Chapter 5: Auctions

Algorithmic Game Theory

Four basic formats | Vickrey truthfulness proof | Bid shading | Revenue equivalence | Reserve prices | Winner's curse | Double auctions

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/agt/slides/Chapter_05_Auctions.html

Warm-up Game: The Vickrey Auction

The rules

Everyone writes a bid in a sealed envelope; the highest bidder gets the item but pays the second-highest bid, not their own.

Think about it

- If the item is worth 70 TL to you, would you write less than 70, exactly 70, or more?
- Would your bid change if you knew the rivals' bids?
- If the winner paid their own bid (first price), would writing 70 still be a good idea?



You get the item and pay 60 TL.

If your value is 70 TL, you gain 70 − 60 = 10.

Learning Outcomes

Running the formats

Find the winner and payment in English, Dutch, first-price and second-price auctions.

Proving truthfulness

Show by case analysis that truthful bidding is weakly dominant in the Vickrey auction.

Equilibrium bids

Derive the first-price shading $b = (n-1)v/n$; compute revenue equivalence (2 bidders $\rightarrow$ 1/3) and the effect of a reserve price (5/12).

Market phenomena

Explain the winner's curse; find the matching and price in a double auction.

Why Auctions Matter

- Efficient mechanisms for allocating resources: the item tends to go to whoever values it most.
- Price discovery and market transparency: the price is found without knowing the buyers' values.
- The most widespread application of mechanism design: an allocation rule + a payment rule.

Art
Ascending-price English auctions at auction houses.

Online advertising
Ad auctions run on every search.

Telecom licenses
Spectrum auctions; multi-item designs worth billions.

Procurement, treasury
Sealed bids; government bond auctions.

Key Terms and the Model

Value v_i

The bidder's true value for the item; private information.

Bid b_i

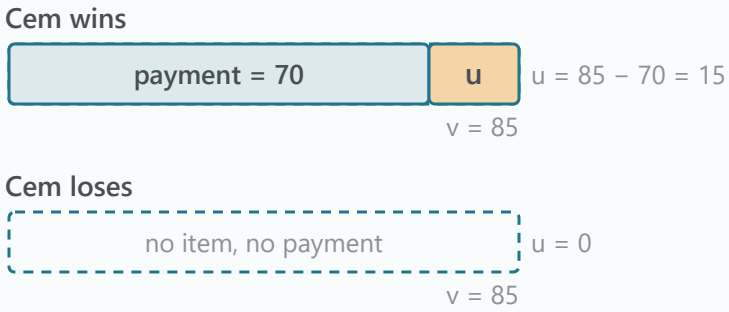
The number reported to the mechanism; it may differ from the value.

Utility

$u_i = v_i - p_i$ if they win (p_i: payment), 0 if they lose.

Independent private values (IPV)

Values are drawn independently and each bidder knows their own. The baseline model of auction theory.



Second-price example (Cem 85, second bid 70): quasi-linear utility.

Examples usually assume $v_i \sim U[0, 1]$ (uniform) and risk-neutral bidders.

BASIC CONCEPTS

Valuation Models

Private values

Each bidder values the item based on their own information (e.g. a painting for their own wall).

Common value

The item's true value is the same for everyone, but the bidders' estimates differ (e.g. an oil field).

Interdependent values

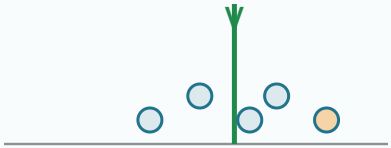
Both private information and the other bidders' information affect valuations.

private value



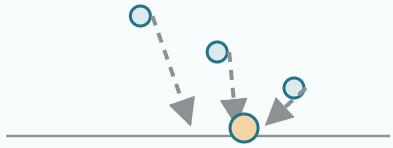
each has their own: $v_1 \neq v_2$ ...

common value



one V, noisy estimates

interdependent



v_i = f(own + others' signals)

The common-value model leads to the **winner's curse**: the winner tends to be the most optimistic estimator, which produces expected losses.

Four Basic Auction Formats

Format	How it works	Winner	Pays
English	The price rises openly; bidders drop out one by one.	Last one standing	Price at which the last rival drops out
Dutch	The price starts high and falls; the first to say “mine” takes it.	First to accept	The current price
First-price sealed-bid	Bids are submitted secretly, once.	Highest bid	Their own bid
Second-price (Vickrey)	Bids are submitted secretly, once.	Highest bid	The second-highest bid

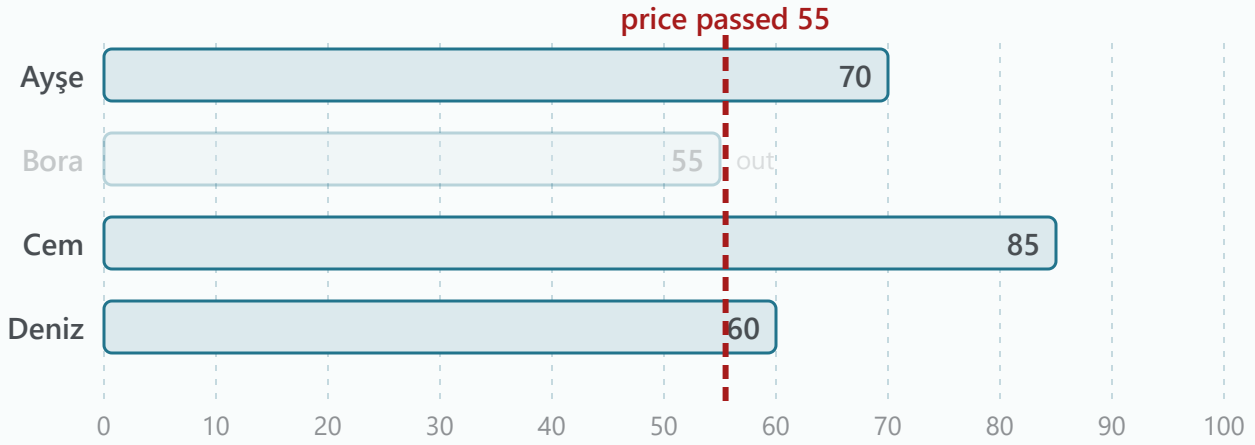
Let's run all four rules with the same four people. The most each is willing to pay (their limit): Ayşe 70, Bora 55, Cem 85, Deniz 60. For now everyone bids their limit.

Step by Step: The English Auction (Ascending Price)



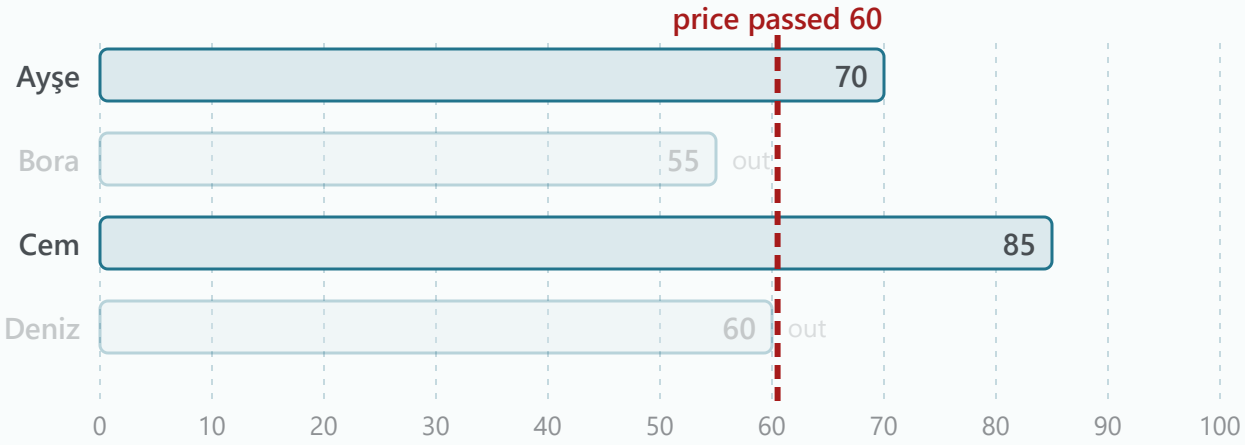
The price starts at 40 and rises. At 50, everyone's limit is above 50: **all four are still in**.

Step by Step: The English Auction (Ascending Price)



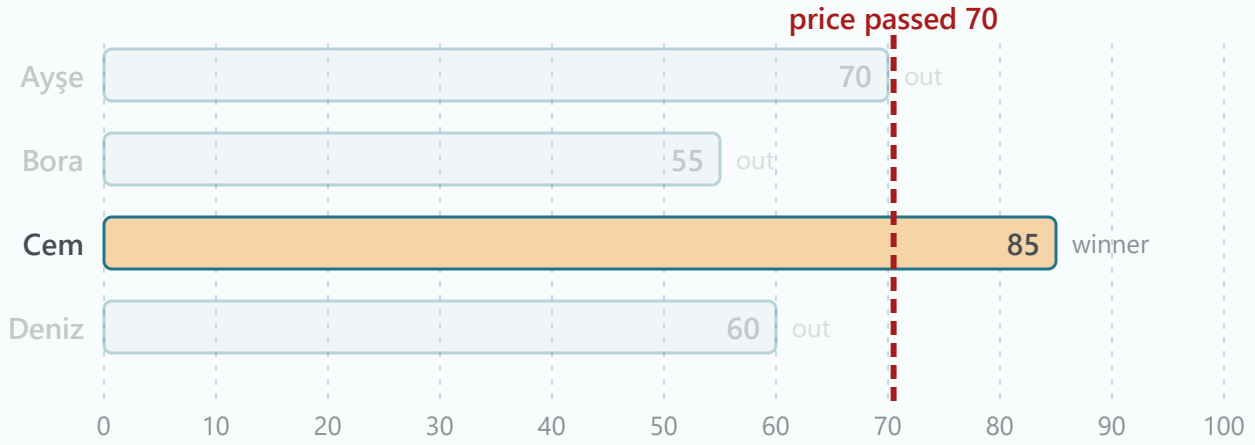
Once the price passes **55**, Bora drops out (limit 55). Left: Ayşe, Cem, Deniz.

Step by Step: The English Auction (Ascending Price)



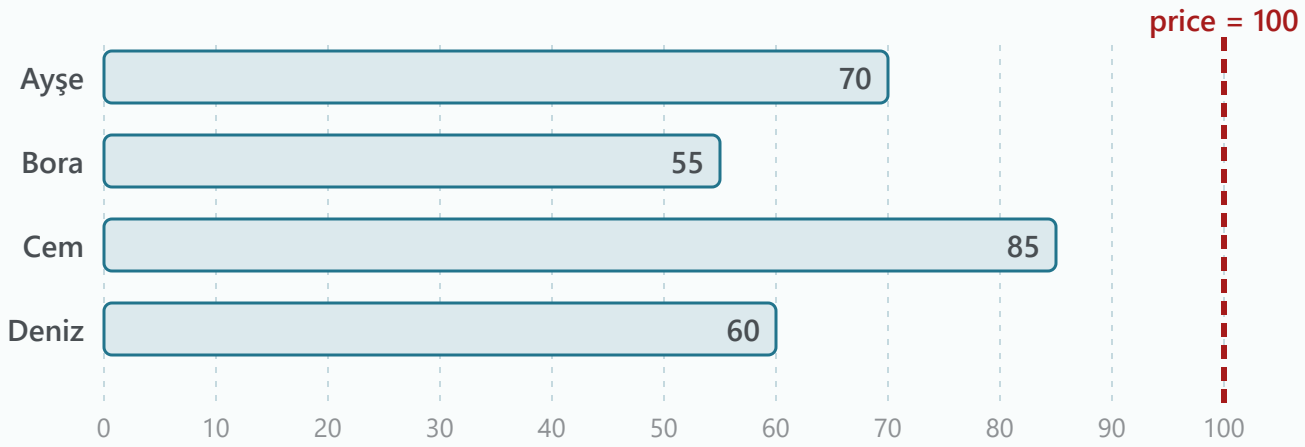
Once the price passes **60**, Deniz drops out (limit 60). Left: Ayşe, Cem.

Step by Step: The English Auction (Ascending Price)



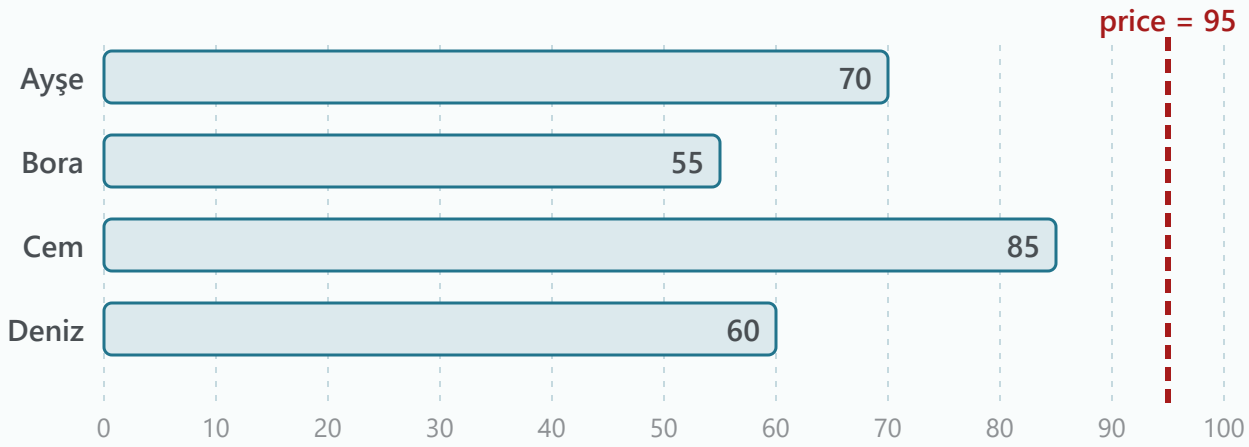
Once the price passes 70, the last rival, Ayşe, drops out → **Cem wins and pays ≈ 70** (exactly 70 as the increment shrinks): the second-highest value.

Step by Step: The Dutch Auction (Descending Price)



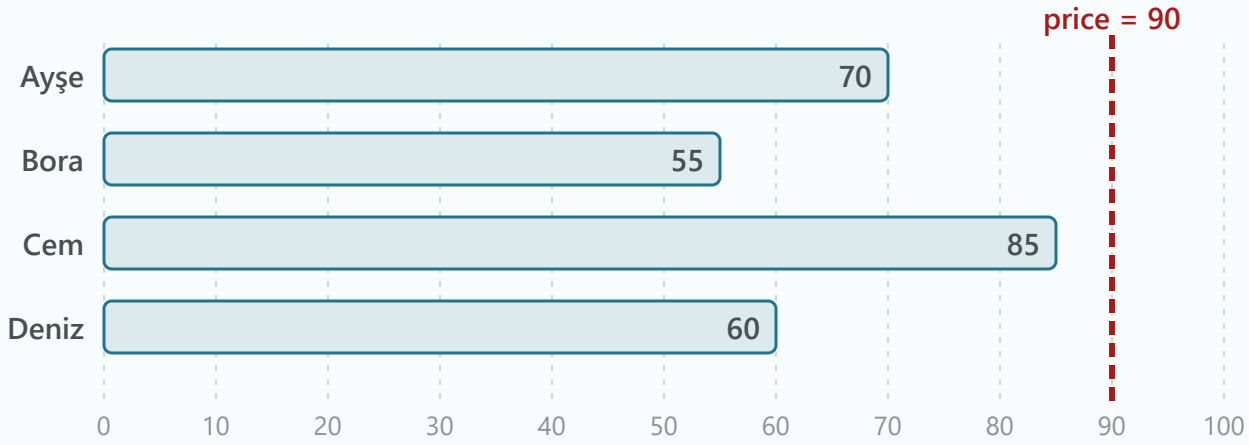
At 100 the price is above everyone's limit; **nobody accepts**, and it drops by 5.

