Chapter 13: Learning and Adaptation

Algorithmic Game Theory

Best-response dynamics · Fictitious play · Regret · Hedge · Correlated equilibrium · ε-greedy and UCB · Reinforcement learning · Replicator dynamics · Tit-for-Tat

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_13_Learning_and_Adaptation.html

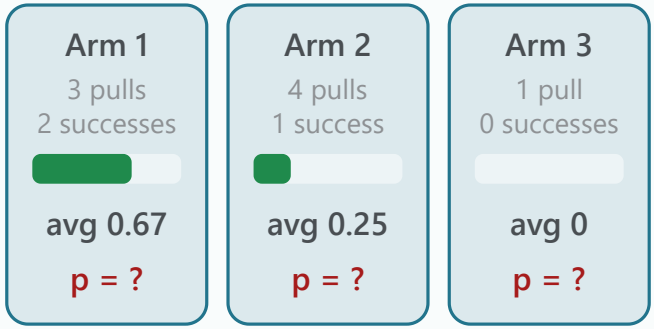
Warm-up: The Multi-Armed Bandit

The rules

Each arm pays 1 point with a hidden probability; pulling an arm shows only its own outcome, and the goal is to maximize total reward.

Think about it

- Would you stick with Arm 1, which has the best average so far, or try Arm 3 again?
- Out of a hundred pulls, how many would you spend on trying arms (exploration)?



True probabilities are hidden and redrawn in every new trial.
Arm 3 has been tried only once.

Learning Outcomes

Dynamics

Run best-response dynamics and fictitious play step by step; explain when they converge to equilibrium.

Regret

Compute external regret; explain why multiplicative weights (Hedge) has no regret.

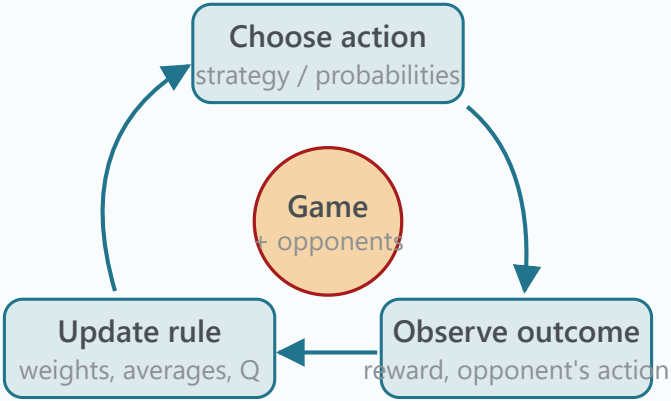
Exploration–exploitation

Trace and compare ε-greedy and UCB on the multi-armed bandit problem.

Evolution and repetition

Explain replicator dynamics and the success of Tit-for-Tat in the repeated Prisoner's Dilemma.

Why Learning and Adaptation?



Learning: improving a strategy from past experience. **Adaptation**: updating the strategy as opponents and the environment change.

- **Equilibria can be hard to compute**: Nash equilibrium in two-player games is PPAD-complete (so is approximate equilibrium with many players), a class with no known fast algorithm for large games. How do players "find" an equilibrium?
- **They may not know the game**: the payoff matrix is unknown; they only play and see the outcome.
- The question: do simple, local update rules eventually lead to **equilibrium**, or at least to **good performance**?

Examples: chess and poker AIs learn through self-play; bidding algorithms in ad auctions adapt daily; navigation apps reroute with traffic.

Learning Models: An Overview

Model	What does the player observe?	How does it update?
Best-response dynamics	The opponent's last action	Plays a best response to it
Fictitious play	The opponent's whole history	Best response to the frequencies
Regret minimization (Hedge)	The payoff of every action	Multiplicatively boosts good actions
Multi-armed bandit (ε-greedy, UCB)	Only the chosen action's reward	Mean estimate + exploration
Reinforcement learning (Q-learning)	Reward and new state	Updates the Q(s, a) values
Replicator dynamics	Average success in the population	Successful strategies spread

Top to bottom: the player stops watching the opponent and watches its own reward; in the last row the view shifts from the individual to the population.

Best-Response Dynamics (Step by Step)

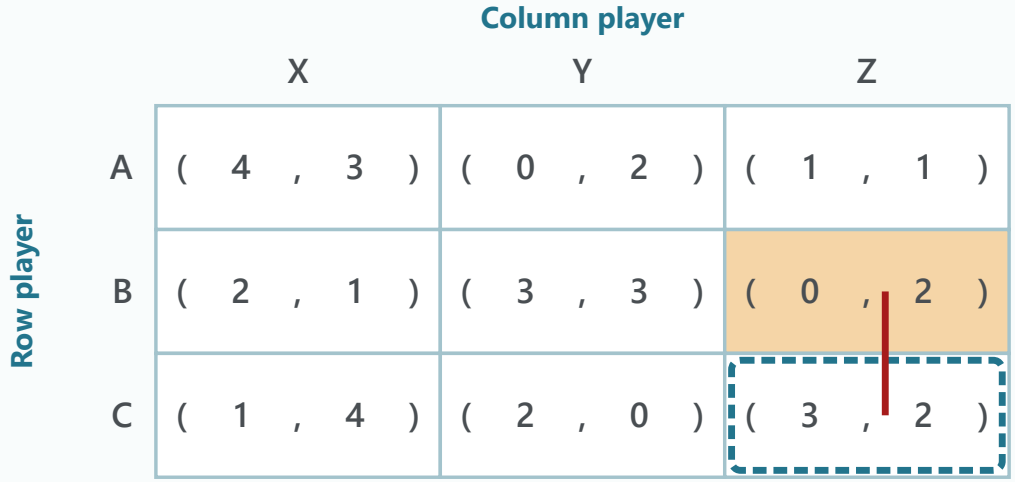
Row player \ Column player	X	Y	Z
A	(4, 3)	(0, 2)	(1, 1)
B	(2, 1)	(3, 3)	(0, 2)
C	(1, 4)	(2, 0)	(3, 2)

Starting profile **(B, Z)**. Players take turns best-responding to the opponent's *current* action.

