wasted votes:** all of the loser's votes, plus the winner's votes beyond a bare majority
- **Efficiency gap:** (wasted A − wasted B) ÷ total votes; positive favours B in this chapter's convention
- **Polsby–Popper:** $4\pi \times \text{area} \div \text{perimeter}^2$; 1 is a circle, near 0 is thin or ragged
- **Mapmaker's power:** many valid maps, many possible seat counts
- **Always state:** the election used, the sign convention and the normalisation when quoting a gap

COMING NEXT

How Voting Systems Work 9 is about power and weighted votes: voting weight versus power, and the Shapley–Shubik and Banzhaf indices computed in Python.

How Voting Systems Work

Chapter 9 · Power and Weighted Votes

How Voting Systems Work 8 showed that the lines on a map can matter as much as the votes. This chapter looks at a different hidden lever: **weighted voting**, where voters or bodies carry different numbers of votes, as shareholders do in a company, member states do on a council, or parties do in a parliament. The tempting assumption is that someone with 40% of the votes has 40% of the say. It is often wrong. The maths stays at counting and one small factorial formula; the Python lists every coalition and every ordering so you can watch the effect for yourself.

> **WHAT WAS RUN, AND WHAT TO TRUST**
>
> Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Historical and technical claims come from two web searches whose results were encyclopedia mirrors, lecture notes, textbook PDFs and arXiv papers; fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. **The sources conflict** on credit: one says the Banzhaf index was first created in 1946 by Lionel Penrose and reintroduced by John Banzhaf in 1965, while others mention only Banzhaf, so I name both and do not rank them. All weights, quotas, populations and parties in the code are *invented*, not real bodies. Statements marked (check) were not confirmed against a source.

Weight Is Not Power

Picture an illustrative parliament of 100 seats with three parties holding 49, 48 and 3 seats. A bill needs 51 votes. The 3-seat party looks like an afterthought, yet any two of the three parties can pass a bill, so every party is equally able to join a winning pair, and equally able to be left out. The small party is exactly as necessary as the big ones. Power in a voting body is about the *ability to change the outcome*, not the number printed beside your name.

Two ways of measuring this have lasted. Both start from a **weighted voting system** written `[quota: w1, w2, ...]`: each voter has a weight, and a motion passes when the weights of its supporters add up to at least the quota. A group of supporters is a **coalition**; it is **winning** if it reaches the quota.

The Banzhaf Index: Who Is Critical?

In a winning coalition, a voter is **critical** if the coalition would lose without them. Count how many winning coalitions each voter is critical in; a voter's Banzhaf share is their count divided by the total of all the counts. The index carries John Banzhaf's name from his 1965 analysis of a county board in New York, which the sources say was used in a lawsuit that led to a change in the voting system (I did not confirm the details, so I leave them out), and it is tied to Penrose's earlier work as noted above. Some authors use the raw count instead of the share; the share is easier to compare across systems.

```
def banzhaf(weights, quota):
    """Banzhaf counts and shares: how often each voter is critical in a winning coalitio
    n = len(weights)
    critical = [0] * n
    for mask in range(1 << n):
        members = [i for i in range(n) if mask >> i & 1]
        total = sum(weights[i] for i in members)
        if total >= quota:
            for i in members:
                if total - weights[i] < quota:
                    critical[i] += 1
    s = sum(critical)
    return critical, [c / s for c in critical]

# Illustrative parliament: three parties with 49, 48 and 3 seats; 51 of 100 needed to pa
weights, quota = [49, 48, 3], 51
critical, share = banzhaf(weights, quota)
for name, w, c, p in zip("ABC", weights, critical, share):
    print(f"party {name}: {w:>2} seats ({w/sum(weights):5.1%})  critical {c}  Banzhaf sh
# party A: 49 seats (49.0%)  critical 2  Banzhaf share 0.333
# party B: 48 seats (48.0%)  critical 2  Banzhaf share 0.333
# party C:  3 seats ( 3.0%)  critical 2  Banzhaf share 0.333
```

Each party is critical in exactly two coalitions: A is critical in {A, B} and {A, C}, but not in {A, B, C}, where the others already reach 51 without A. So the 3% party has the same Banzhaf share, one third, as the 49% party. That is the whole lesson in three lines of output.