Step by Step: The Dutch Auction (Descending Price)



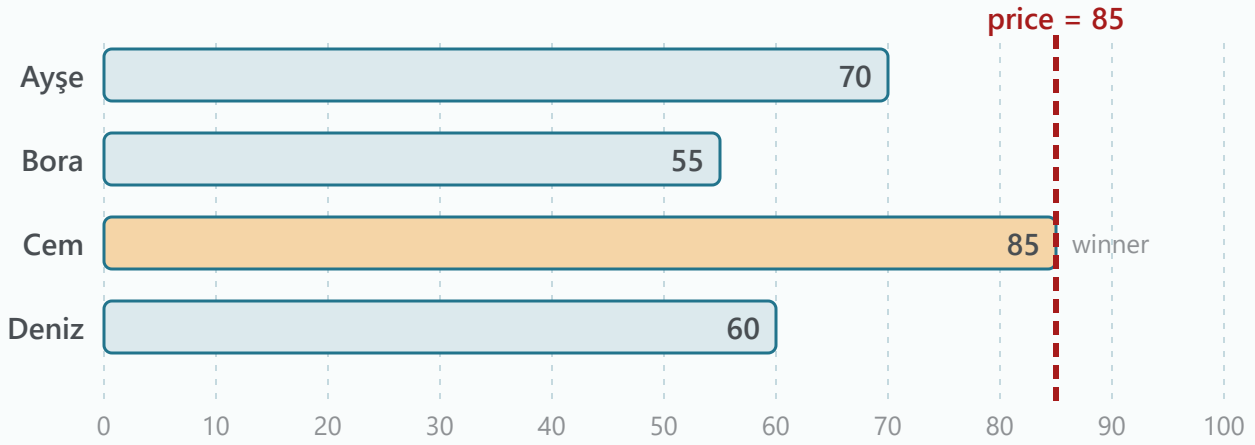
At 95: still nobody accepts (the highest limit is 85).

Step by Step: The Dutch Auction (Descending Price)



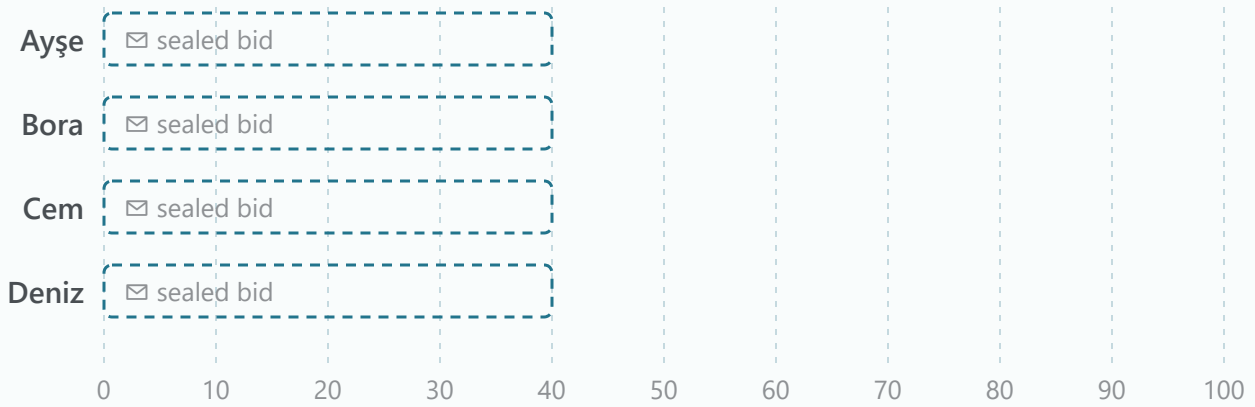
At 90: still nobody accepts (the highest limit is 85).

Step by Step: The Dutch Auction (Descending Price)



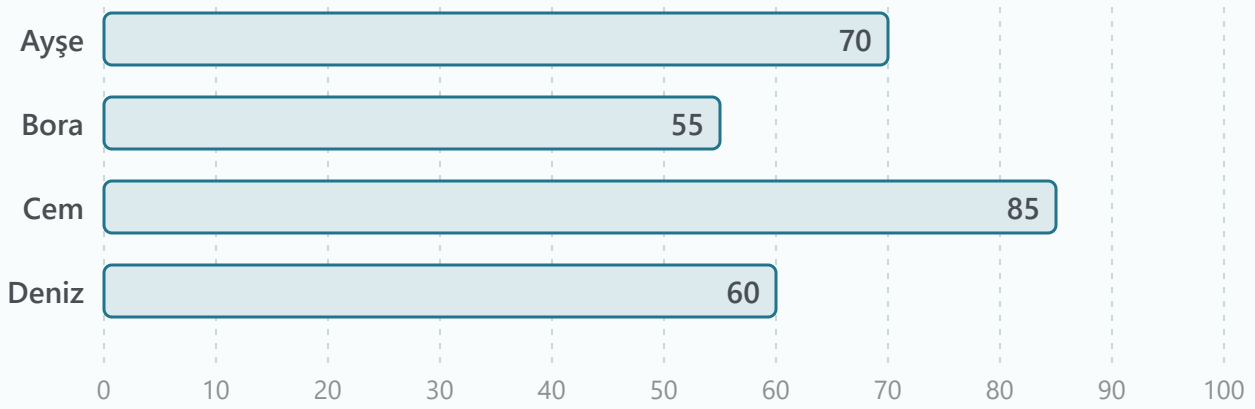
At 85 the price reaches Cem's limit → **Cem says “mine” and pays 85**. The auction ends before anyone else can speak.

Step by Step: First- and Second-Price Sealed Bids



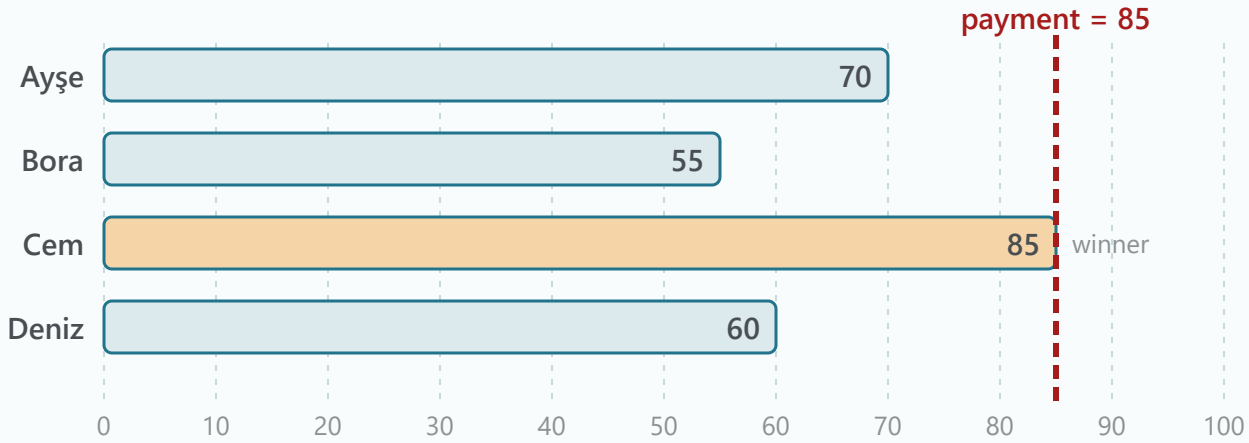
Sealed bids: everyone bids once, in secret. Nobody sees the others' bids.

Step by Step: First- and Second-Price Sealed Bids



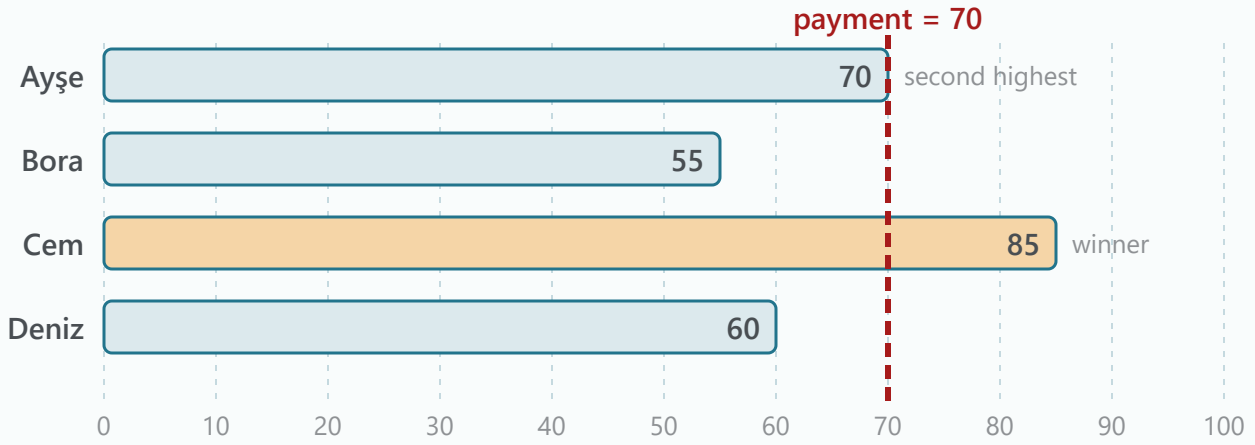
The envelopes are opened: 70, 55, 85, 60. Under both rules the highest bid wins: **Cem**.

Step by Step: First- and Second-Price Sealed Bids



First price: Cem pays his own bid → **85**. If his value is 85, his utility is 0.

Step by Step: First- and Second-Price Sealed Bids



Second price (Vickrey): Cem pays the second-highest bid → **70**. His utility is 85 − 70 = 15.

FOUR BASIC FORMATS

Same Bids, Four Outcomes

Format	Winner	Payment	Whose bid sets it?
English	Cem	≈ 70	The last rival to drop out (Ayşe)
Second price	Cem	70	The second-highest bid (Ayşe)
Dutch	Cem	85	The winner's own
First price	Cem	85	The winner's own

English ≈ second price

With private values, staying in up to your value is dominant; both charge the second-highest value.

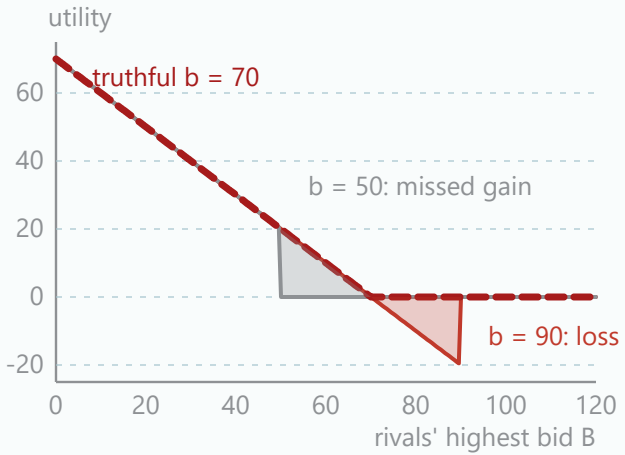
Dutch ≡ first price

Strategically equivalent: “at what price will I accept?” is the same as submitting a sealed bid.

Note: in this table everyone bid their limit. In a first-price auction a rational bidder **shades the bid**, so expected revenues actually come out equal (revenue equivalence).

The Strategic Property of the Vickrey Auction

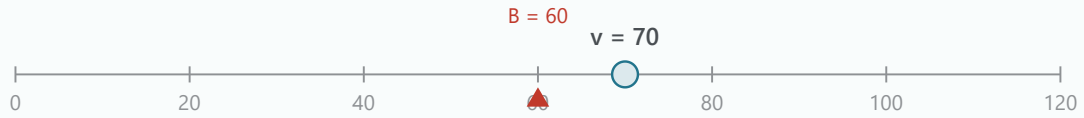
- The highest bidder wins and pays the second-highest bid.
- With independent private values and a single item, bidding your true value is a **weakly dominant strategy** (Vickrey, 1961).
- The idea: your bid determines only *whether you win*, not *what you pay*.



v = 70. Dashed red: the truthful bid. No bid beats it for any B.

The result may fail with common values, budget constraints or interdependent values. Vickrey is the single-item special case of VCG.

Step by Step: Proving Truthfulness in Vickrey



Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—

Your value is v = 70. B = max b_{-i}, the others' highest bid (unknown to you). Bidding truthfully with B = 60, you win and pay 60 → 10.

Step 1 / 8

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey

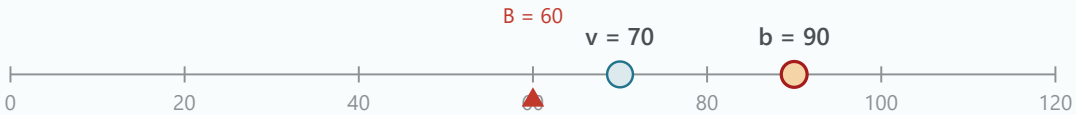


Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—

Bidding truthfully with B = 80, you lose → 0. The truthful bid always makes the right call: win when there is a profit, lose when there would be a loss.

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey



Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same

Overbid b = 90, B = 60 (< v): you still win and still pay 60 → 10. No difference.

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey



Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same
Over b = 90	80	wins, pays 80	**−10**	**worse**

b = 90, B = 80 (between v and b): you win but **pay 80** → **−10**. Bidding truthfully you would get 0. A loss!

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey



Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same
Over b = 90	80	wins, pays 80	**−10**	**worse**
Over b = 90	95	loses	**0**	same

b = 90, B = 95 (> b): you lose → 0, same as truthful.

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey

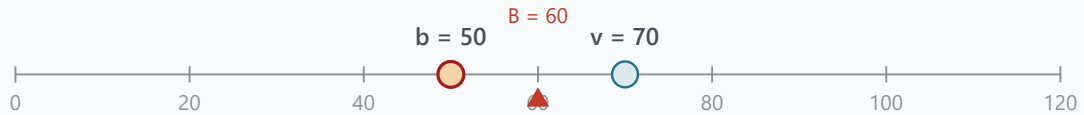


Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same
Over b = 90	80	wins, pays 80	**−10**	**worse**
Over b = 90	95	loses	**0**	same
Under b = 50	40	wins, pays 40	**30**	same

Underbid b = 50, B = 40 (< b): you win and pay 40 → 30. Same as truthful (B, not b, sets your payment).

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey

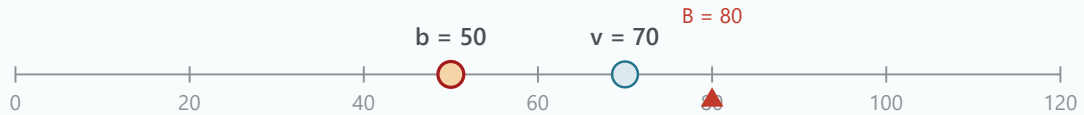


Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same
Over b = 90	80	wins, pays 80	**−10**	**worse**
Over b = 90	95	loses	**0**	same
Under b = 50	40	wins, pays 40	**30**	same
Under b = 50	60	loses	**0**	**worse**

b = 50, B = 60 (between b and v): **you lose** → **0**; bidding truthfully you would have gained 70 − 60 = 10. A missed chance!

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Step by Step: Proving Truthfulness in Vickrey



Strategy	B (rival)	Outcome	Utility	vs. truthful
Truthful b = 70	60	wins, pays 60	**10**	—
Truthful b = 70	80	loses	**0**	—
Over b = 90	60	wins, pays 60	**10**	same
Over b = 90	80	wins, pays 80	**−10**	**worse**
Over b = 90	95	loses	**0**	same
Under b = 50	40	wins, pays 40	**30**	same
Under b = 50	60	loses	**0**	**worse**
Under b = 50	80	loses	**0**	same

b = 50, B = 80: you lose, same as truthful. Conclusion: deviating is never better and sometimes worse → **b = v is weakly dominant**.