Step 1 / 5

Starting from (C, Y) would lead to the equilibrium (B, Y): where you end up depends on where you start.

Best-Response Dynamics (Step by Step)

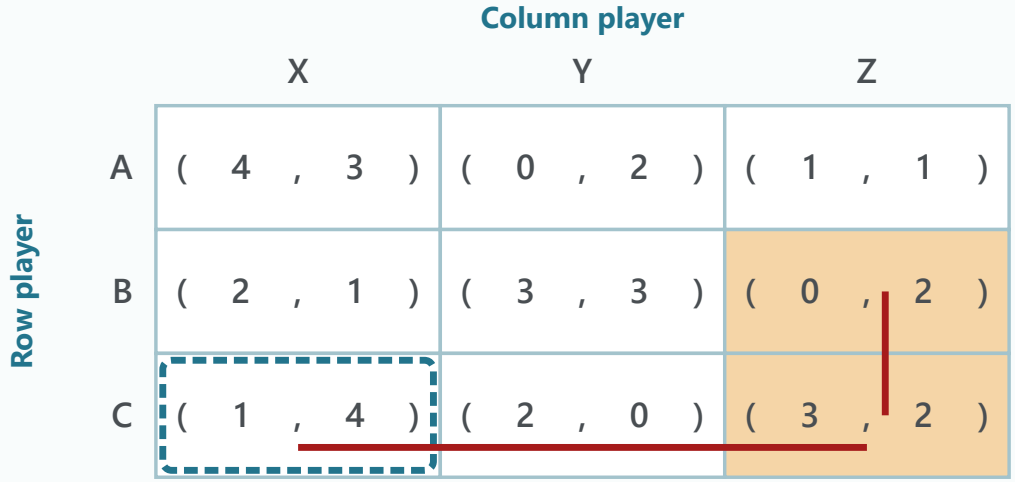


Row looks for a best response to Z: $u_1 = 1, 0, 3 \rightarrow \mathbf{C}$. Profile (C, Z).

Step 2 / 5

Starting from (C, Y) would lead to the equilibrium (B, Y): where you end up depends on where you start.

Best-Response Dynamics (Step by Step)

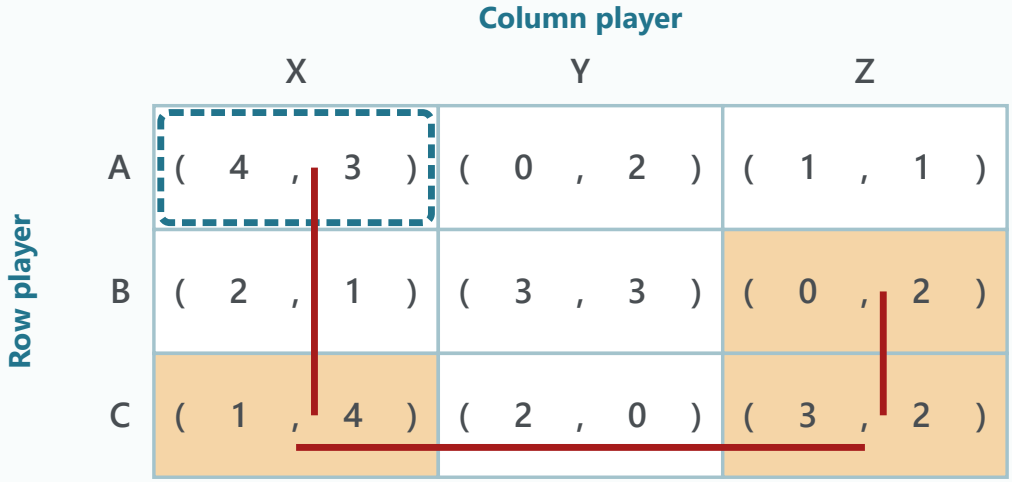


Column responds to C: $u_2 = 4, 0, 2 \rightarrow$ **X**. Profile (C, X).

Step 3 / 5

Starting from (C, Y) would lead to the equilibrium (B, Y): where you end up depends on where you start.