The Shapley–Shubik Index: Who Tips the Balance?

Lloyd Shapley and Martin Shubik introduced a second index in 1954, built on the Shapley value from cooperative game theory (according to the sources I found). Imagine the voters joining a motion one at a time in a random order. The voter whose arrival lifts the running total to the quota is the **pivot**. A voter's share is the fraction of all possible orderings in which they are the pivot. With n voters there are $n!$ orderings, so this brute-force version stops being practical around ten or eleven voters; Exercise 2 shows a formula that avoids listing them.

```
from itertools import permutations

def shapley_shubik(weights, quota):
    """Share of the orderings in which each voter is the pivot (the one who tips the tota
    n = len(weights)
    pivots = [0] * n
    count = 0
    for order in permutations(range(n)):
        count += 1
        running = 0
        for i in order:
            running += weights[i]
            if running >= quota:
                pivots[i] += 1
                break
    return pivots, [p / count for p in pivots]

weights, quota = [49, 48, 3], 51
pivots, share = shapley_shubik(weights, quota)
for name, w, p, s in zip("ABC", weights, pivots, share):
    print(f"party {name}: {w:>2} seats  pivot in {p} of 6 orderings  Shapley-Shubik share
# party A: 49 seats  pivot in 2 of 6 orderings  Shapley-Shubik share 0.333
# party B: 48 seats  pivot in 2 of 6 orderings  Shapley-Shubik share 0.333
# party C:  3 seats  pivot in 2 of 6 orderings  Shapley-Shubik share 0.333
```

For this parliament the two indices agree: each party is the pivot in 2 of the 6 orderings. They do not always agree, as the next section shows.

Comparing the Two Indices

The next run uses an illustrative six-member board with weights 9, 9, 7, 3, 1 and 1 and a quota of 16 out of 30. Look at the weight column against the two power columns. A lecture I found for a six-member county-board example reports shares of one third for three members and zero for the other three, which is the same pattern as here; I used my own invented weights rather than copy that example. Members D, E and F can never be critical: they are **dummies**, however many votes they hold. The second system, with weights 3, 2, 1 and 1, is there to show that the two indices can give different answers.

```
from itertools import permutations

def banzhaf(weights, quota):
    n = len(weights)
    critical = [0] * n
    for mask in range(1 << n):
        members = [i for i in range(n) if mask >> i & 1]
        total = sum(weights[i] for i in members)
        if total >= quota:
            for i in members:
                if total - weights[i] < quota:
                    critical[i] += 1
    return [c / sum(critical) for c in critical]

def shapley_shubik(weights, quota):
    n = len(weights)
    pivots, count = [0] * n, 0
    for order in permutations(range(n)):
        count += 1
        running = 0
        for i in order:
            running += weights[i]
            if running >= quota:
                pivots[i] += 1
                break
    return [p / count for p in pivots]

def report(weights, quota):
    total = sum(weights)
    bz, ss = banzhaf(weights, quota), shapley_shubik(weights, quota)
    print(f"weights {weights}, quota {quota} of {total}")
    print("member weight  weight%  Banzhaf  Shapley-Shubik")
    for name, w, b, s in zip("ABCDEF", weights, bz, ss):
        print(f"{name:>6} {w:>6} {w/total:8.1%} {b:8.3f} {s:15.3f}")
```

```
# Illustrative board: six members with weights 9, 9, 7, 3, 1, 1 (total 30); 16 votes nee
report([9, 9, 7, 3, 1, 1], 16)
print()
# A second illustrative system where the two indices disagree.
report([3, 2, 1, 1], 5)
# weights [9, 9, 7, 3, 1, 1], quota 16 of 30
# member weight  weight%  Banzhaf  Shapley-Shubik
#      A      9    30.0%    0.333           0.333
#      B      9    30.0%    0.333           0.333
#      C      7    23.3%    0.333           0.333
#      D      3    10.0%    0.000           0.000
#      E      1     3.3%    0.000           0.000
#      F      1     3.3%    0.000           0.000
#
# weights [3, 2, 1, 1], quota 5 of 7
# member weight  weight%  Banzhaf  Shapley-Shubik
#      A      3    42.9%    0.500           0.583
#      B      2    28.6%    0.300           0.250
#      C      1    14.3%    0.100           0.083
#      D      1    14.3%    0.100           0.083
```

On the board, A, B and C hold 30%, 30% and 23.3% of the weight yet each has exactly one third of the power, and the 3-vote member has none, because any two of A, B and C reach 16 on their own. In the second system, A has 42.9% of the weight and 0.500 of the Banzhaf power but 0.583 of the Shapley–Shubik power. Neither index is the "true" one; they answer slightly different questions ("in how many coalitions am I decisive?" against "how often do I arrive as the one who tips the total?"). The sources I found say the two usually give similar results; I did not test that beyond these examples.