Trying all pairs b, B ∈ {0, …, 120} for v = 70, no b ever beats the truthful bid.

Back to the Game: The Vickrey Auction

Theory's answer: write your value

Opened envelopes: you 70, rivals 60 and 45. Bidding truthfully you win and pay 60: a gain of 10. Writing 65 or 100 you would pay the same 60; writing 55, the item goes for 60 and you miss the gain of 10.

Your bid only decides whether you win; the rivals' highest bid sets the price: **truthful bidding is weakly dominant**. Under first price, the same 70 leaves 70 − 70 = 0 even if you win; there you must bid below your value.

In practice

In lab experiments, overbidding in second-price sealed-bid auctions is common and does not easily disappear with repetition; the same subjects converge to equilibrium much faster in ascending English auctions (Kagel, Harstad and Levin, 1987).

In everyday life: in eBay's automatic (proxy) bidding, the winner pays one increment above the second-highest bid — very close to the second-price rule.

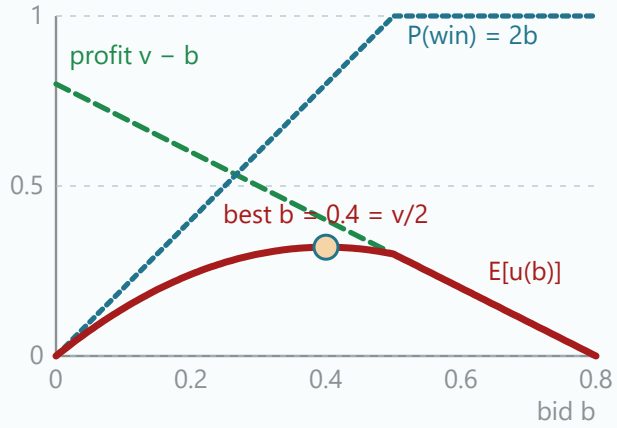
Moving the bid above or below the value never raises the gain for any B; it only lowers it for some B.

Bid Shading in First-Price Auctions

- In a first-price auction, a truthful bid is pointless: even if you win, utility is v − v = 0.
- Lowering the bid raises the profit when you win but lowers the chance of winning:

$$E[u(b)] = (v - b) \cdot P(\text{win with } b)$$

- The equilibrium bid balances this trade-off and depends on how others bid → a **Bayes–Nash equilibrium**.
- The Dutch auction is strategically the same: "at what price will I accept?" = a shaded bid.



Two bidders, v = 0.8; the rival bids half their value.

Step by Step: The Equilibrium Bid b = (n − 1)v / n

1. Let the other n − 1 bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).

Assumption: rivals bid a fixed fraction of their values.

Step 1 / 6

Maximizing E[u(b)] on a grid (10,001 points) for n = 2, 3, 5, 10 and v = 0.3, 0.6, 0.8, the best bid matches (n−1)v/n.

Step by Step: The Equilibrium Bid b = (n − 1)v / n

1. Let the other n − 1 bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).

2. $P(\text{win}) = P(\beta\, v_j < b \text{ for all } j) = (b/\beta)^{n-1}$

For each rival $P(v_j < b/\beta) = b/\beta$; by independence these multiply.

Maximizing E[u(b)] on a grid (10,001 points) for n = 2, 3, 5, 10 and v = 0.3, 0.6, 0.8, the best bid matches (n−1)v/n.

Step by Step: The Equilibrium Bid $b = (n - 1)v / n$

1. Let the other $n - 1$ bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).
2. $P(\text{win}) = P(\beta\, v_j < b \text{ for all } j) = (b/\beta)^{n-1}$
3. $E[u(b)] = (v - b) \cdot (b/\beta)^{n-1}$

Expected utility: profit when winning × probability of winning.

Maximizing $E[u(b)]$ on a grid (10,001 points) for $n = 2, 3, 5, 10$ and $v = 0.3, 0.6, 0.8$, the best bid matches $(n-1)v/n$.

Step by Step: The Equilibrium Bid $b = (n - 1)v / n$

1. Let the other $n - 1$ bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).
2. $P(\text{win}) = P(\beta\, v_j < b \text{ for all } j) = (b/\beta)^{n-1}$
3. $E[u(b)] = (v - b) \cdot (b/\beta)^{n-1}$
4. $dE[u(b)]/db = [\, -b + (n - 1)(v - b)\,] \cdot b^{n-2}/\beta^{n-1} = 0$

The derivative of $(v - b)$ is -1 and that of b^{n-1} is $(n - 1)b^{n-2}$; the product rule adds them and factors out b^{n-2}. At the maximum the derivative is zero.

Maximizing E[u(b)] on a grid (10,001 points) for n = 2, 3, 5, 10 and v = 0.3, 0.6, 0.8, the best bid matches (n−1)v/n.

Step by Step: The Equilibrium Bid b = (n − 1)v / n

1. Let the other n − 1 bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).
2. $P(\text{win}) = P(\beta\, v_j < b \text{ for all } j) = (b/\beta)^{n-1}$
3. $E[u(b)] = (v - b) \cdot (b/\beta)^{n-1}$
4. $dE[u(b)]/db = [\, -b + (n - 1)(v - b)\,] \cdot b^{n-2}/\beta^{n-1} = 0$
5. $-b + (n - 1)(v - b) = 0 \Rightarrow$ **$b = (n - 1)\, v / n$**

After simplifying, the solution does not depend on β.

Maximizing E[u(b)] on a grid (10,001 points) for n = 2, 3, 5, 10 and v = 0.3, 0.6, 0.8, the best bid matches (n−1)v/n.

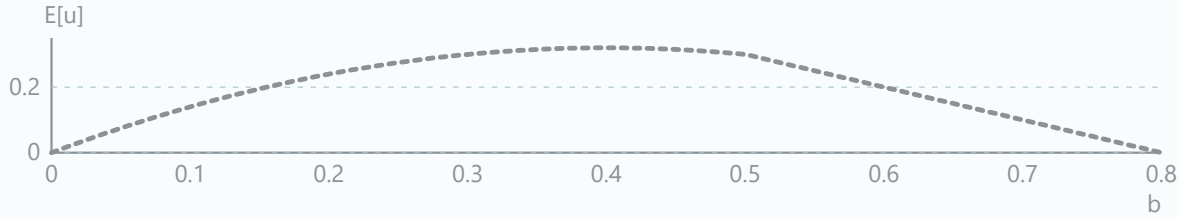
Step by Step: The Equilibrium Bid b = (n − 1)v / n

1. Let the other n − 1 bidders bid $b_j = \beta \cdot v_j$ ($v_j \sim U[0, 1]$).
2. $P(\text{win}) = P(\beta\, v_j < b \text{ for all } j) = (b/\beta)^{n-1}$
3. $E[u(b)] = (v - b) \cdot (b/\beta)^{n-1}$
4. $dE[u(b)]/db = [\, -b + (n - 1)(v - b)\,] \cdot b^{n-2}/\beta^{n-1} = 0$
5. $-b + (n - 1)(v - b) = 0 \Rightarrow$ **$b = (n - 1)\, v / n$**
6. Consistency: $\beta = (n - 1)/n$ ✓ $\quad n = 2 \Rightarrow b = v/2$

My best response has the same form → a symmetric Bayes–Nash equilibrium. With two bidders: half the value.

Maximizing E[u(b)] on a grid (10,001 points) for n = 2, 3, 5, 10 and v = 0.3, 0.6, 0.8, the best bid matches (n−1)v/n.

Code Trace: `best_bid(0.8, 2)`



Rivals bid b = v/2 (β = 0.5). Grid: b = 0, 0.1, ..., 0.8.

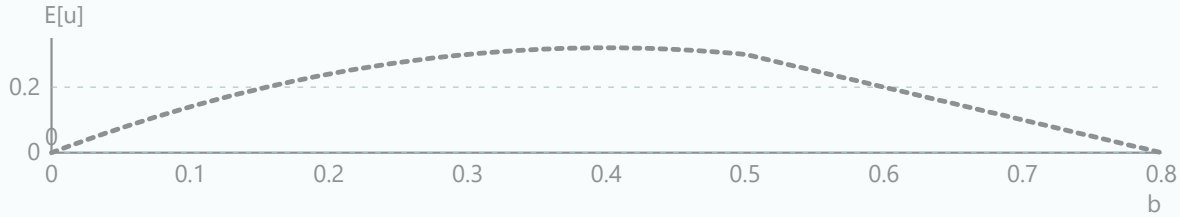
Step 1 / 11

```
1  def best_bid(v, n, step=0.1):
▶    beta = (n - 1) / n        # rivals bid beta*v
▶    best, b_best = -1, 0
▶    b = 0.0
5    while b <= v:
6      p = min(1, b / beta) ** (n - 1)  # P(win)
7      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `best = -1` `b_best = 0` `b = 0`

Code Trace: `best_bid(0.8, 2)`



b = 0: P(win) = 0, E[u] = (0.8 − 0)·0 = **0** → new best.

Step 2 / 11

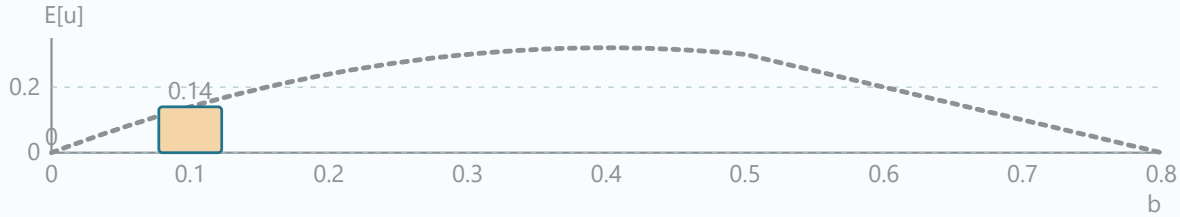
```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
▶        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0`

`p = 0` `eu = 0` `best = 0` `b_best = 0`

Code Trace: `best_bid(0.8, 2)`



b = 0.1: P(win) = 0.2, E[u] = (0.8 − 0.1)·0.2 = **0.14** → new best.

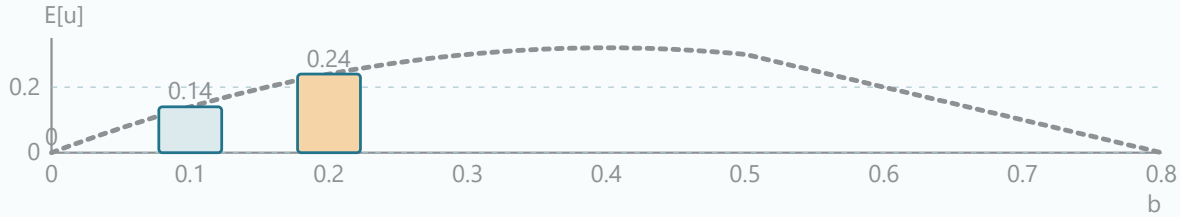
Step 3 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
▶        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.1` `p = 0.2` `eu = 0.14` `best = 0.14` `b_best = 0.1`

Code Trace: `best_bid(0.8, 2)`



b = 0.2: P(win) = 0.4, E[u] = (0.8 − 0.2)·0.4 = **0.24** → new best.

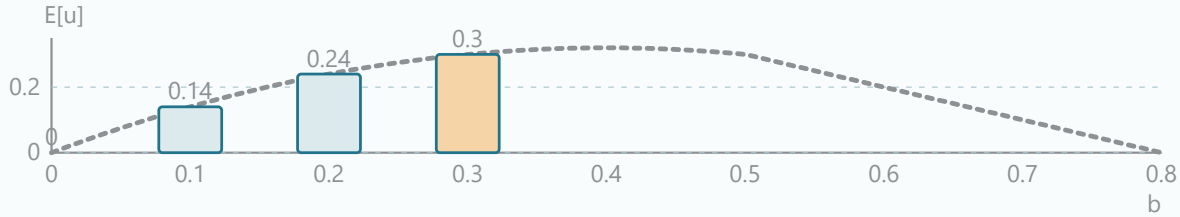
Step 4 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
▶        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.2` `p = 0.4` `eu = 0.24` `best = 0.24` `b_best = 0.2`

Code Trace: `best_bid(0.8, 2)`



b = 0.3: P(win) = 0.6, E[u] = (0.8 − 0.3)·0.6 = **0.3** → new best.

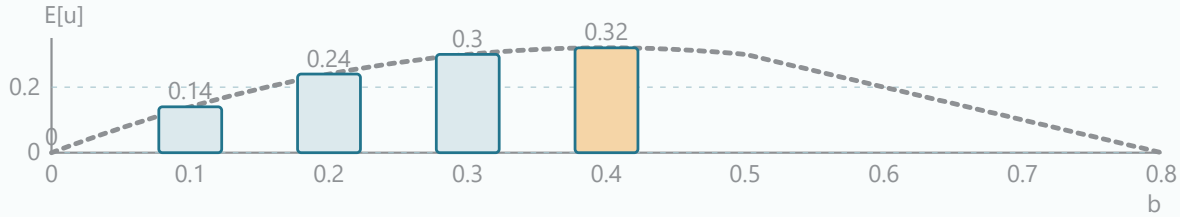
Step 5 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
▶        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.3` `p = 0.6` `eu = 0.3` `best = 0.3` `b_best = 0.3`

Code Trace: `best_bid(0.8, 2)`



b = 0.4: P(win) = 0.8, E[u] = (0.8 − 0.4)·0.8 = **0.32** → new best.

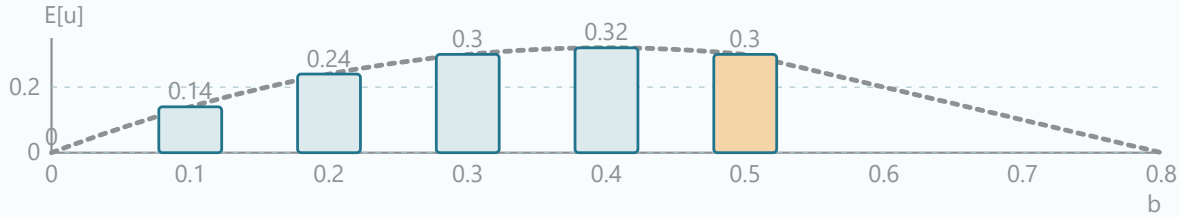
Step 6 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
▶        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.4` `p = 0.8` `eu = 0.32` `best = 0.32` `b_best = 0.4`

Code Trace: `best_bid(0.8, 2)`



b = 0.5: P(win) = 1, E[u] = (0.8 − 0.5)·1 = **0.3** < 0.32: best unchanged.

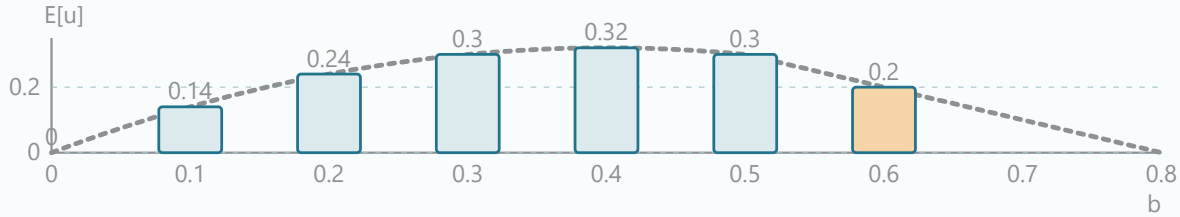
Step 7 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n          # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.5` `p = 1` `eu = 0.3` `best = 0.32` `b_best = 0.4`

Code Trace: `best_bid(0.8, 2)`



b = 0.6: P(win) = 1, E[u] = (0.8 − 0.6)·1 = **0.2** < 0.32: best unchanged.