Best-Response Dynamics (Step by Step)

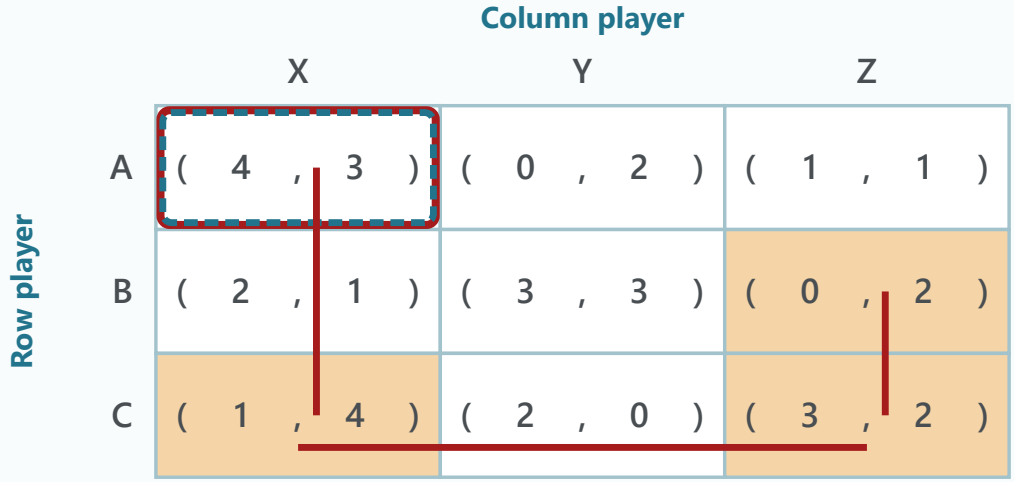


Row responds to X: $u_1 = 4, 2, 1 \rightarrow$ **A**. Profile (A, X).

Step 4 / 5

Starting from (C, Y) would lead to the equilibrium (B, Y): where you end up depends on where you start.

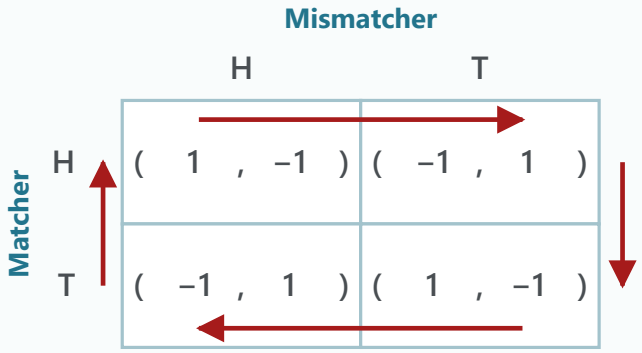
Best-Response Dynamics (Step by Step)



Column already picks X against A (3 > 2, 1), and Row does not leave A → the dynamics stop at the **Nash equilibrium (A, X)**.

Starting from (C, Y) would lead to the equilibrium (B, Y): where you end up depends on where you start.

When Does Best-Response Dynamics Converge?



Matching Pennies: (H,H) → (H,T) → (T,T) → (T,H) → (H,H) … an endless cycle

Potential games: converges

Every improving deviation increases a common potential function, which cannot increase forever in a finite game (Monderer–Shapley 1996; congestion games).

No pure equilibrium: cycles

In Matching Pennies every step makes one player happy and the other unhappy.

Remedy: remember the past

Respond to the whole history, not just the last action → fictitious play.

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2						
3						
4						
5						
6						
7						
8						
9						
10						

Round 1 is arbitrary: both play **H** (no history yet).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3						
4						
5						
6						
7						
8						
9						
10						

The matcher sees 1 / 0 in the opponent's history → **H** (match the majority, ties go to H). The mismatcher sees 1 / 0 → **T** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4						
5						
6						
7						
8						
9						
10						

The matcher sees 1 / 1 in the opponent's history → **H** (match the majority, ties go to H). The mismatcher sees 2 / 0 → **T** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5						
6						
7						
8						
9						
10						

The matcher sees 1 / 2 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 0 → **T** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6						
7						
8						
9						
10						

The matcher sees 1 / 3 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 1 → **T** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6	1 / 4	**T**	3 / 2	**T**	0.50	0.17
7						
8						
9						
10						

The matcher sees 1 / 4 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 2 → **T** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6	1 / 4	**T**	3 / 2	**T**	0.50	0.17
7	1 / 5	**T**	3 / 3	**H**	0.43	0.29
8						
9						
10						

The matcher sees 1 / 5 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 3 → **H** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6	1 / 4	**T**	3 / 2	**T**	0.50	0.17
7	1 / 5	**T**	3 / 3	**H**	0.43	0.29
8	2 / 5	**T**	3 / 4	**H**	0.38	0.38
9						
10						

The matcher sees 2 / 5 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 4 → **H** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6	1 / 4	**T**	3 / 2	**T**	0.50	0.17
7	1 / 5	**T**	3 / 3	**H**	0.43	0.29
8	2 / 5	**T**	3 / 4	**H**	0.38	0.38
9	3 / 5	**T**	3 / 5	**H**	0.33	0.44
10						

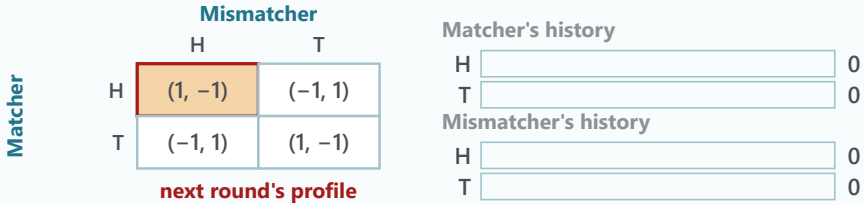
The matcher sees 3 / 5 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 5 → **H** (opposite of the majority, ties go to H).