INDEX	COUNTS	QUESTION IT ANSWERS	COST FOR *N* VOTERS
Weight share	votes held	how much of the total is mine?	trivial
Banzhaf	coalitions where I am critical	how often can I change a result?	2^n coalitions
Shapley–Shubik	orderings where I am the pivot	how often do I tip the balance?	$n!$ orderings, or 2^n with a formula

Quotas, Councils and the Square-Root Idea

The quota is as important as the weights. Lionel Penrose argued in 1946 that for a body whose members represent populations, fairness for individual citizens requires each member's power to scale with the *square root* of its population (the sources I found call this Penrose's law or the square root law). It is a claim about two-tier voting: citizens elect representatives, who then vote as a bloc. The sources also say the Council of the European Union uses a double-majority rule under the Treaty of Lisbon, roughly 55% of member states representing at least 65% of the EU population (I did not check a primary legal source, so confirm current details before relying on this), and that several authors find the current rules deviate from the square-root law. Estimates of the best quota differ between papers: one study reports about 61.4%, while another notes that proposed quotas usually fall between 60% and 74%. That is a genuine conflict in the literature, not a settled number.

The experiment below sweeps the quota for an invented seven-member council, weights equal to populations in millions. For each quota it prints every member's Banzhaf share, the number of dummies, and the ratio between the largest and smallest values of power divided by the square root of population (members with no power are left out of that ratio). A ratio of 1 would match the Penrose idea exactly.

```
from math import sqrt

def banzhaf(weights, quota):
    n = len(weights)
    critical = [0] * n
    for mask in range(1 << n):
        members = [i for i in range(n) if mask >> i & 1]
        total = sum(weights[i] for i in members)
        if total >= quota:
            for i in members:
                if total - weights[i] < quota:
                    critical[i] += 1
    return [c / sum(critical) for c in critical]

# Illustrative council of seven members whose populations (millions) are invented.
pops = [80, 60, 40, 20, 10, 5, 2]
total = sum(pops)
print("quota   " + "  ".join(f"{p:>5}" for p in pops) + "   dummies  spread of power/sqr
for pct in (50, 55, 60, 65, 70, 75):
```

```
    quota = -(-total * pct // 100)          # smallest whole number of votes reaching pc
    bz = banzhaf(pops, quota)
    ratios = [b / sqrt(p) for b, p in zip(bz, pops) if b > 0]   # leave out members with
    spread = max(ratios) / min(ratios)
    print(f"{pct:>3}%  " + "  ".join(f"{b:5.3f}" for b in bz) + f"   {bz.count(0):>5}
# quota      80     60     40     20     10      5      2   dummies  spread of power/sqr
#  50%  0.385  0.231  0.231  0.077  0.077  0.000  0.000       2       2.50
#  55%  0.417  0.250  0.250  0.083  0.000  0.000  0.000       3       2.50
#  60%  0.417  0.250  0.146  0.083  0.062  0.021  0.021       0       5.00
#  65%  0.452  0.286  0.095  0.095  0.024  0.024  0.024       0       6.72
#  70%  0.375  0.300  0.100  0.100  0.075  0.025  0.025       0       3.75
#  75%  0.344  0.344  0.156  0.094  0.031  0.031  0.000       1       4.49
```

Three things stand out. Power is lumpy: moving the quota from 55% to 60% turns three dummies into voters with some power. No quota here gets close to a ratio of 1, partly because with only seven members and whole-number weights the counts are coarse. And a quota that is too low or too high can erase small members entirely (the 75% row has a dummy again). With only one invented council, I would not draw a conclusion about the best real quota; the point is how sensitive power is to a number that looks like a technicality.