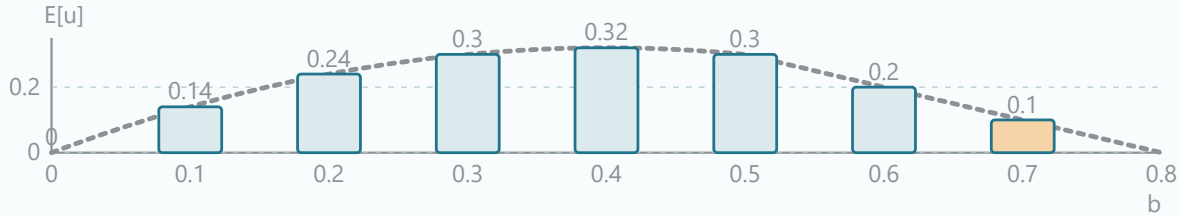
Step 8 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.6` `p = 1` `eu = 0.2` `best = 0.32` `b_best = 0.4`

Code Trace: `best_bid(0.8, 2)`



b = 0.7: P(win) = 1, E[u] = (0.8 − 0.7)·1 = **0.1** < 0.32: best unchanged.

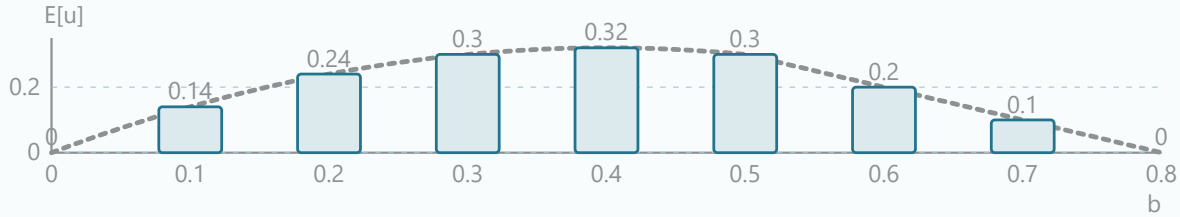
Step 9 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.7` `p = 1` `eu = 0.1` `best = 0.32` `b_best = 0.4`

Code Trace: `best_bid(0.8, 2)`



b = 0.8: P(win) = 1, E[u] = (0.8 − 0.8)·1 = **0** < 0.32: best unchanged.

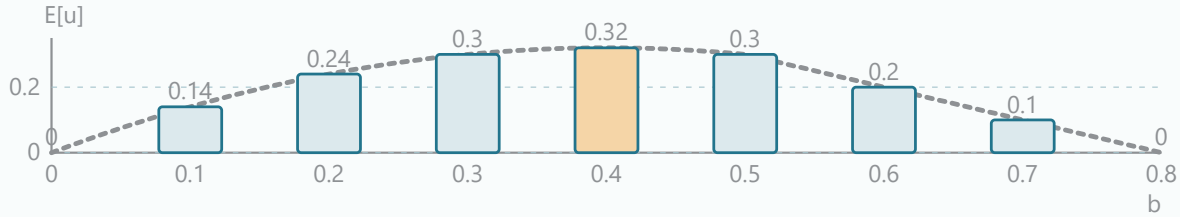
Step 10 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
▶      p = min(1, b / beta) ** (n - 1)  # P(win)
▶      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
11   return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `beta = 0.5` `b = 0.8` `p = 1` `eu = 0` `best = 0.32` `b_best = 0.4`

Code Trace: `best_bid(0.8, 2)`



Result: **b = 0.4 = v/2**, E[u] = 0.32. Bidding low kills the chance of winning; bidding high kills the profit.

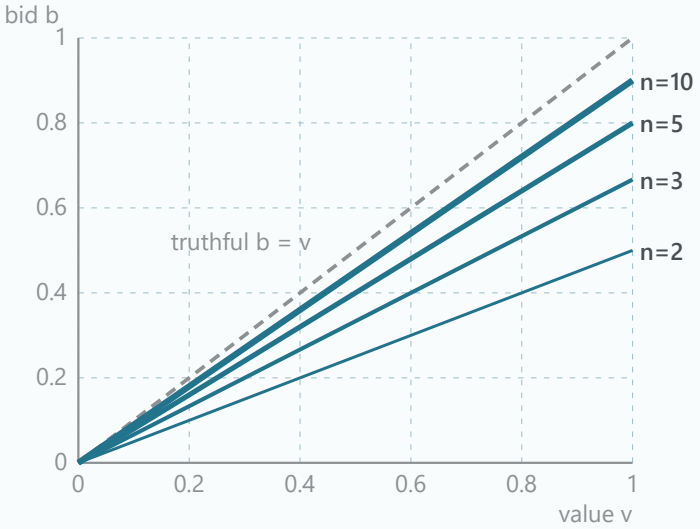
Step 11 / 11

```
1  def best_bid(v, n, step=0.1):
2    beta = (n - 1) / n        # rivals bid beta*v
3    best, b_best = -1, 0
4    b = 0.0
5    while b <= v:
6      p = min(1, b / beta) ** (n - 1)  # P(win)
7      eu = (v - b) * p
8      if eu > best:
9        best, b_best = eu, b
10     b = round(b + step, 10)
▶    return b_best, best
```

VARIABLES

`v = 0.8` `n = 2` `b_best = 0.4` `best = 0.32`

More Competition, Less Shading



n	b(v)	v = 0.8 → b	shading
2	1/2·v	0.4	0.4
3	2/3·v	0.53	0.27
5	4/5·v	0.64	0.16
10	9/10·v	0.72	0.08

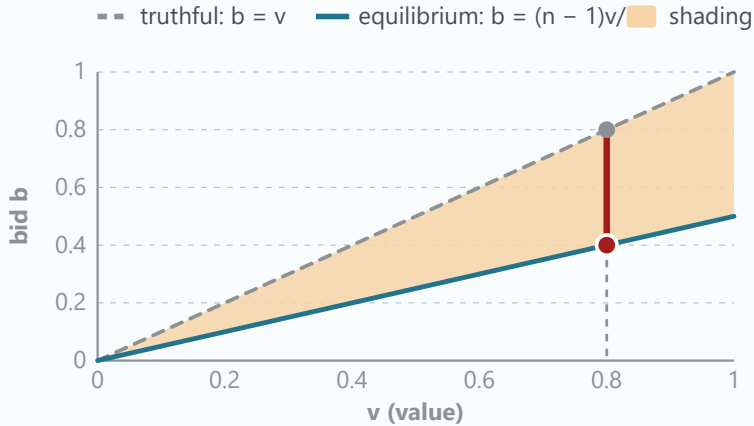
With many bidders, winning requires bidding close to your value; as $n \to \infty$, $b \to v$.

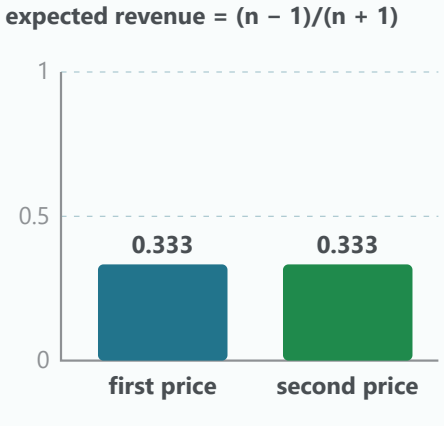
Bid Shading as the Number of Bidders Grows

Equilibrium bid **b(v) = (n − 1)v / n** · shading = v − b(v) = v/n · expected revenue = **(n − 1)/(n + 1)**

n = 2

v = 0.8: bid b = **0.4** | shading v/n = **0.4** | expected revenue = **1/3 ≈ 0.333** (same under both rules)





As the number of bidders grows, competition reduces shading and bids approach values; expected revenue is the same under first and second price (revenue equivalence) and rises with n.

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue

Two bidders with independent values $v_1, v_2 \sim U[0, 1]$. Question: which rule earns the seller more?

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue
Second price	$\min(v_1, v_2)$	$E[\min] = \int_0^1 (1 - t)^2 \, dt = 1/3$

Second price: everyone is truthful and the seller gets the smaller value. The smaller of two values exceeds t only if both do: $P(\min > t) = (1 - t) \cdot (1 - t)$. Using $E[X] = \int_0^1 P(X > t) \, dt$, the expectation is **1/3**.

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue
Second price	$\min(v_1, v_2)$	$E[\min] = \int_0^1 (1 - t)^2 \, dt = 1/3$
First price	$\max(v_1, v_2) / 2$	$E[\max] = \int_0^1 (1 - t^2) \, dt = 2/3 \to 1/3$

First price: the equilibrium bid is $b = v/2$ and the higher value wins. $P(\max < t) = t^2$, so $P(\max > t) = 1 - t^2 \to E[\max] = 2/3$, revenue **1/3**.

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue
Second price	$\min(v_1, v_2)$	$E[\min] = \int_0^1 (1 - t)^2 \, dt = 1/3$
First price	$\max(v_1, v_2) / 2$	$E[\max] = \int_0^1 (1 - t^2) \, dt = 2/3 \rightarrow 1/3$
English	≈ second price	1/3

With private values the English auction gives the same result as second price → 1/3.

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue
Second price	$\min(v_1, v_2)$	$E[\min] = \int_0^1 (1 - t)^2 \, dt = 1/3$
First price	$\max(v_1, v_2) / 2$	$E[\max] = \int_0^1 (1 - t^2) \, dt = 2/3 \to 1/3$
English	≈ second price	1/3
Dutch	≡ first price	1/3

The Dutch auction is strategically equivalent to first price → 1/3. **Four rules, the same expected revenue.**

Step by Step: Expected Revenue with Two Bidders = 1/3

Rule	Seller's revenue	Expected revenue
Second price	$\min(v_1, v_2)$	$E[\min] = \int_0^1 (1 - t)^2 \, dt = 1/3$
First price	$\max(v_1, v_2) / 2$	$E[\max] = \int_0^1 (1 - t^2) \, dt = 2/3 \to 1/3$
English	≈ second price	1/3
Dutch	≡ first price	1/3
Simulation	200,000 draws	Second price 0.333 · first price 0.333

A Monte Carlo simulation confirms it: revenue differs auction by auction but is **equal on average**.

The Revenue Equivalence Theorem

- With independent private values, risk-neutral symmetric bidders and the same allocation (highest value wins), standard auctions yield **the same expected revenue** (Vickrey 1961; Myerson 1981; Riley and Samuelson 1981).
- **Why?** Myerson's payment formula $p(v) = v \cdot x(v) - \int_0^v x(z)\,dz$ fixes the payment from $x(\cdot)$ and the lowest type's payment (0): same $x(\cdot) \rightarrow$ same revenue. For $n = 2$, $U[0, 1]$: $p(v) = v^2 - v^2/2 = v^2/2$; first price: bid $v/2$ × win probability v; second price: v × expected rival value $v/2$. All agree.
- Risk aversion, budgets, common values or asymmetry can break it.

n	Theory: (n−1)/(n+1)	Second price (simulation)	First price (simulation)
2	1/3 = 0.33	0.332	0.333
3	1/2 = 0.5	0.500	0.500
4	3/5 = 0.6	0.600	0.600
5	2/3 = 0.67	0.668	0.667

So formats are chosen for simplicity, transparency and participation, not just revenue.

Change the Assumption: What If Bidders Are Risk-Averse?

Two bidders, values U[0, 1]. The winner's profit is $x = v - \text{price}$ and their utility $\mathbf{u(x) = x^{\rho}}$ ($0 < \rho \leq 1$). $\rho = 1$: risk-neutral; $\rho < 1$: **risk-averse** ($\rho = ½$ gives $u(x) = \sqrt{x}$).

If bidders are risk-averse, **which rule earns the seller more?**

Change the Assumption: What If Bidders Are Risk-Averse?

Two bidders, values U[0, 1]. The winner's profit is x = v − price and their utility $\mathbf{u(x) = x^{\rho}}$ ($0 < \rho \leq 1$). ρ = 1: risk-neutral; ρ < 1: **risk-averse** (ρ = ½ gives u(x) = √x).

If bidders are risk-averse, **which rule earns the seller more?**

Second price: no change

The rival's bid, not yours, sets the price. Truthful bidding is still weakly dominant. Revenue: E[min] = 1/3.

First price: bids rise

Bidders raise their bids to reduce the risk of losing: $\mathbf{b(v) = v / (1 + \rho)}$. ρ = 1: v/2; ρ = ½: 2v/3.

Change the Assumption: What If Bidders Are Risk-Averse?

Two bidders, values U[0, 1]. The winner's profit is x = v − price and their utility $\mathbf{u(x) = x^{\rho}}$ ($0 < \rho \leq 1$). ρ = 1: risk-neutral; ρ < 1: **risk-averse** (ρ = ½ gives u(x) = √x).

If bidders are risk-averse, **which rule earns the seller more?**

Second price: no change

The rival's bid, not yours, sets the price. Truthful bidding is still weakly dominant. Revenue: E[min] = 1/3.

First price: bids rise

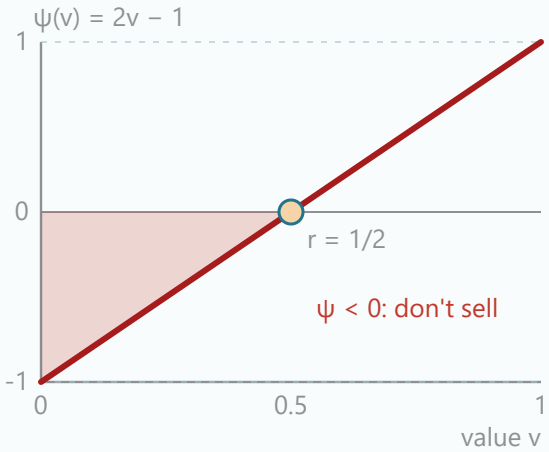
Bidders raise their bids to reduce the risk of losing: **b(v) = v / (1 + ρ)**. ρ = 1: v/2; ρ = ½: 2v/3.

Expected revenue	**ρ = 1**	**ρ = ½**
Second price	1/3 ≈ 0,333	1/3 ≈ 0,333
First price (E[max] = 2/3)	(2/3)·(1/2) = 1/3	**(2/3)·(2/3) = 4/9 ≈ 0,444**

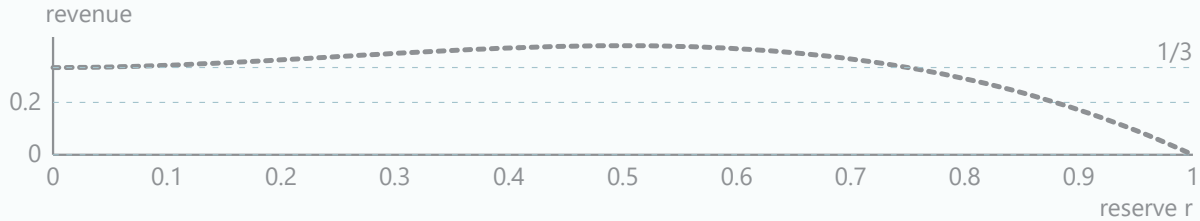
Risk aversion breaks revenue equivalence: first price earns more.

Revenue Maximization: Myerson and the Reserve Price

- What if the seller wants to maximize **expected revenue** rather than efficiency? (Myerson, 1981)
- **Virtual value** $\psi(v) = v - (1 - F(v))/f(v)$. $F(v)$: the probability that the value is below v; $f(v)$: the density. For $U[0, 1]$, $F(v) = v$, $f(v) = 1 \Rightarrow \psi(v) = 2v - 1$.
- Expected revenue equals the expected virtual value of the winner. The optimal mechanism: give the item to the largest ψ, and do not sell if $\psi < 0$.
- If ψ is increasing and bidders are symmetric, this is a **second-price auction with a reserve price**: $\psi(r) = 0 \Rightarrow 2r - 1 = 0 \Rightarrow r = 1/2$ (independent of n).
- With two bidders and reserve r, expected revenue is $1/3 + r^2 - 4r^3/3$; at $r = 1/2$ it rises from $1/3 \to$ **5/12 ≈ 0.417**. The cost: sometimes nothing is sold (an efficiency loss).