Fictitious Play: Matching Pennies

t	Mism. history H/T (earlier rounds)	Matcher	Match. history H/T (earlier rounds)	Mismatcher	Match. H rate (incl. this round)	Mism. H rate (incl. this round)
1	0 / 0	**H**	0 / 0	**H**	1.00	1.00
2	1 / 0	**H**	1 / 0	**T**	1.00	0.50
3	1 / 1	**H**	2 / 0	**T**	1.00	0.33
4	1 / 2	**T**	3 / 0	**T**	0.75	0.25
5	1 / 3	**T**	3 / 1	**T**	0.60	0.20
6	1 / 4	**T**	3 / 2	**T**	0.50	0.17
7	1 / 5	**T**	3 / 3	**H**	0.43	0.29
8	2 / 5	**T**	3 / 4	**H**	0.38	0.38
9	3 / 5	**T**	3 / 5	**H**	0.33	0.44
10	4 / 5	**T**	3 / 6	**H**	0.30	0.50

The matcher sees 4 / 5 in the opponent's history → **T** (match the majority, ties go to H). The mismatcher sees 3 / 6 → **H** (opposite of the majority, ties go to H).

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Call `fictitious(6)`. Counters are zero; with no history, both play **H** in round 1.

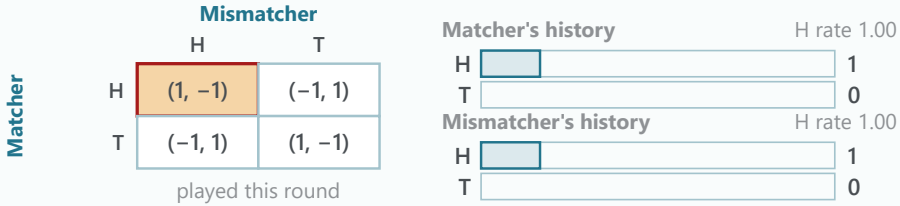
Step 1 / 13

```
1  def fictitious(T):
▶      nm, nx = [0, 0], [0, 0]   # H/T counters
▶      a, b = 0, 0               # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1            # matcher's history
6          nx[b] += 1            # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`T = 6` `nm = [0, 0]` `nx = [0, 0]` `a = 0 (H)` `b = 0 (H)`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 1: (H, H) is played and the matcher wins. Counters: matcher H/T = 1/0, mismatcher H/T = 1/0.

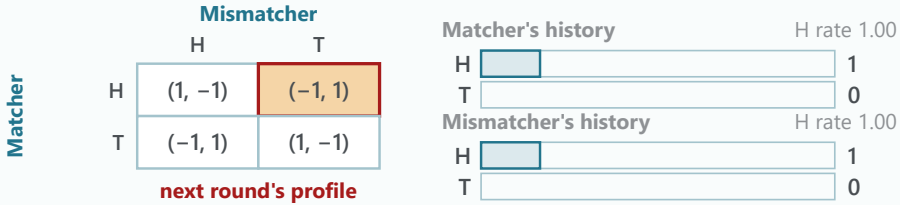
Step 2 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 1` `a = 0 (H)` `b = 0 (H)` `nm = [1, 0]` `nx = [1, 0]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Next round: the mismatcher mostly played H → matcher **H**; the matcher mostly played H → mismatcher **T**.

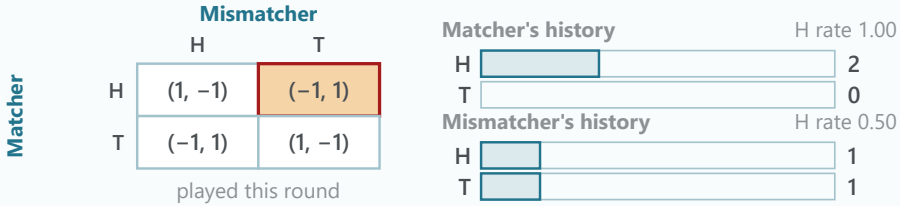
Step 3 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]   # H/T counters
3      a, b = 0, 0               # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1            # matcher's history
6          nx[b] += 1            # mismatcher's history
7          # matcher: match the mismatcher's majority
▶          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
▶          b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 1` `a = 0 (H)` `b = 1 (T)` `nm = [1, 0]` `nx = [1, 0]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 2: (H, T) is played and the mismatcher wins. Counters: matcher H/T = 2/0, mismatcher H/T = 1/1.

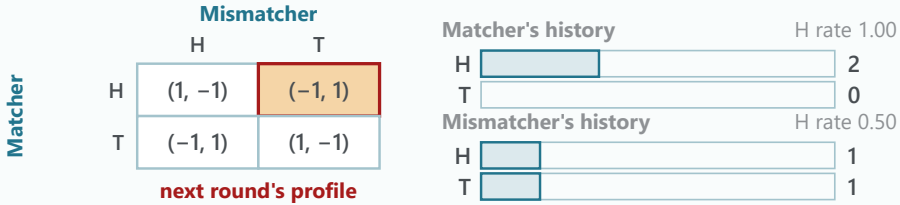
Step 4 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 2` `a = 0 (H)` `b = 1 (T)` `nm = [2, 0]` `nx = [1, 1]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Next round: the mismatcher's counters are tied (ties go to H) → matcher **H**; the matcher mostly played H → mismatcher **T**.

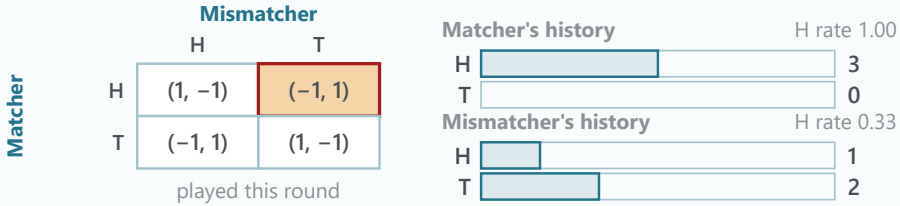
Step 5 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]   # H/T counters
3      a, b = 0, 0               # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1            # matcher's history
6          nx[b] += 1            # mismatcher's history
7          # matcher: match the mismatcher's majority
▶          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