ROLE	MEANING	EXAMPLE FROM THIS CHAPTER
Dictator	wins alone	51 of quota 51 (Exercise 1)
Veto holder	no winning coalition without them	everyone under unanimity
Dummy	never critical, whatever the weight	members D, E and F on the board
Ordinary	critical in at least one coalition	all three parties in the 49/48/3 parliament

Weighted voting also connects back to earlier chapters. Apportionment (How Voting Systems

numbers do not translate directly into influence. And like the fairness criteria in How Voting Systems Work 6, the two indices are value choices: each defines “power” one way and no single number ends the argument.

> Common mistakes
> Equating vote share with power; forgetting that the quota changes everything, including who is a dummy; treating a power index as a prediction of how people will vote (it assumes every coalition is

equally likely, which real alliances are not); comparing a Banzhaf *count* in one source with a Banzhaf *share* in another; and listing every ordering for large bodies (16 voters already means about 20 trillion orderings). Also remember these indices measure a priori power from the rules alone, not political influence.

EXERCISE 1

Classify every voter as dictator, veto holder, dummy or ordinary in five systems: `[51: 51, 25, 24]`, `[100: 40, 30, 30]`, `[51: 49, 48, 3]`, `[16: 9, 9, 7, 3, 1, 1]` and `[60: 30, 20, 20, 15, 15]`. Print one line per system.

View solution

EXERCISE 2

Compute the Shapley–Shubik index with a formula (a voter critical in a coalition of size s gets $(s-1)! \times (n-s)!$ orderings out of $n!$) instead of listing orderings. Check it matches the ordering method on eight voters, then run it on sixteen.

View solution

EXERCISE 3

Generate 200 random five-voter systems (seeded, weights 1 to 12, strict-majority quota). Report how many have a dummy, whether anyone ever has more weight but less Banzhaf power, how many have a voter whose power differs from weight share by over 10 points, and the most extreme system.

View solution

QUICK REFERENCE

- **Weighted system:** `[quota: w1, w2, ...]`; a coalition wins if its weights reach the quota
- **Critical voter:** in a winning coalition that would lose without them
- **Banzhaf share:** a voter's critical count ÷ all critical counts
- **Pivot:** the voter who lifts the running total to the quota in a given ordering
- **Shapley–Shubik share:** orderings in which a voter is the pivot ÷ $n!$

- **Dummy, veto holder, dictator:** never critical, always needed, wins alone
- **Penrose's idea:** for fairness between citizens, power should scale with the square root of population
- **Always state:** the weights, the quota and which index (and which normalisation) a figure uses

COMING NEXT

How Voting Systems Work 10 is the capstone, Design an Election: comparing systems on simulated electorates and choosing one for a small club.

How Voting Systems Work

Chapter 10 · Design an Election

This is the capstone. Nine chapters have given you a shelf of tools: plurality (How Voting Systems Work 2), runoffs and instant-runoff (How Voting Systems Work 3), pairwise methods (How Voting Systems Work 4), points and scores (How Voting Systems Work 5), the impossibility results and strategy (How Voting Systems Work 6). Here you use them together on a practical problem: an invented club of 25 members must pick one of five options, and you have to choose the rule. Instead of arguing from taste, you build a small laboratory in Python, run thousands of simulated elections, and let the numbers show the trade-offs. Seats, districts and weighted votes (How Voting Systems Work 7, 8 and 9) do not appear here, because a single-winner club vote needs none of them.

WHAT WAS RUN, AND WHAT TO TRUST

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. The two background facts (the median voter result and the Gibbard–Satterthwaite theorem) come from two web searches whose results were encyclopedia mirrors, lecture notes and arXiv papers; I did not read the original papers, so treat the wording as a paraphrase. **Everything else is a model I invented**: the club, the 2-D "issue space", how voters turn distances into ballots, the polarised electorate, and every percentage in the tables. Those percentages describe this model with these settings, *not* real elections. Statements marked (check) were not confirmed against a source.

The Problem and the Yardsticks

Suppose a club of 25 members must choose among 5 candidate options (a venue, a chairperson, a project; the details do not matter). Before comparing rules you must say what "good" means, because How Voting Systems Work 6 showed that no rule satisfies every fairness idea at once. This chapter uses three yardsticks, each easy to compute:

- **Agreement with the Condorcet winner** (the option that beats every other head to head, How Voting Systems Work 4). A rule that skips it needs a good reason.
- **Regret**: the average distance from the members to the winner, minus the average distance to the best possible option. Zero means the winner is the closest option to the club as a whole.