Code Trace: `best_reserve()`



Try reserve prices on the grid r = 0, 0.1, …, 1; `revenue(r)` averages the payment over a 100 × 100 grid of values.

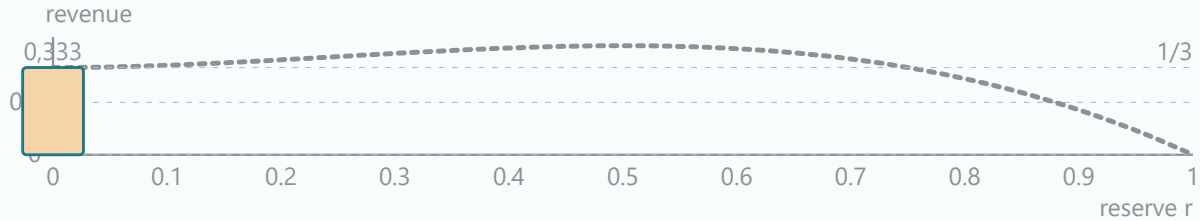
Step 1 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2      # mean payment
6  def best_reserve(step=0.1):
▶    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
9      R = revenue(k * step)
10     if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`best = -1` `r_best = None`

Code Trace: `best_reserve()`



r = 0: revenue(r) = **0.333** → new best. No reserve = second price: 1/3.

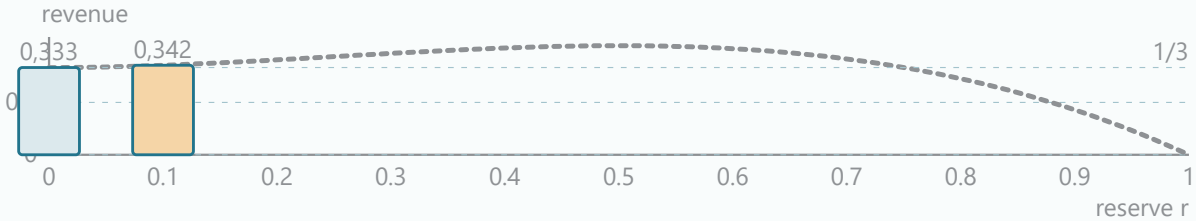
Step 2 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2      # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 0` `R = 0.333` `best = 0.333` `r_best = 0`

Code Trace: `best_reserve()`



r = 0.1: revenue(r) = **0.342** → new best.

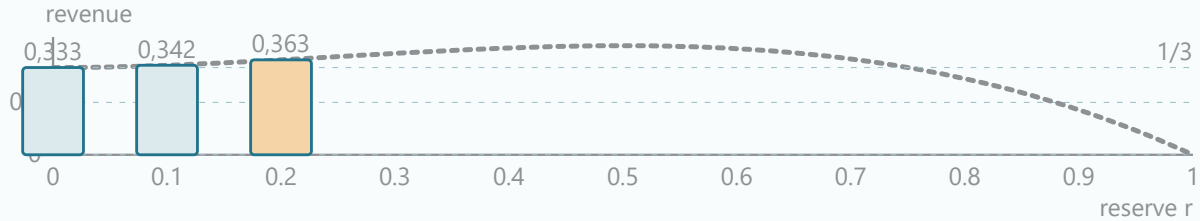
Step 3 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 1` `R = 0.342` `best = 0.342` `r_best = 0.1`

Code Trace: `best_reserve()`



r = 0.2: revenue(r) = **0.363** → new best.

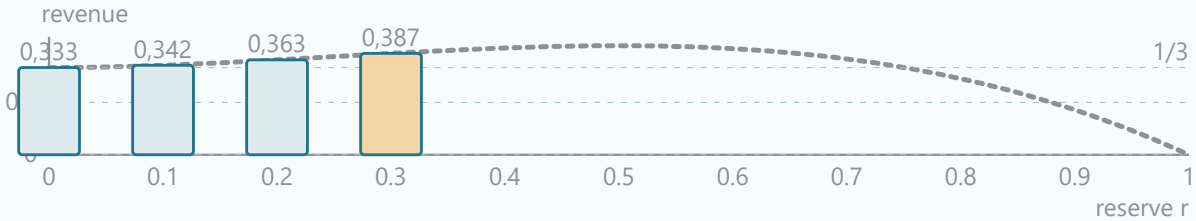
Step 4 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 2` `R = 0.363` `best = 0.363` `r_best = 0.2`

Code Trace: `best_reserve()`



r = 0.3: revenue(r) = **0.387** → new best.

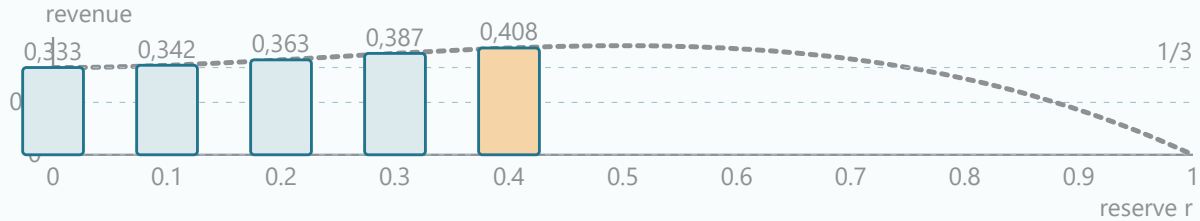
Step 5 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 3` `R = 0.387` `best = 0.387` `r_best = 0.3`

Code Trace: `best_reserve()`



r = 0.4: revenue(r) = **0.408** → new best.

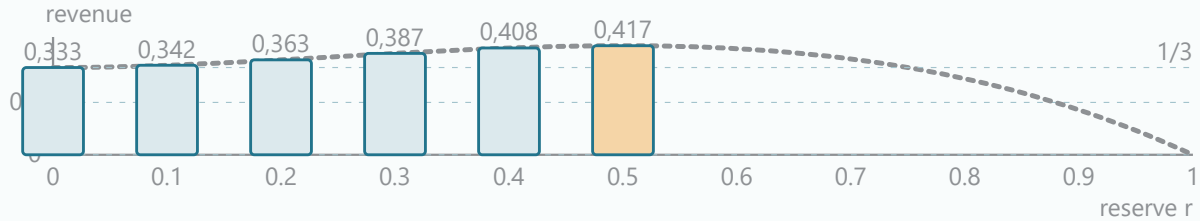
Step 6 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 4` `R = 0.408` `best = 0.408` `r_best = 0.4`

Code Trace: `best_reserve()`



r = 0.5: revenue(r) = **0.417** → new best.

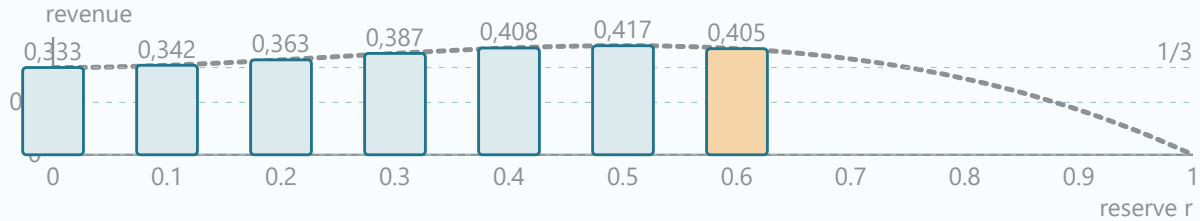
Step 7 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 5` `R = 0.417` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



r = 0.6: revenue(r) = **0.405** < 0.417.

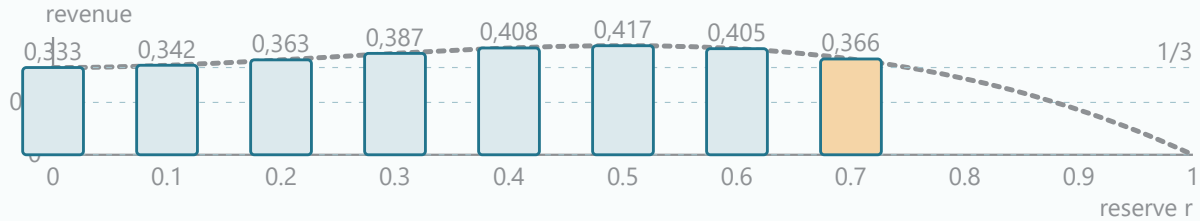
Step 8 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 6` `R = 0.405` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



r = 0.7: revenue(r) = **0.366** < 0.417.

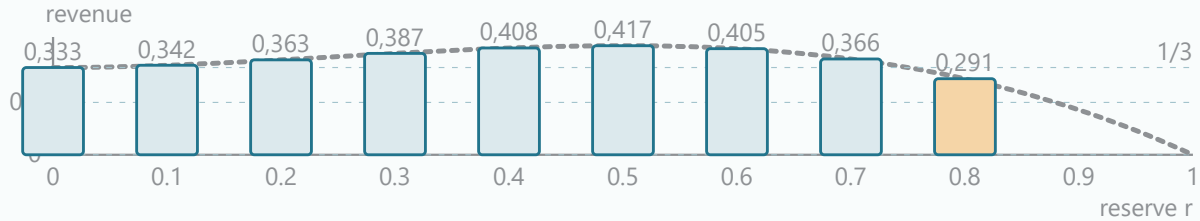
Step 9 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 7` `R = 0.366` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



r = 0.8: revenue(r) = **0.291** < 0.417.

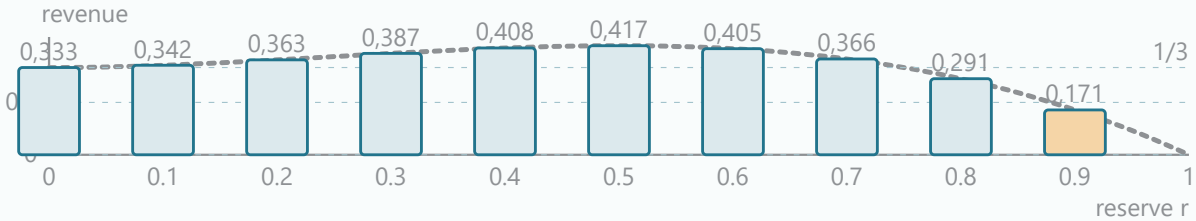
Step 10 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 8` `R = 0.291` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



r = 0.9: revenue(r) = **0.171** < 0.417.

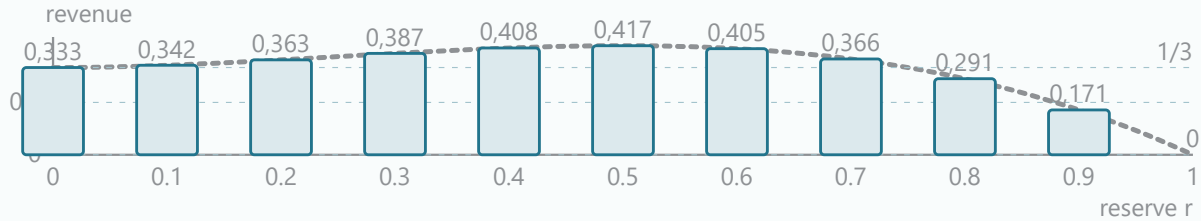
Step 11 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 9` `R = 0.171` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



r = 1: revenue(r) = **0** < 0.417.

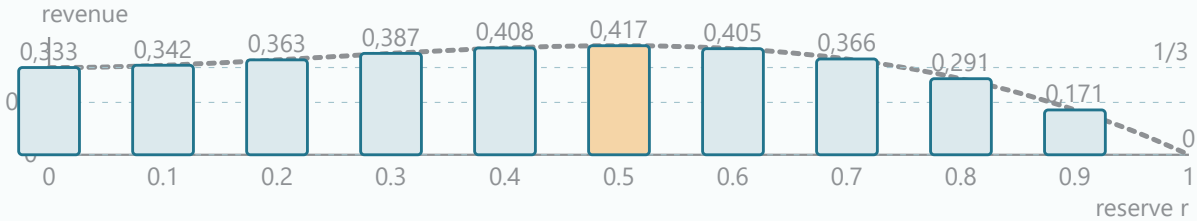
Step 12 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
▶      R = revenue(k * step)
▶      if R > best: best, r_best = R, k * step
11   return r_best, best
```

VARIABLES

`k = 10` `R = 0` `best = 0.417` `r_best = 0.5`

Code Trace: `best_reserve()`



The best reserve is **r = 0.5**: revenue ≈ 0.417 = 5/12 > 1/3. A high r often leaves the item unsold.

Step 13 / 13