▶          b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 2` `a = 0 (H)` `b = 1 (T)` `nm = [2, 0]` `nx = [1, 1]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 3: (H, T) is played and the mismatcher wins. Counters: matcher H/T = 3/0, mismatcher H/T = 1/2.

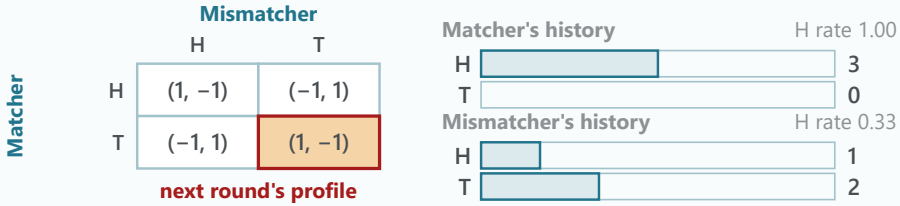
Step 6 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 3` `a = 0 (H)` `b = 1 (T)` `nm = [3, 0]` `nx = [1, 2]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Next round: the mismatcher mostly played T → matcher **T**; the matcher mostly played H → mismatcher **T**.

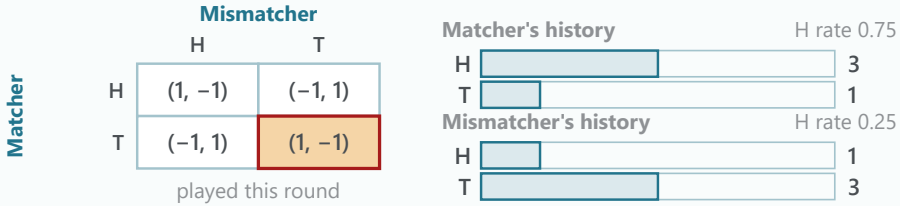
Step 7 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1             # matcher's history
6          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
▶          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
▶          b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 3` `a = 1 (T)` `b = 1 (T)` `nm = [3, 0]` `nx = [1, 2]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 4: (T, T) is played and the matcher wins. Counters: matcher H/T = 3/1, mismatcher H/T = 1/3.

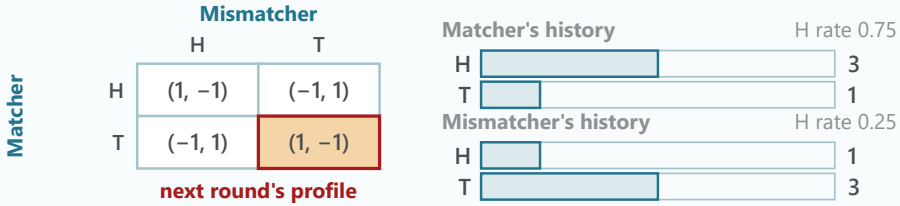
Step 8 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

t = 4 | a = 1 (T) | b = 1 (T) | nm = [3, 1] | nx = [1, 3]

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Next round: the mismatcher mostly played T → matcher **T**; the matcher mostly played H → mismatcher **T**.

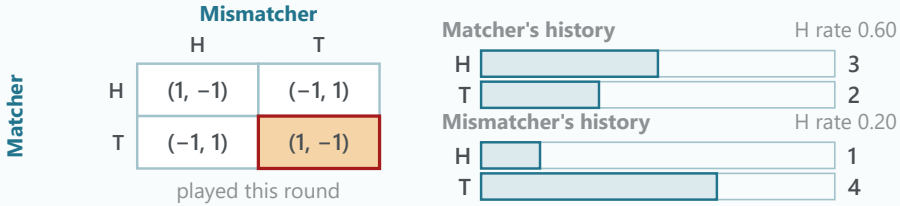
Step 9 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1             # matcher's history
6          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
▶          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
▶          b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 4` `a = 1 (T)` `b = 1 (T)` `nm = [3, 1]` `nx = [1, 3]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 5: (T, T) is played and the matcher wins. Counters: matcher H/T = 3/2, mismatcher H/T = 1/4.

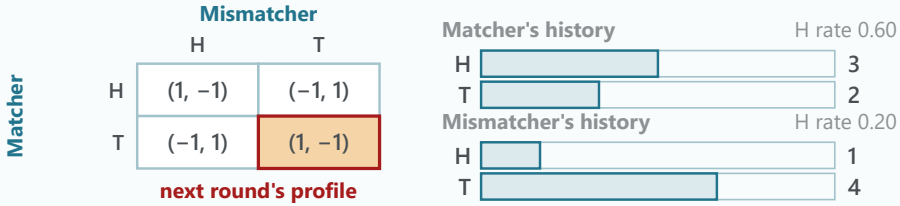
Step 10 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 5` `a = 1 (T)` `b = 1 (T)` `nm = [3, 2]` `nx = [1, 4]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Next round: the mismatcher mostly played T → matcher **T**; the matcher mostly played H → mismatcher **T**.

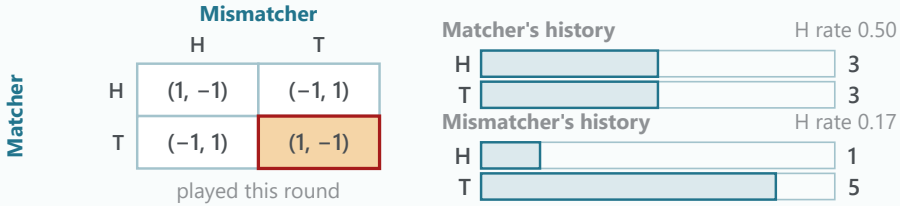
Step 11 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]   # H/T counters
3      a, b = 0, 0               # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1            # matcher's history
6          nx[b] += 1            # mismatcher's history
7          # matcher: match the mismatcher's majority
▶          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
▶          b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 5` `a = 1 (T)` `b = 1 (T)` `nm = [3, 2]` `nx = [1, 4]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



Round 6: (T, T) is played and the matcher wins. Counters: matcher H/T = 3/3, mismatcher H/T = 1/5.

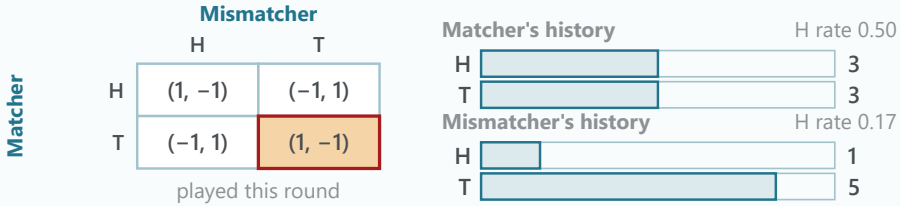
Step 12 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]    # H/T counters
3      a, b = 0, 0                # round 1: both H
▶      for t in range(1, T + 1):
▶          nm[a] += 1             # matcher's history
▶          nx[b] += 1             # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
11     return nm, nx
```