- **Resistance to burying**: if a bloc of members dishonestly ranks the winner last, can they get an option they truly prefer?

To simulate people you need a model of preferences. The one below places every voter and every candidate at a random point in a unit square (an "issue space" with two dimensions), and a voter prefers candidates that are closer. On a single line, the sources I found credit Duncan Black (1948) with the median voter result: when preferences are single-peaked, the median voter's favourite beats every rival head to head, so it is a Condorcet winner. The same sources say this does not carry over cleanly to more dimensions, and that the McKelvey–Schofield chaos theorem shows there may be no Condorcet winner at all. Two dimensions are therefore a harder, more realistic test, and the simulation will report how often a Condorcet winner exists.

The Toolkit

Save this as `votelab.py`; the later snippets and the exercise solutions import from it. Honest ballots come in three forms from the same voter: a full ranking, an approval set and a 0 to 10 score list. Two choices here are my own assumptions: a voter approves the candidates closer than their average, and scores stretch from 10 for the closest candidate to 0 for the furthest. Ties always go to the lowest-numbered candidate, which is arbitrary but keeps runs repeatable. The Copeland rule is simplified: a head-to-head win counts one, a tie counts nothing.

```
import random
from math import dist as distance

def make_electorate(rng, kind="uniform", n_voters=25, n_cands=5):
    """Return (voters, candidates): points in a 2-D 'issue space' from 0 to 1."""
    cands = [(rng.random(), rng.random()) for _ in range(n_cands)]
    if kind == "uniform":
        voters = [(rng.random(), rng.random()) for _ in range(n_voters)]
    else:  # "polarised": two clusters of voters around (0.25, 0.5) and (0.75, 0.5)
        voters = []
        for _ in range(n_voters):
            cx = rng.choice([0.25, 0.75])
            voters.append((rng.gauss(cx, 0.1), rng.gauss(0.5, 0.15)))
    return voters, cands

def make_ballots(voters, cands):
    """Honest ballots: a ranking, an approval set and a 0-10 score list per voter."""
```

```
    rank, approve, score = [], [], []
    for v in voters:
        d = [distance(v, c) for c in cands]
        rank.append(sorted(range(len(cands)), key=lambda c: (d[c], c)))
        mean = sum(d) / len(d)
        approve.append({c for c in range(len(cands)) if d[c] < mean})   # assumption: ap
        lo, hi = min(d), max(d)
        score.append([round(10 * (hi - x) / (hi - lo)) for x in d])    # assumption: bes
    return {"rank": rank, "approve": approve, "score": score, "n": len(cands)}

def top(totals):
    """Index of the largest total; ties go to the lowest index (an arbitrary, stated rul
    return max(range(len(totals)), key=lambda c: (totals[c], -c))

def plurality(b):
    t = [0] * b["n"]
    for r in b["rank"]:
        t[r[0]] += 1
    return top(t)

def irv(b):
    alive = set(range(b["n"]))
    while True:
        t = {c: 0 for c in alive}
        for r in b["rank"]:
            t[next(c for c in r if c in alive)] += 1
        if 2 * max(t.values()) > len(b["rank"]) or len(alive) == 1:
            return max(sorted(t), key=lambda c: t[c])
        alive.remove(min(sorted(t, reverse=True), key=lambda c: t[c]))   # fewest first

def borda(b):
    t = [0] * b["n"]
    for r in b["rank"]:
        for place, c in enumerate(r):
            t[c] += b["n"] - 1 - place
    return top(t)

def pairwise_wins(b):
    n, wins = b["n"], [0] * b["n"]
    for a in range(n):
        for c in range(a + 1, n):
            a_over_c = sum(1 for r in b["rank"] if r.index(a) < r.index(c))
            c_over_a = len(b["rank"]) - a_over_c
            if a_over_c > c_over_a: wins[a] += 1
            elif c_over_a > a_over_c: wins[c] += 1
```

```
    return wins

def condorcet(b):
    w = pairwise_wins(b)
    return w.index(b["n"] - 1) if (b["n"] - 1) in w else None

def copeland(b):
    return top(pairwise_wins(b))     # simplified Copeland: a win counts 1, a tie counts

def approval(b):
    t = [0] * b["n"]
    for s in b["approve"]:
        for c in s:
            t[c] += 1
    return top(t)

def score(b):
    return top([sum(s[c] for s in b["score"]) for c in range(b["n"])])

METHODS = {"plurality": plurality, "irv": irv, "borda": borda,
           "copeland": copeland, "approval": approval, "score": score}

def regret(voters, cands, winner):
    """Mean distance from voters to the winner, minus the best any candidate achieves."""
    means = [sum(distance(v, c) for v in voters) / len(voters) for c in cands]
    return means[winner] - min(means)
```