```
1  def revenue(r, m=100):   # 2 bidders, U[0.1]
2    v = [(i + .5) / m for i in range(m)]
3    total = sum(max(min(a, c), r) for a in v
4                for c in v if max(a, c) >= r)
5    return total / m**2     # mean payment
6  def best_reserve(step=0.1):
7    best, r_best = -1, None
8    for k in range(round(1/step) + 1):
9      R = revenue(k * step)
10     if R > best: best, r_best = R, k * step
▶    return r_best, best
```

VARIABLES

`r_best = 0.5` `best = 0.417`

The Cost of a Reserve: Revenue Up, Efficiency Down

Two bidders, values U[0, 1], a second-price auction with reserve price r. If both values are below r, the item is not sold.

r	Expected revenue $1/3 + r^2 - 4r^3/3$	Probability of no sale r^2	Expected welfare loss $2r^3/3$
0	1/3 ≈ 0,333	0	0
0.25	3/8 = 0,375	1/16 = 6.25%	1/96 ≈ 0,010
0.5	**5/12 ≈ 0,417**	**1/4 = 25%**	**1/12 ≈ 0,083**
0.75	1/3 ≈ 0,333	9/16 = 56.25%	9/32 ≈ 0,281

- **No sale:** both values below r: $r \cdot r = r^2$.
- **Welfare loss:** the item should go to the highest-value bidder; if that value is below r, it is lost. The highest value has density 2t: loss $\int_0^r t \cdot 2t\, dt = 2r^3/3$.
- Welfare without a reserve: E[max] = 2/3.

r = 0.5 maximizes revenue, but the item goes unsold with probability 25% and expected welfare falls from 2/3 to 7/12 (a 12.5% loss).

All-Pay and Multi-Unit Auctions

All-pay auction

The highest bid wins but *everyone* pays their bid: lobbying, R&D races, election campaigns.
Two bidders, U[0, 1]: in equilibrium $b(v) = v^2/2$.
Value 0.8 → bid 0.32 (0.4 under first price).
Types with little chance of winning bid almost nothing.
Expected revenue is again 1/3 (**revenue equivalence**): a type-v bidder's expected payment is $v^2/2$, as under first and second price.

Multi-unit auctions

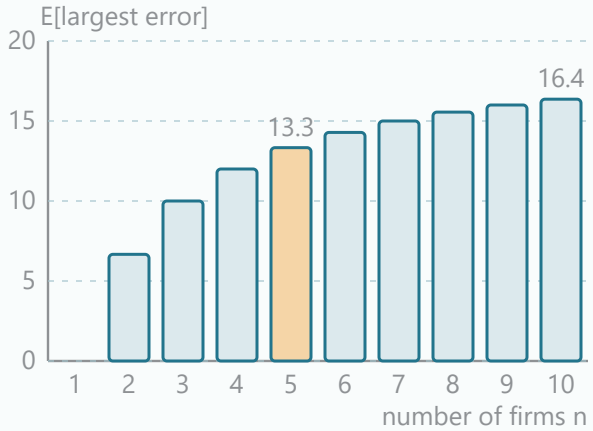
With identical units (government bonds), winners pay either a uniform price per unit or their own bids; for items valuable together (spectrum licenses), package bids are used.

Further reading: Bulow and Klemperer (1996): attracting one extra bidder earns as much as the optimal reserve price, and often more.

The all-pay model describes contests with a single prize winner: when values are equal and commonly known, expected total spending equals the value of the prize.

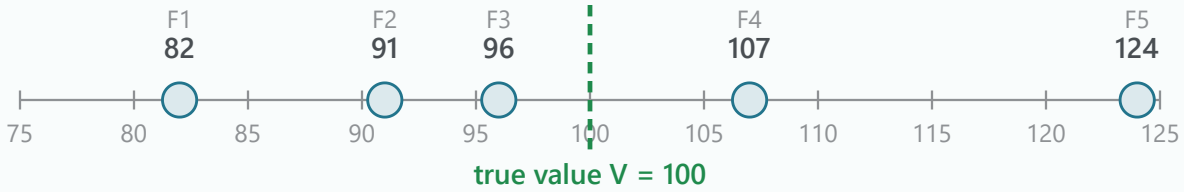
The Winner's Curse

- In a common-value auction, the highest bid comes from the bidder who **overestimates** the value the most.
- If the winner ignores their estimation error, they may pay more than the item's true value.
- Even if estimates are right on average, the *largest* estimate is systematically too high.
- To reduce this risk, bidders bid below their signals and take the quality of their information into account.
- First documented in Gulf of Mexico oil-lease auctions (Capen, Clapp, Campbell, 1971).



Estimation errors U[−20, 20]: the winner's expected overestimate is 20(n−1)/(n+1). More rivals, bigger curse (simulation gives the same values).

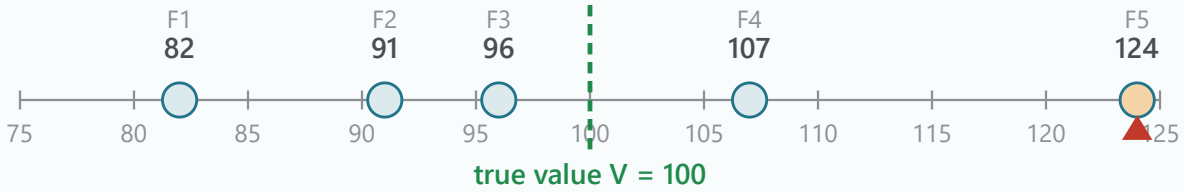
Step by Step: An Oil Lease Auction



Case	Winner / payment	Profit

The field's true value is V = 100 (nobody knows it). Five firms estimate 82, 91, 96, 107, 124. The mean is **100**: the estimates are right on average.

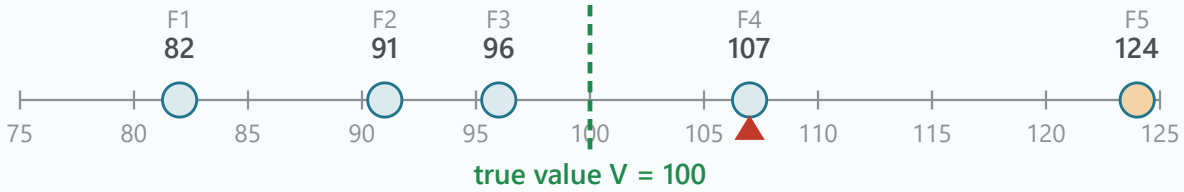
Step by Step: An Oil Lease Auction



Case	Winner / payment	Profit
First price, unshaded bids	F5 → 124	**−24**

If everyone bids their estimate (first price), the most optimistic, F5, wins and pays 124 → profit **−24**.

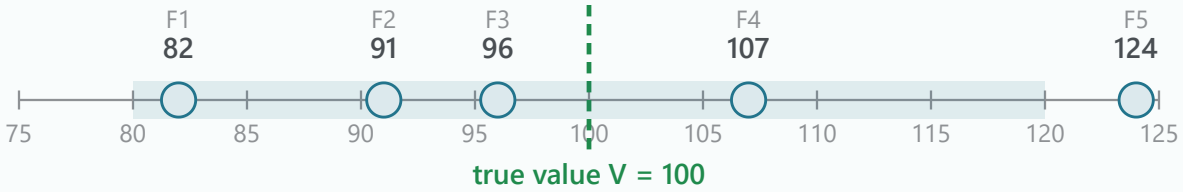
Step by Step: An Oil Lease Auction



Case	Winner / payment	Profit
First price, unshaded bids	F5 → 124	**−24**
Second price, unshaded bids	F5 → 107	**−7**

Even under second price F5 pays 107 → profit **−7**. Winning means answering yes to “am I the one who overestimated most?”

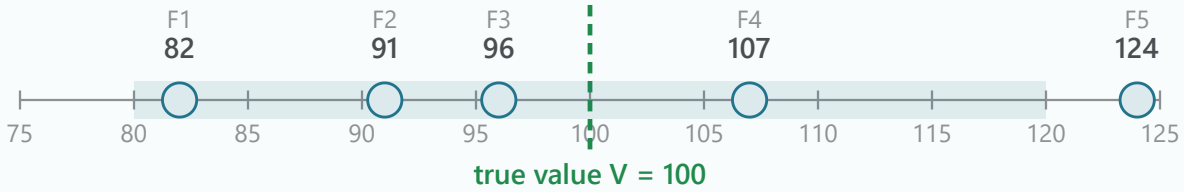
Step by Step: An Oil Lease Auction



Case	Winner / payment	Profit
First price, unshaded bids	F5 → 124	**−24**
Second price, unshaded bids	F5 → 107	**−7**
Expected largest error (n = 5)	20·4/6	**≈ 13.3**

With errors U[−20, 20], the largest of five estimates is off by 20·4/6 ≈ **13.3** on average. Bidding your estimate under first price gives expected profit ≈ −13.3; under second price you pay the second-largest estimate, whose error averages 20·(n − 3)/(n + 1) = 20·2/6 ≈ 6.7, so ≈ −6.7.

Step by Step: An Oil Lease Auction



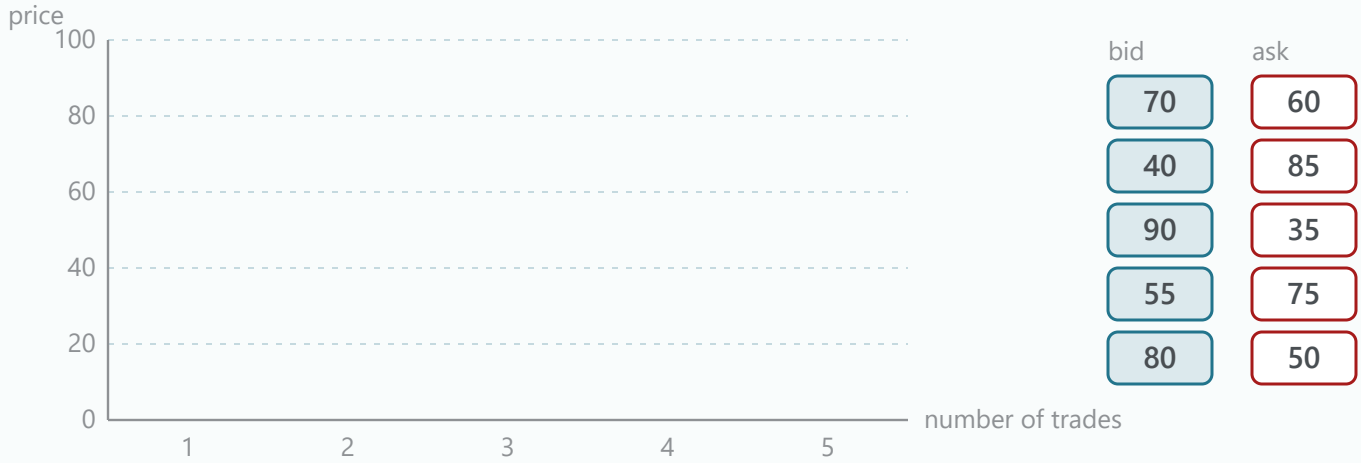
Case	Winner / payment	Profit
First price, unshaded bids	F5 → 124	**−24**
Second price, unshaded bids	F5 → 107	**−7**
Expected largest error (n = 5)	20·4/6	**≈ 13.3**
Simulation, first price, unshaded bids	mean	**−13.3**
Simulation, first price, shaded by 13.3	mean	**0**

First-price simulation (100,000 auctions): bidding your estimate averages a profit of −13.3; shading by 13.3 brings it to ≈ 0. With 10 firms the required shading is 16.4: more rivals, shade **more**.

Double Auctions

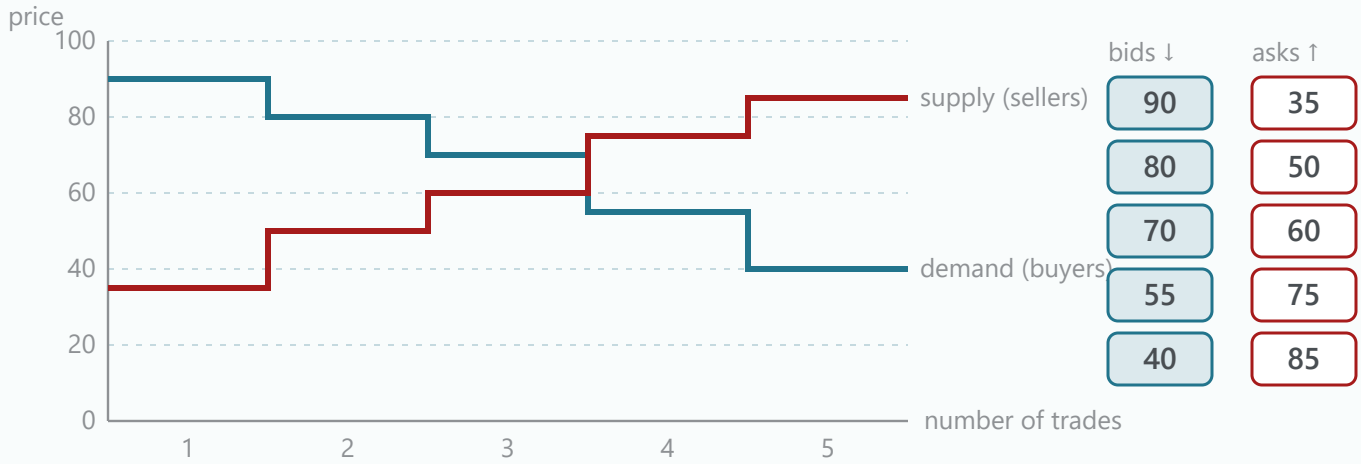
- Many buyers and many sellers submit orders at once: buyers post **bids**, sellers post **asks**.
- Stock exchanges, electricity markets and carbon permit markets work this way.
- Matching: sort bids in decreasing and asks in increasing order; pairs match as long as bid ≥ ask.
- Price: a single market price is chosen from the interval between the last matched pair and the first unmatched pair (e.g. the midpoint).
- Total surplus (buyer + seller surplus) = Σ (bid − ask) over matched pairs → the largest possible value.

Step by Step: Matching in a Double Auction



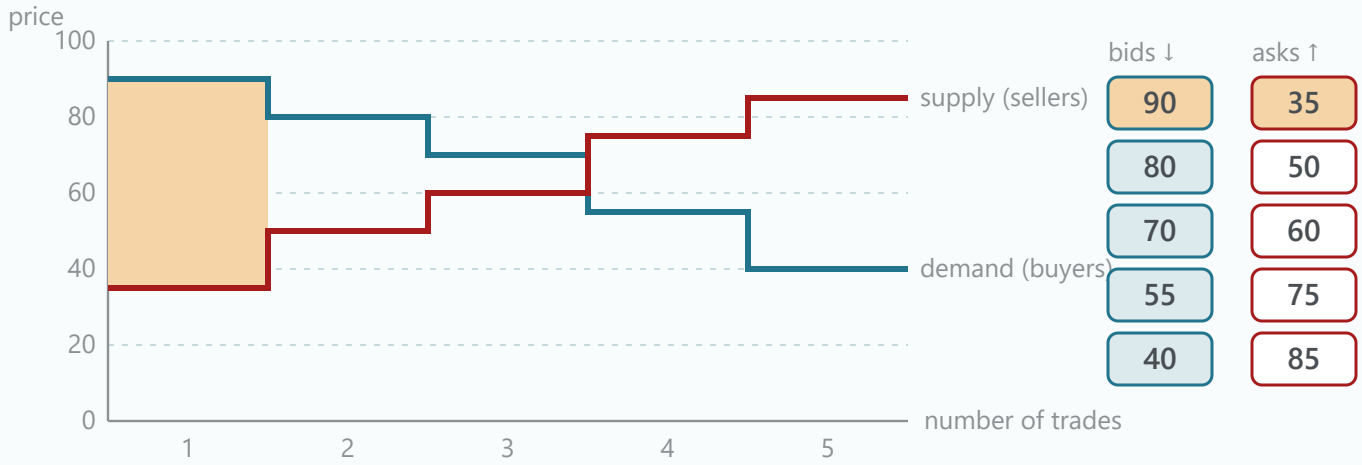
Order book: buyers are willing to pay 70, 40, 90, 55, 80; sellers to accept 60, 85, 35, 75, 50.

Step by Step: Matching in a Double Auction



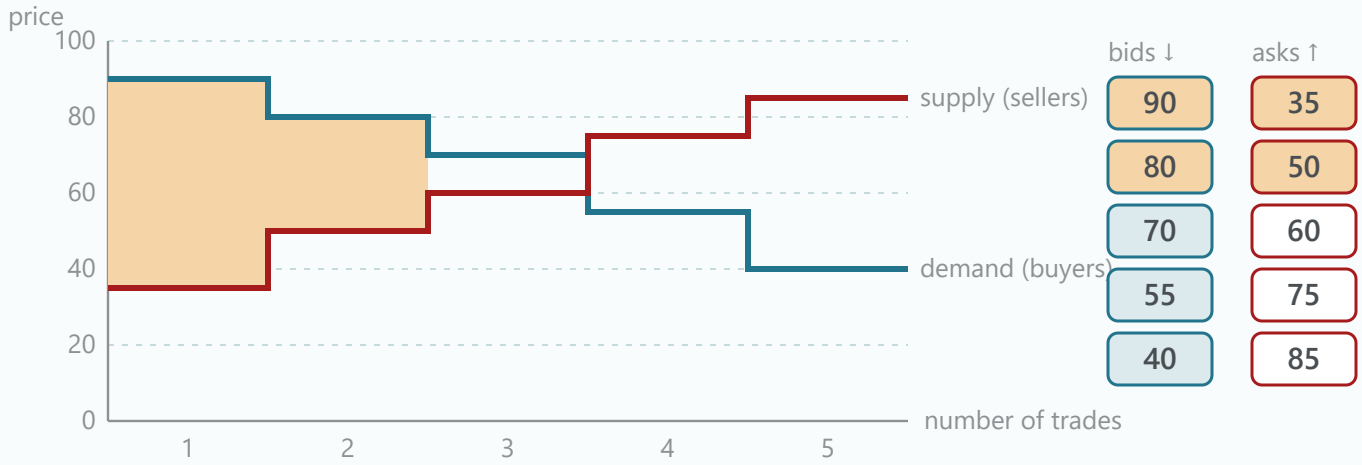
Bids are sorted **decreasing** (the demand curve), asks **increasing** (the supply curve).

Step by Step: Matching in a Double Auction



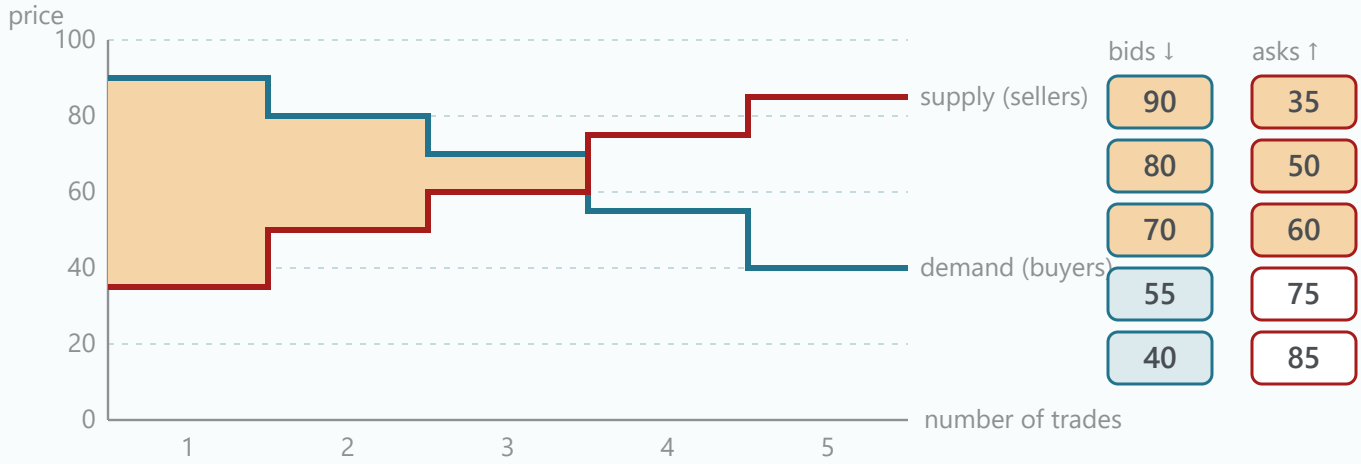
Pair 1: 90 ≥ 35 → **match**; this trade creates surplus 90 − 35 = 55.

Step by Step: Matching in a Double Auction



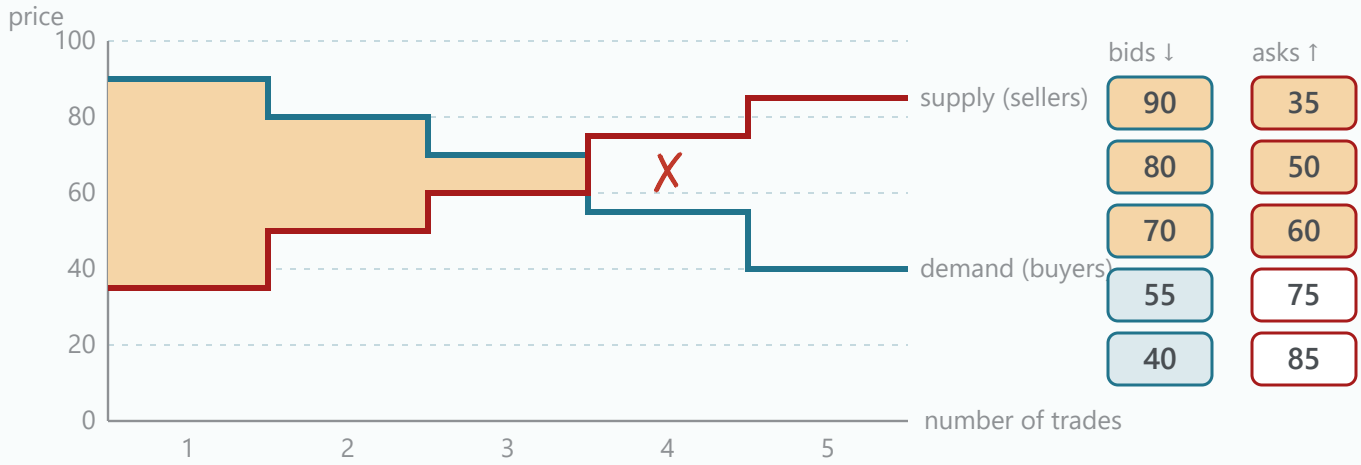
Pair 2: 80 ≥ 50 → **match**; this trade creates surplus 80 − 50 = 30.

Step by Step: Matching in a Double Auction



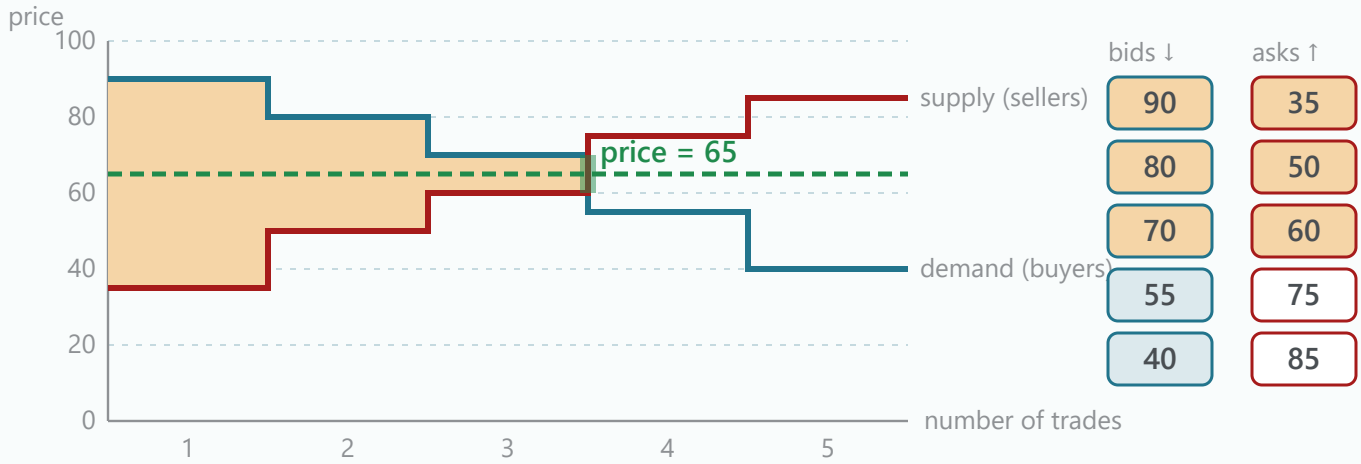
Pair 3: 70 ≥ 60 → **match**; this trade creates surplus 70 − 60 = 10.

Step by Step: Matching in a Double Auction



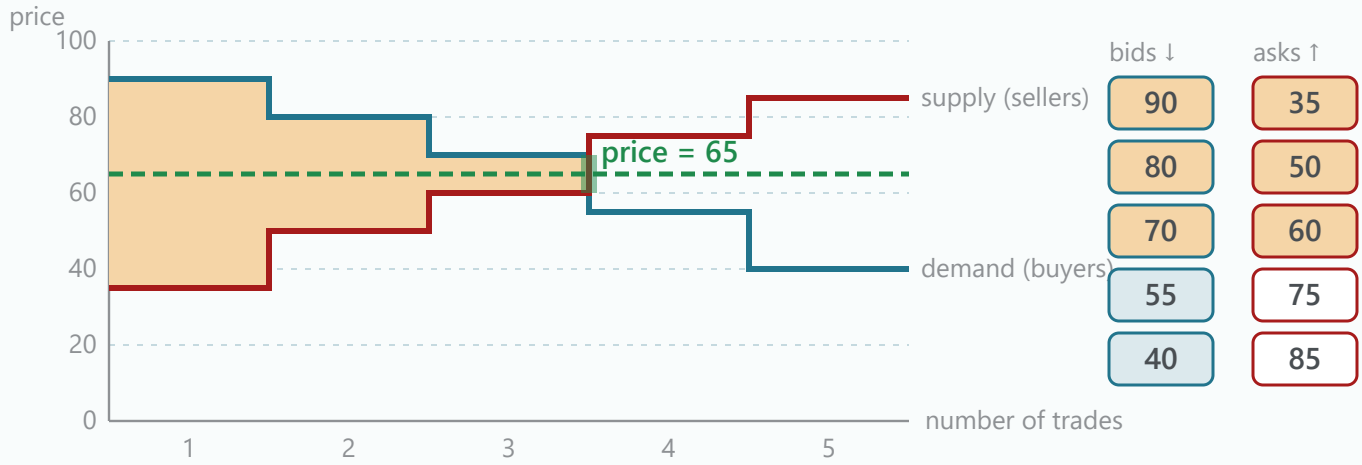
Pair 4: 55 < 75 → the buyer won't pay what the seller asks; **stop**. 3 trades take place.

Step by Step: Matching in a Double Auction



Price interval: [max(60, 55), min(70, 75)] = [60, 70]. The midpoint is **price = 65**; every matched trader trades at this price.

Step by Step: Matching in a Double Auction



Buyer surplus (90−65)+(80−65)+(70−65) = 45, seller surplus (65−35)+(65−50)+(65−60) = 50; total **95** = the largest possible total surplus.

Code Trace: `double_auction(bids, asks)`

b	90	80	70	55	40
s	60	85	35	75	50

Sort bids from high to low: the most eager buyer first.

Step 1 / 14