VARIABLES

`t = 6` `a = 1 (T)` `b = 1 (T)` `nm = [3, 3]` `nx = [1, 5]`

Code Trace: Fictitious Play (Matching Pennies, 6 rounds)



return nm = [3, 3], nx = [1, 5]. The actions cycle (HH → HT → TT → TH …); as the cycles lengthen, the frequencies approach ½ (0.50 and 0.17 after 6 rounds; 0.472 and 0.492 after 400).

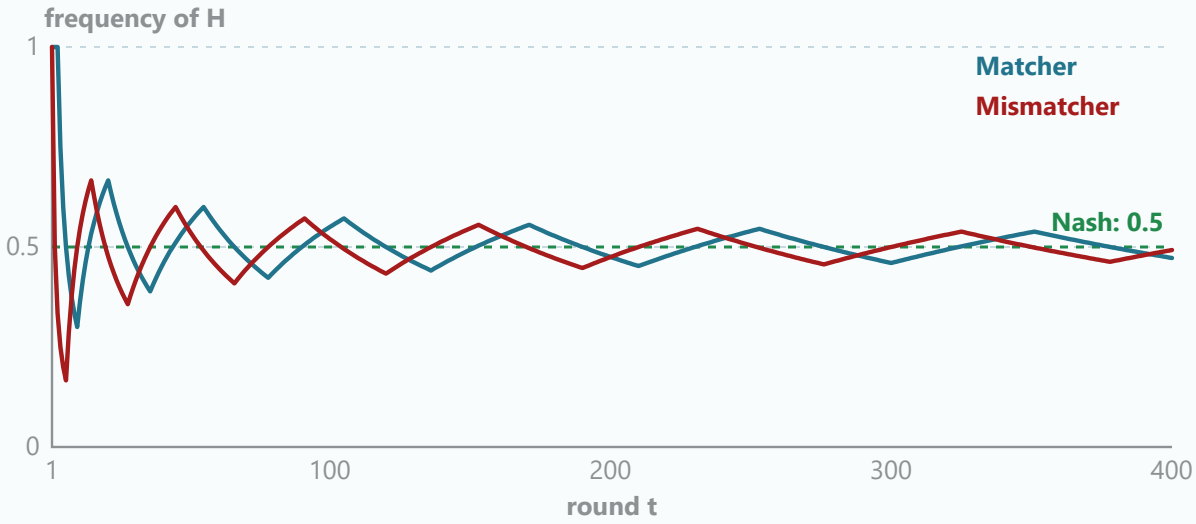
Step 13 / 13

```
1  def fictitious(T):
2      nm, nx = [0, 0], [0, 0]   # H/T counters
3      a, b = 0, 0               # round 1: both H
4      for t in range(1, T + 1):
5          nm[a] += 1            # matcher's history
6          nx[b] += 1            # mismatcher's history
7          # matcher: match the mismatcher's majority
8          a = 0 if nx[0] >= nx[1] else 1
9          # mismatcher: avoid the matcher's majority
10         b = 1 if nm[0] > nm[1] else 0
▶      return nm, nx
```

VARIABLES

`nm = [3, 3]` `nx = [1, 5]`

Fictitious Play: Frequencies Converge to Equilibrium



- Actions change in ever-longer cycles, but the **frequencies** approach the mixed equilibrium (½, ½): 0.472 and 0.492 after 400 rounds.
- Guaranteed convergence: two-player zero-sum games (Robinson 1951), 2×2 games, potential games. Shapley (1964): in a 3×3 game the frequencies cycle and never converge.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2					
3					
4					
5					
6					