One Election, Six Answers

Start with a single seeded election of 25 voters and 4 candidates (A to D). The seed was chosen after searching a few dozen seeds for one in which the rules disagree, so this is a deliberately dramatic example, not a typical one; the next section measures how typical it is.

```
import random
from votelab import make_electorate, make_ballots, condorcet, METHODS, regret

rng = random.Random(11)
voters, cands = make_electorate(rng, "uniform", n_voters=25, n_cands=4)
b = make_ballots(voters, cands)
```

```
names = "ABCD"
firsts = [sum(1 for r in b["rank"] if r[0] == c) for c in range(4)]
print("first places:", dict(zip(names, firsts)))
cw = condorcet(b)
print("Condorcet winner:", names[cw] if cw is not None else "none (a cycle or ties)")
for name, method in METHODS.items():
    w = method(b)
    print(f"{name:>9} -> {names[w]}  regret {regret(voters, cands, w):.3f}")
# first places: {'A': 2, 'B': 9, 'C': 5, 'D': 9}
# Condorcet winner: C
# plurality -> B  regret 0.112
#       irv -> D  regret 0.083
#    borda -> C  regret 0.000
#  copeland -> C  regret 0.000
#  approval -> A  regret 0.004
#     score -> A  regret 0.004
```

Four different winners from the same 25 ballots. Plurality picks B, which ties D on 9 first places (the lowest-index tie rule decides) yet is the worst choice here by regret. Instant-runoff picks D. Borda and Copeland pick C, the Condorcet winner. Approval and score pick A, who has only 2 first places but is acceptable to many voters, so a method that counts breadth rather than first places favours A. Each answer is defensible under some definition of the word “winner”, and that is the course in one output.

Thousands of Elections

One dramatic election proves little, so repeat it. The next snippet runs 2000 seeded elections of 25 voters and 5 candidates for two kinds of electorate, uniform and polarised (voters in two clusters), and tallies how often each rule picks the Condorcet winner and how much regret it leaves on average.

```
import random
from votelab import make_electorate, make_ballots, condorcet, METHODS, regret

def trial(kind, runs=2000, seed=31):
    rng = random.Random(seed)
    has_cw = 0
    agree = {m: 0 for m in METHODS}
```

```
    total_regret = {m: 0.0 for m in METHODS}
    for _ in range(runs):
        voters, cands = make_electorate(rng, kind)
        b = make_ballots(voters, cands)
        cw = condorcet(b)
        if cw is not None:
            has_cw += 1
        for name, method in METHODS.items():
            w = method(b)
            total_regret[name] += regret(voters, cands, w)
            if cw is not None and w == cw:
                agree[name] += 1
    return runs, has_cw, agree, total_regret

for kind in ("uniform", "polarised"):
    runs, has_cw, agree, total_regret = trial(kind)
    print(f"{kind}: {runs} elections, Condorcet winner exists in {has_cw / runs:.1%}")
    print("   method    picks Condorcet winner   mean regret")
    for name in METHODS:
        print(f"{name:>9} {agree[name] / has_cw:>20.1%} {total_regret[name] / runs:>13.4
    print()
# uniform: 2000 elections, Condorcet winner exists in 96.7%
#    method    picks Condorcet winner   mean regret
# plurality                54.4%        0.0289
#       irv                77.2%        0.0111
#     borda                84.9%        0.0041
#  copeland               100.0%        0.0028
#  approval                75.4%        0.0059
#     score                84.2%        0.0019
#
# polarised: 2000 elections, Condorcet winner exists in 96.9%
#    method    picks Condorcet winner   mean regret
# plurality                68.0%        0.0242
#       irv                84.4%        0.0116
#     borda                81.4%        0.0053
#  copeland               100.0%        0.0059
#  approval                72.3%        0.0089
#     score                81.8%        0.0024
```