```
1  def double_auction(bids, asks):
▶    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

VARIABLES

`bids = [70, 40, 90, 55, 80]`

`b = [90, 80, 70, 55, 40]`

Code Trace: `double_auction(bids, asks)`

b	90	80	70	55	40
s	35	50	60	75	85

Asks from low to high: the cheapest seller first.

Step 2 / 14

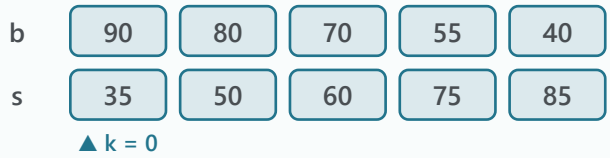
```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
▶    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

VARIABLES

b = [90, 80, 70, 55, 40]

s = [35, 50, 60, 75, 85]

Code Trace: `double_auction(bids, asks)`



k: number of matched pairs; n: length of the shorter list.

Step 3 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
▶    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

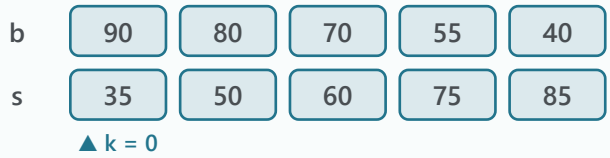
VARIABLES

b = [90, 80, 70, 55, 40]

s = [35, 50, 60, 75, 85] k = 0

n = 5

Code Trace: `double_auction(bids, asks)`



Condition: b[0] = 90 ≥ s[0] = 35? **Yes**.

Step 4 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
▶    while k < n and b[k] >= s[k]:
6      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

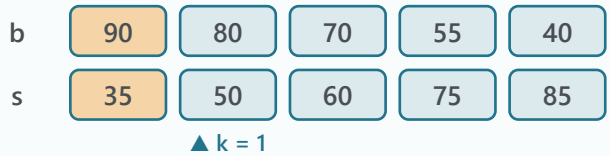
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 0`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Pair matched → **k = 1**.

Step 5 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
▶      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

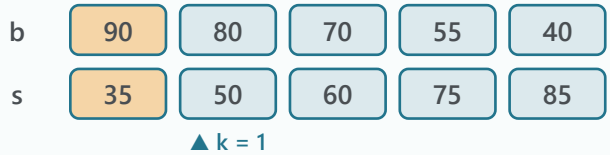
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 1`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Condition: b[1] = 80 ≥ s[1] = 50? **Yes**.

Step 6 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
▶    while k < n and b[k] >= s[k]:
6      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

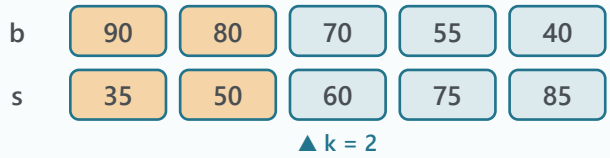
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 1`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Pair matched → **k = 2**.

Step 7 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
▶      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

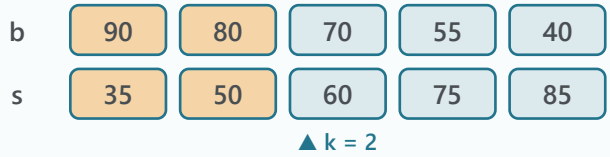
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 2`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Condition: b[2] = 70 ≥ s[2] = 60? **Yes**.

Step 8 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
▶    while k < n and b[k] >= s[k]:
6      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

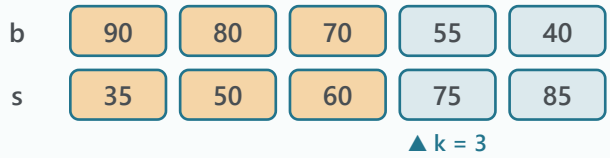
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 2`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Pair matched → **k = 3**.

Step 9 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
▶      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

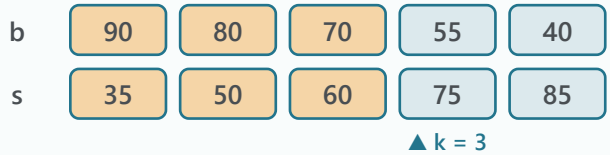
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 3`

`n = 5`

Code Trace: `double_auction(bids, asks)`



Condition: b[3] = 55 ≥ s[3] = 75? **No**. The loop ends.


```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
▶    while k < n and b[k] >= s[k]:
6      k += 1                      # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

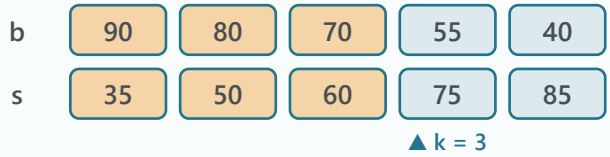
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 3`

`n = 5`

Code Trace: `double_auction(bids, asks)`



k ≠ 0: at least one trade, so compute the price.

Step 11 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                      # pair matched
▶    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

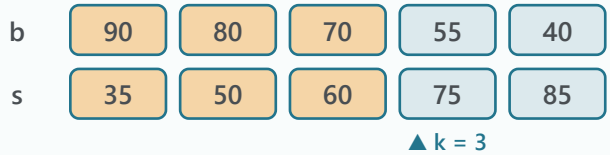
VARIABLES

`b = [90, 80, 70, 55, 40]`

`s = [35, 50, 60, 75, 85]` `k = 3`

`n = 5`

Code Trace: `double_auction(bids, asks)`



lo = max(s[k−1], b[k]) = max(60, 55) = **60**: the last seller must not lose, and the excluded buyer must not want to buy.

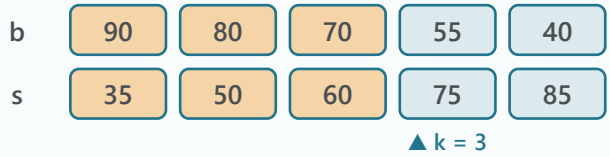
Step 12 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
▶    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

VARIABLES

`k = 3` `lo = 60`

Code Trace: `double_auction(bids, asks)`



hi = min(b[k−1], s[k]) = min(70, 75) = **70**.