Leader A (totals 0.0 / 0.0) → **A** is chosen; round payoff 0.5. Algorithm's total: 0.5.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2	0.5 / 0.0	0.0	1.0	**A**	0.0
3					
4					
5					
6					

Leader A (totals 0.5 / 0.0) → **A** is chosen; round payoff 0.0. Algorithm's total: 0.5.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2	0.5 / 0.0	0.0	1.0	**A**	0.0
3	0.5 / 1.0	1.0	0.0	**B**	0.0
4					
5					
6					

Leader B (totals 0.5 / 1.0) → **B** is chosen; round payoff 0.0. Algorithm's total: 0.5.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2	0.5 / 0.0	0.0	1.0	**A**	0.0
3	0.5 / 1.0	1.0	0.0	**B**	0.0
4	1.5 / 1.0	0.0	1.0	**A**	0.0
5					
6					

Leader A (totals 1.5 / 1.0) → **A** is chosen; round payoff 0.0. Algorithm's total: 0.5.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2	0.5 / 0.0	0.0	1.0	**A**	0.0
3	0.5 / 1.0	1.0	0.0	**B**	0.0
4	1.5 / 1.0	0.0	1.0	**A**	0.0
5	1.5 / 2.0	1.0	0.0	**B**	0.0
6					

Leader B (totals 1.5 / 2.0) → **B** is chosen; round payoff 0.0. Algorithm's total: 0.5.

External Regret: Why Does "Follow the Leader" Fail?

Regret = (the total you would have earned by always playing the best single action in hindsight) − (the total you actually earned). **External regret** $R_T = \max_i \Sigma_t\, g_t(i) - \Sigma_t\, g_t(a_t)$ (using expected payoff for randomized choices): the gap to the best *fixed* action in hindsight. If $R_T/T \to 0$, the algorithm has **no regret**.

t	Totals so far A / B	g(A)	g(B)	Follow the leader	Payoff
1	0.0 / 0.0	0.5	0.0	**A**	0.5
2	0.5 / 0.0	0.0	1.0	**A**	0.0
3	0.5 / 1.0	1.0	0.0	**B**	0.0
4	1.5 / 1.0	0.0	1.0	**A**	0.0
5	1.5 / 2.0	1.0	0.0	**B**	0.0
6	2.5 / 2.0	0.0	1.0	**A**	0.0

Leader A (totals 2.5 / 2.0) → **A** is chosen; round payoff 0.0. Algorithm's total: 0.5. Best fixed action B: 3. **Regret = 3 − 0.5 = 2.5 ≈ T/2**.

Multiplicative Weights / Hedge

1. Give each action a weight: $w_i = 1$.
2. Each round, play with probabilities $p_i = w_i / \Sigma_j w_j$.
3. Observe every action's payoff and update:

$$w_i \leftarrow w_i \cdot e^{\eta \cdot g_t(i)}$$

An action with payoff 1 has its weight multiplied by e^{η} (exactly doubled if $\eta = \ln 2$).

Theorem. With payoffs in [0, 1] and n actions, $R_T \le (\ln n)/\eta + \eta T$ ($\ln n$ is the natural log of n; ≈ 1.1 for $n = 3$). Choosing $\eta = \sqrt{(\ln n / T)}$ gives $R_T \le 2\sqrt{(T \ln n)}$, so $\mathbf{R_T/T \to 0}$.

Randomness is essential

Follow the leader is deterministic, so an adversary exploits it; Hedge plays mixed.

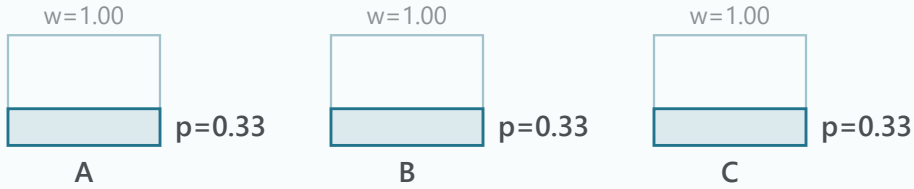
η: the learning rate

A large η reacts fast but is sensitive to noise; a small η is slow but stable.

In games

If everyone plays a no-regret algorithm, the time average approaches a *coarse correlated equilibrium*, and minimax in zero-sum games.

Code Trace: Multiplicative Weights



Three experts (actions) A, B, C; η = 0.5. Initial weights are equal: **w = [1, 1, 1]**.