Read the table with care. Copeland scores 100% on agreement because it is built to choose the Condorcet winner when one exists, so that column cannot tell Copeland apart from its own definition. A Condorcet winner existed in about 97% of these elections, so cycles are rare in this

model (real clubs, with strong factions, may differ; I did not test that). Plurality is clearly the weakest on both measures. Instant-runoff beats plurality on both, and is close to approval on agreement but behind Borda, Score and Copeland. On mean regret the score method is lowest in both electorates, but it benefits from my assumption that voters report stretched, honest scores, which hands it more information than a ranking carries. Polarisation changed the order a little: for example, plurality's agreement rose from 54.4% to 68.0%, and Borda's fell from 84.9% to 81.4%.

METHOD	BALLOT ASKED FOR	AGREES WITH CONDORCET WINNER (UNIFORM)	MEAN REGRET (UNIFORM)
Plurality	one name	54.4%	0.0289
Instant-runoff	full ranking	77.2%	0.0111
Borda	full ranking	84.9%	0.0041
Copeland	full ranking	100.0% (by definition)	0.0028
Approval	a set of names	75.4%	0.0059
Score	a number per name	84.2%	0.0019

Can the Ballots Be Gamed?

The Gibbard–Satterthwaite theorem, proved independently by Allan Gibbard (1973) and Mark Satterthwaite (1975) according to the sources I found, says that with three or more candidates any deterministic rule in which every candidate can win and which is never manipulable must be a dictatorship. In plain terms, every sensible rule can be gamed somewhere; the useful question is how *easily*. This test is one narrow kind of gaming. A bloc is formed from the voters whose favourite is the strongest non-winning candidate, and every one of them buries the current winner: last in the ranking, unapproved, scored 0. The bloc succeeds if the new winner is closer to them, on average, than the old one by their true positions.

```
import random
from math import dist as distance
from votelab import make_electorate, make_ballots, METHODS
```

```python
def bury(b, voter, c):
    """Return a copy of the ballots in which one voter puts candidate c last, unapproved
    new = {"n": b["n"], "rank": list(b["rank"]), "approve": list(b["approve"]), "score":
    new["rank"][voter] = [x for x in b["rank"][voter] if x != c] + [c]
    new["approve"][voter] = b["approve"][voter] - {c}
    new["score"][voter] = [0 if x == c else s for x, s in enumerate(b["score"][voter])]
    return new

def success_rate(method, kind, runs=1000, seed=7):
    rng = random.Random(seed)
    tried = worked = 0
    for _ in range(runs):
        voters, cands = make_electorate(rng, kind)
        b = make_ballots(voters, cands)
        winner = method(b)
        # the bloc: voters whose favourite is the non-winner with most first places
        firsts = [sum(1 for r in b["rank"] if r[0] == c) for c in range(b["n"])]
        rival = max((c for c in range(b["n"]) if c != winner), key=lambda c: (firsts[c],
        bloc = [i for i, r in enumerate(b["rank"]) if r[0] == rival]
        if not bloc:
            continue
        tried += 1
        fake = b
        for i in bloc:
            fake = bury(fake, i, winner)
        new = method(fake)
        mean = lambda c: sum(distance(voters[i], cands[c]) for i in bloc) / len(bloc)
        if new != winner and mean(new) < mean(winner):
            worked += 1                      # the bloc really is better off by its true
    return worked, tried

for name, method in METHODS.items():
    worked, tried = success_rate(method, "uniform")
    print(f"{name:>9}: burying helped the bloc in {worked:>3} of {tried} elections ({wor
# plurality: burying helped the bloc in   0 of 1000 elections (0.0%)
#       irv: burying helped the bloc in  31 of 1000 elections (3.1%)
#    borda: burying helped the bloc in 437 of 1000 elections (43.7%)
#  copeland: burying helped the bloc in 280 of 1000 elections (28.0%)
#  approval: burying helped the bloc in 461 of 1000 elections (46.1%)
#     score: burying helped the bloc in 489 of 1000 elections (48.9%)
```

Plurality scores 0.0% because burying someone who is not your first choice cannot change a first-place count; it is not that plurality resists gaming, only that *this* manoeuvre is useless against it. Plurality's weakness is of another kind, the compromise voting of How Voting Systems Work 2, which this test does not try. Within what it does measure, instant-runoff is the hardest to bury (3.1%), while Borda, approval and score are open to it in 43.7%, 46.1% and 48.9% of elections. In Copeland the figure is 28.0%. A single narrow strategy tells you little about the others, so read these as signs of where to look, not a final verdict on manipulation.