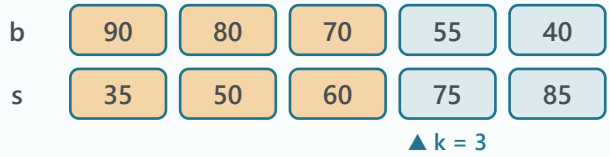
Step 13 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
▶    hi = min([b[k-1]] + s[k:k+1])
11   return k, (lo + hi) / 2
```

VARIABLES

`k = 3` `lo = 60` `hi = 70`

Code Trace: `double_auction(bids, asks)`



Result: **3 trades at price (60 + 70)/2 = 65**.

Step 14 / 14

```
1  def double_auction(bids, asks):
2    b = sorted(bids, reverse=True)  # buyers
3    s = sorted(asks)                # sellers
4    k, n = 0, min(len(b), len(s))
5    while k < n and b[k] >= s[k]:
6      k += 1                    # pair matched
7    if k == 0:
8      return 0, None
9    lo = max([s[k-1]] + b[k:k+1])
10   hi = min([b[k-1]] + s[k:k+1])
▶    return k, (lo + hi) / 2
```

VARIABLES

`k = 3` `lo = 60` `hi = 70` `p = 65.0`

Incentives in Two-Sided Markets

Impossibility

With independent values and costs whose ranges overlap, bilateral trade cannot achieve efficiency + IC + IR + budget balance together (Myerson–Satterthwaite, 1983).

The midpoint rule

Efficient and budget-balanced, but not IC: the marginal buyer can shade their bid to pull the price down.

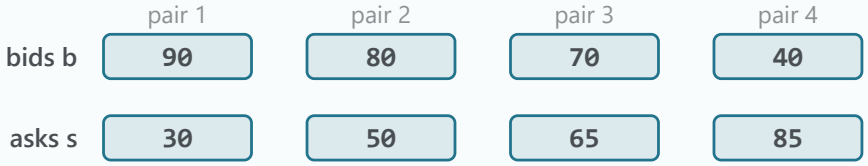
McAfee's rule (1992)

Let p_0 be the average of the first excluded buyer and seller. If p_0 falls between the last matched buyer and seller, everyone trades at p_0; otherwise the last (least profitable) pair is dropped. Truthfulness is dominant; the cost is sometimes one trade.

In our example

$p_0 = (55 + 75)/2 = 65 \in [60, 70] \rightarrow$ all three trades happen at 65; no trade is lost.

Code Trace: `mcafee(bids, asks)`



A new example (buyers 90, 80, 70, 40; sellers 30, 50, 65, 85): bids sorted decreasing, asks increasing; sentinels (−∞ and ∞) are appended.

Step 1 / 11

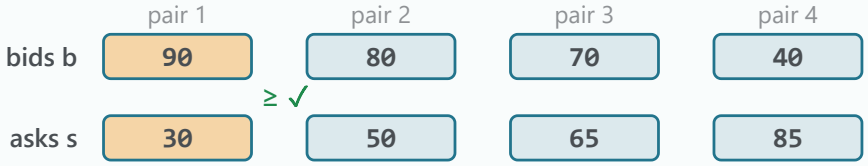
```
1  def mcafee(bids, asks):
▶    b = sorted(bids, reverse=True) + [float('-inf')]
▶    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]`

Code Trace: `mcafee(bids, asks)`



b[0] = 90 ≥ s[0] = 30? **Yes**: an efficient pair.

Step 2 / 11

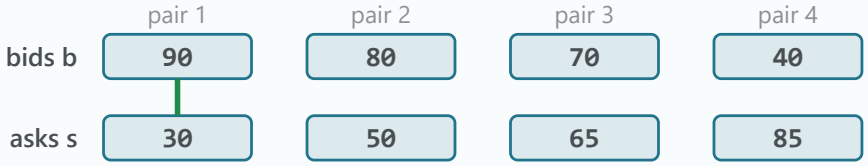
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
▶    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 0`

Code Trace: `mcafee(bids, asks)`



k = 1: 1 efficient trade found.

Step 3 / 11

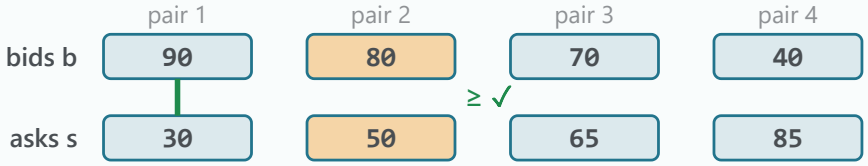
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
▶      k += 1                  # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0        # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 1`

Code Trace: `mcafee(bids, asks)`



b[1] = 80 ≥ s[1] = 50? **Yes**: an efficient pair.

Step 4 / 11

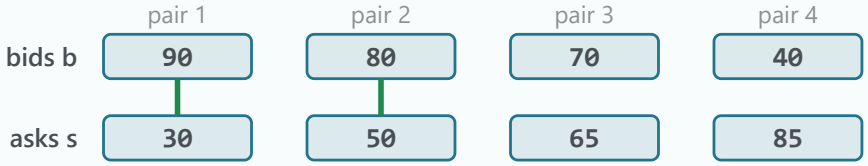
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
▶    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 1`

Code Trace: `mcafee(bids, asks)`



k = 2: 2 efficient trades found.

Step 5 / 11

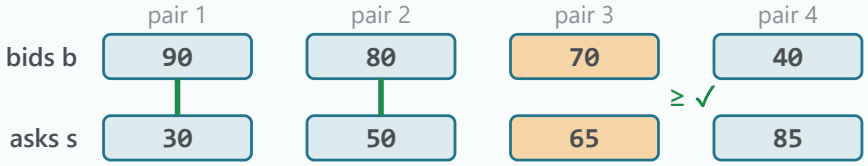
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
▶      k += 1                  # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0        # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 2`

Code Trace: `mcafee(bids, asks)`



b[2] = 70 ≥ s[2] = 65? **Yes**: an efficient pair.

Step 6 / 11

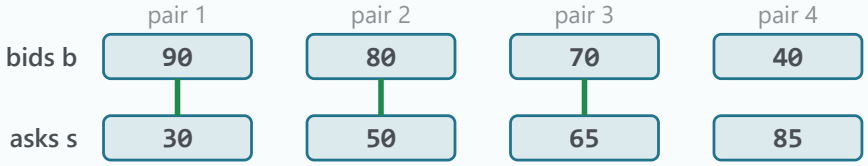
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
▶    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 2`

Code Trace: `mcafee(bids, asks)`



k = 3: 3 efficient trades found.

Step 7 / 11

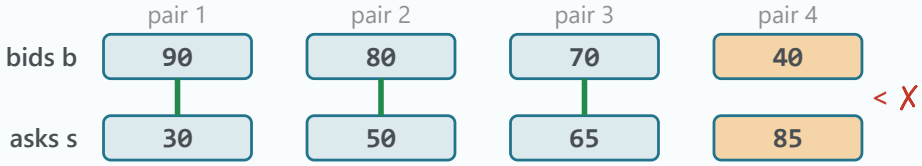
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
▶      k += 1                  # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 3`

Code Trace: `mcafee(bids, asks)`



b[3] = 40 ≥ s[3] = 85? **No**: the loop ends.

Step 8 / 11

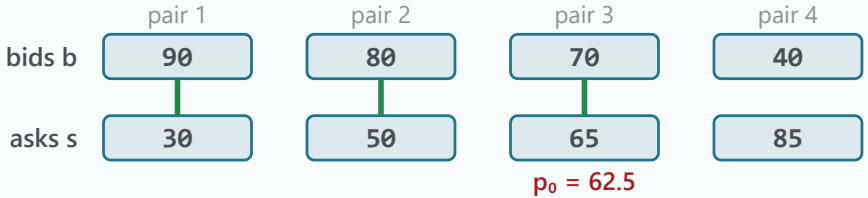
```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
▶    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 3`

Code Trace: `mcafee(bids, asks)`



Candidate price from the first excluded pair: $p_0 = (40 + 85)/2 = \mathbf{62.5}$.

Step 9 / 11

```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
6      k += 1                  # efficient trade
7    if k == 0: return 0, None, None
▶    p0 = (b[k] + s[k]) / 2    # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0        # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

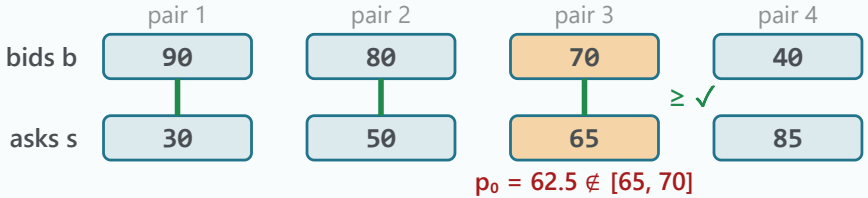
VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 3`

`p0 = 62.5`

Code Trace: `mcafee(bids, asks)`



Is $p_0 \in [s[2], b[2]] = [65, 70]$? **No** (62.5 < 65): the 3rd seller would lose money.

Step 10 / 11

```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
6      k += 1                  # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2    # excluded pair
▶    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0        # one price
11   return k - 1, b[k-1], s[k-1]  # two prices
```

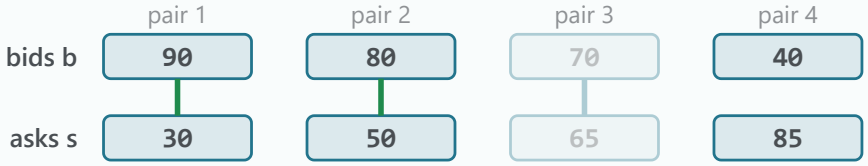
VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 3`

`p0 = 62.5`

Code Trace: `mcafee(bids, asks)`



Drop the weakest pair (70, 65): **2 trades**; buyers pay 70, sellers receive 65. The leftover 2·5 = 10 stays with the market maker.

Step 11 / 11

```
1  def mcafee(bids, asks):
2    b = sorted(bids, reverse=True) + [float('-inf')]
3    s = sorted(asks) + [float('inf')]
4    k = 0
5    while b[k] >= s[k]:
6      k += 1                # efficient trade
7    if k == 0: return 0, None, None
8    p0 = (b[k] + s[k]) / 2   # excluded pair
9    if s[k-1] <= p0 <= b[k-1]:
10     return k, p0, p0       # one price
▶    return k - 1, b[k-1], s[k-1]  # two prices
```

VARIABLES

`b = [90, 80, 70, 40, -inf]`

`s = [30, 50, 65, 85, inf]` `k = 3`

`p0 = 62.5`

Auctions and Market Design

- Market design decides who gets matched with whom, and under which rules.
- The core goals are efficiency, incentive compatibility, fairness, stability and practicality; rules should be strategyproof.
- Auctions are market-design tools that allocate scarce resources by prices and bids.
- Electricity markets, emission permits, procurement and spectrum licenses use prices; in markets without money, priority and matching rules take the place of prices.
- There can be trade-offs among revenue, efficiency, simplicity and strategyproofness.

From Theory to Practice: A Brief History

An ancient institution

Herodotus describes auctions in Babylon in the 5th century BC. In AD 193 the Praetorian Guard auctioned the Roman throne to Didius Julianus.

Theory

Vickrey (1961) compared the four formats and found revenue equivalence; Myerson (1981) characterized the revenue-maximizing auction. Nobel: Vickrey 1996, Myerson 2007.

Spectrum auctions

In 1994 the US FCC ran its first spectrum auction using the simultaneous multi-round ascending design proposed by Milgrom, Wilson and McAfee. Milgrom and Wilson: Nobel 2020.

Matching markets

Roth and Shapley (Nobel 2012): stable matching and exchange rules for markets that do not use prices.

The designer's tools: allocation and payment rules, **information disclosure** (which bids are shown, and when?), entry conditions. Designs are tested in lab experiments and with field data before deployment.

ASSESSMENT

Why Are Vickrey Auctions Rarely Used?

- **Trust**: bidders must trust that the second-highest bid is reported honestly (risk of shill bids).
- **Intuition**: “pay the second price” feels unnatural to most people; overbidding is common in the lab.
- **Model assumptions**: with common or interdependent values, truthful bidding may no longer be dominant.
- **Multiple items**: budget constraints and complementary goods complicate the payment rule (VCG).
- **Seller concerns**: the risk of a low realized price or collusive bidding.

So practical designs often balance incentives for truthfulness against ease of use (e.g. the English auction reaches the same outcome transparently). Details: Rothkopf, Teisberg and Kahn (1990); Ausubel and Milgrom (2006).

ASSESSMENT

Discussion: Which Auction, and How Fair?

Which format would you rather join?

- English: transparent, and seeing the price step by step is an advantage.
- Vickrey: truthful bidding is best; no need to strategize.
- Dutch / first price: fast, but you must guess how much to shade.
- The choice depends on personal strategy and appetite for risk.

Are auctions and market design fair?

- Rules are fixed in advance → predictability; manipulation can be reduced.
- But deep-pocketed players have an advantage; unequal access and information asymmetry matter.
- Seller revenue and equal opportunity are different goals; “perfect fairness for all” is not guaranteed.

ASSESSMENT

Summary: The Four Formats Side by Side

Format	Pays	Equilibrium bid (IPV, U[0.1], n bidders)	Equivalent to
English	price at which the last rival drops out	stay in up to your value	second price
Second price	second-highest bid	b = v (weakly dominant)	English
Dutch	the accepted price	b = (n−1)v/n	first price
First price	own bid	b = (n−1)v/n	Dutch

Revenue equivalence

All four have expected revenue (n−1)/(n+1); 1/3 with two bidders.

Winner's curse

With common values the winner may overpay; shade more as the number of rivals grows.

Double auctions

Match bids in decreasing and asks in increasing order; the price comes from [lo, hi].

ASSESSMENT

Sample Problems

1. Limits 40, 65, 80, 90. Under the four rules, who wins and what do they pay (everyone bids their limit)?
2. n = 4 bidders, values U[0, 1]. What is the first-price equilibrium bid of a bidder with value 0.6?
3. Three bidders, U[0, 1]: what is the expected revenue of a second-price auction?
4. In Vickrey your value is 50 and the rivals' highest bid is 55. What happens if you bid 60?
5. Buyers 100, 70, 45; sellers 30, 60, 80. How many trades, and what is the midpoint price?
6. Two bidders, U[0, 1], second price, reserve r = 0.4. What is the expected revenue? More than with no reserve?

▶ **Show answers**

References (1/2)

- V. Krishna, *Auction Theory*, 2nd ed., Academic Press, 2010 · P. Milgrom, *Putting Auction Theory to Work*, Cambridge University Press, 2004.
- T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, Cambridge University Press, 2016; course page: CS364A · N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge University Press, 2007 (Chapters 9 and 13).
- A. Karlin, Y. Peres, *Game Theory, Alive*, American Mathematical Society, 2017 (auction chapters) · D. Easley, J. Kleinberg, *Networks, Crowds, and Markets*, Cambridge University Press, 2010 (Chapter 9: Auctions).
- W. Vickrey, "Counterspeculation, Auctions, and Competitive Sealed Tenders", *Journal of Finance*, 1961 · R. Myerson, "Optimal Auction Design", *Mathematics of Operations Research*, 1981.
- J. G. Riley, W. F. Samuelson, "Optimal Auctions", *AER* 71(3), 1981 · J. Bulow, P. Klemperer, "Auctions versus Negotiations", *AER* 86(1), 1996.
- M. H. Rothkopf, T. J. Teisberg, E. P. Kahn, "Why Are Vickrey Auctions Rare?", *J. Political Economy* 98(1), 1990 · L. M. Ausubel, P. Milgrom, "The Lovely but Lonely Vickrey Auction", *Combinatorial Auctions*, MIT Press, 2006.

References (2/2)

- R. P. McAfee, “A Dominant Strategy Double Auction”, *Journal of Economic Theory*, 1992 · E. Capen, R. Clapp, W. Campbell, “Competitive Bidding in High-Risk Situations”, *Journal of Petroleum Technology*, 1971.
- J. H. Kagel, R. M. Harstad, D. Levin, “Information Impact and Allocation Rules in Auctions with Affiliated Private Values: A Laboratory Study”, *Econometrica* 55(6), 1987 · R. B. Myerson, M. A. Satterthwaite, “Efficient Mechanisms for Bilateral Trading”, *Journal of Economic Theory* 29(2), 1983.
- Course pages: **N. Shah, CSC304 (Toronto)** · **M. Dinitz (Johns Hopkins)** · **M. Hoefer (Frankfurt)**.

The End

Chapter 5: Auctions

Previous: Chapter 4: Mechanism Design · Next: Chapter 6: Online Platforms