Choosing for the Club

The simulation cannot name a winner for you, because the yardsticks pull in different directions. No rule leads on all three. Weights make the decision, and the weights are a statement about what your club values.

IF THE CLUB CARES MOST ABOUT…	LEAN TOWARDS	WHY, FROM THIS MODEL	WATCH OUT FOR
Never choosing a loser to a rival head to head	Copeland or another Condorcet method	picks the Condorcet winner whenever one exists; low regret	needs full rankings; a cycle needs a pre-agreed tie rule
Lowest average dissatisfaction	Score voting	lowest mean regret in both electorates	relies on honest, comparable scores; easy to exaggerate
Hard to game by burying	Instant-runoff or plurality	burying helped the bloc in 3.1% and 0.0% of elections	weaker on the other two yardsticks; plurality has compromise voting
Simplest to explain and count	Plurality or approval	one mark per name; no ranking arithmetic	plurality had the most regret here

A practical suggestion of mine, not a result from the simulation: for a 25-member club that is comfortable ranking its options, collect ranked ballots, compute the Condorcet winner, and write down in advance what happens if there is none (the Copeland rule, say). Rankings are cheap to collect, they let you check several rules on the same ballots, and the rule is agreed *before* anyone knows the candidates, which removes the temptation to pick the rule that favours a particular option. Exercise 3 lets you change the weights and watch the recommendation move.

> Common mistakes

Treating a simulation as evidence about real elections when it only shows what the model implies; reading a method's 100% score when the yardstick is its own definition; picking a rule after seeing the candidates, so that the rule is chosen to produce a result; comparing methods on a single dramatic election instead of thousands; giving score and approval ballots more information than the ranking ballots and then crediting the method; and forgetting that one strategy test (burying) covers only one kind of manipulation. Also remember the lowest-index tie rule is arbitrary: with only 25 voters, ties are common enough that a real club must say how it breaks them.

EXERCISE 1

Add a top-two runoff (the two largest first-place totals meet in a second round) to the toolkit. On 2000 uniform elections, compare plurality, the runoff and instant-runoff on agreement with the Condorcet winner and mean regret.

View solution

EXERCISE 2

A club may care most about never electing someone a large part of the membership dislikes intensely. For 2000 polarised elections, count how often each method's winner is ranked last by at least a quarter of the voters.

View solution

EXERCISE 3

Build a weighted scorecard from the three yardsticks (Condorcet agreement, low regret, burying resistance) and rank the six methods under three sets of invented weights: fairness first, closeness first, honesty first. Does the best method change with the weights?

View solution

QUICK REFERENCE

- **Choose the yardsticks first:** Condorcet agreement, regret, resistance to gaming, simplicity; none wins on all
- **Spatial model:** voters and candidates as points; closer means preferred (an assumption)

- **Regret:** mean distance to the winner minus mean distance to the best candidate
- **Burying test:** a bloc ranks the winner last; success means a truly better result for the bloc
- **Gibbard–Satterthwaite:** with three or more candidates, every non-dictatorial rule in which anyone can win is open to manipulation somewhere
- **Always state:** the ballot model, the tie rule, the number of elections and the seed behind any percentage
- **Settle the rule, including tie-breaks, before the candidates are known**

WHERE TO GO FROM HERE

You now have a working toolkit and the habit of asking what a number measures. To go further, change the model: add voters who do not turn out, candidates who enter late, or more issue dimensions, and see which conclusions survive. Re-read How Voting Systems Work 6 with the toolkit open and test another fairness criterion by code. Try the seat and district chapters on real published results (How Voting Systems Work 7 and 8), and use the power indices from How Voting Systems Work 9 on a real council. Whatever you choose, keep the three rules of this course: run it, label your assumptions, and distrust any single number.