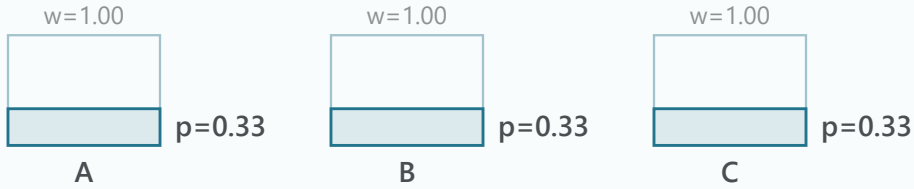
Step 1 / 17

```
1  import math
2  def hedge(G, eta):
▶      w = [1.0] * len(G[0])
▶      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`eta = 0.5` `w = [1.00, 1.00, 1.00]` `total = 0.00`

Code Trace: Multiplicative Weights



Round 1: W = 3.00, probabilities **p = [0.33, 0.33, 0.33]**.

Step 2 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
▶          W = sum(w)
▶          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

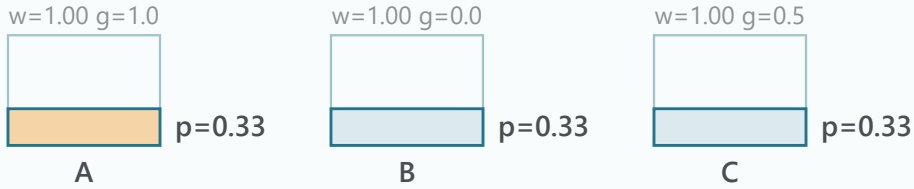
`t = 1` `W = 3.00`

`p = [0.33, 0.33, 0.33]`

`w = [1.00, 1.00, 1.00]`

`total = 0.00`

Code Trace: Multiplicative Weights



Payoffs revealed: g = [1, 0, 0.5]. Expected payoff Σ p·g = 0.50 → **total = 0.50**.

Step 3 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
▶          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

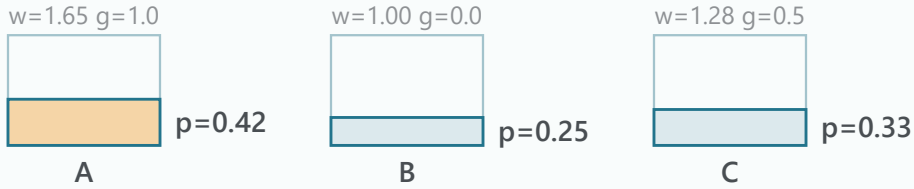
VARIABLES

`t = 1` `g = [1, 0, 0.5]`

`p = [0.33, 0.33, 0.33]`

`total = 0.50`

Code Trace: Multiplicative Weights



Update $w \leftarrow w \cdot e^{0.5 \cdot g}$: **w = [1.65, 1.00, 1.28]**. Experts that did well this round gain weight.

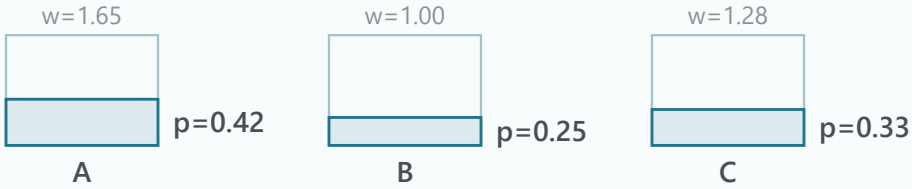
Step 4 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
▶          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`t = 1` `g = [1, 0, 0.5]` `w = [1.65, 1.00, 1.28]` `total = 0.50`

Code Trace: Multiplicative Weights



Round 2: W = 3.93, probabilities **p = [0.42, 0.25, 0.33]**.

Step 5 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
▶          W = sum(w)
▶          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

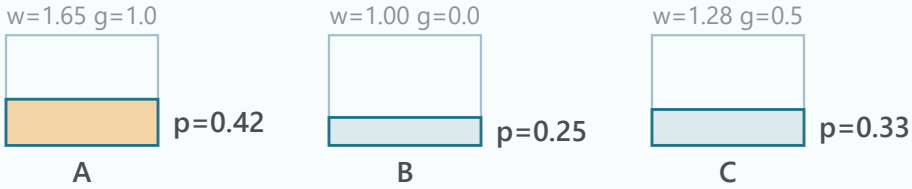
`t = 2` `W = 3.93`

`p = [0.42, 0.25, 0.33]`

`w = [1.65, 1.00, 1.28]`

`total = 0.50`

Code Trace: Multiplicative Weights



Payoffs revealed: g = [1, 0, 0.5]. Expected payoff Σ p·g = 0.58 → **total = 1.08**.


```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
▶          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

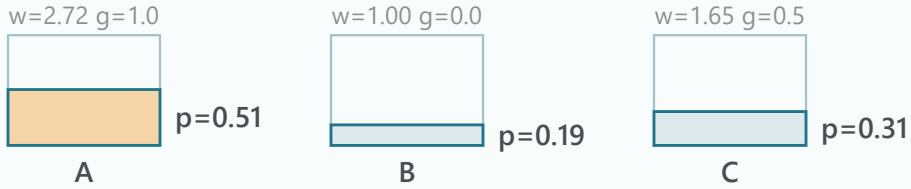
VARIABLES

`t = 2` `g = [1, 0, 0.5]`

`p = [0.42, 0.25, 0.33]`

`total = 1.08`

Code Trace: Multiplicative Weights



Update $w \leftarrow w \cdot e^{0.5 \cdot g}$: **w = [2.72, 1.00, 1.65]**. Experts that did well this round gain weight.

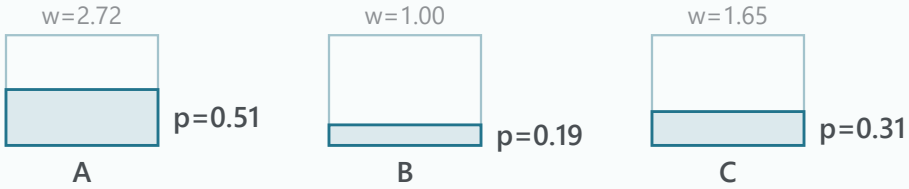
Step 7 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
▶          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`t = 2` `g = [1, 0, 0.5]` `w = [2.72, 1.00, 1.65]` `total = 1.08`

Code Trace: Multiplicative Weights



Round 3: W = 5.37, probabilities **p = [0.51, 0.19, 0.31]**.

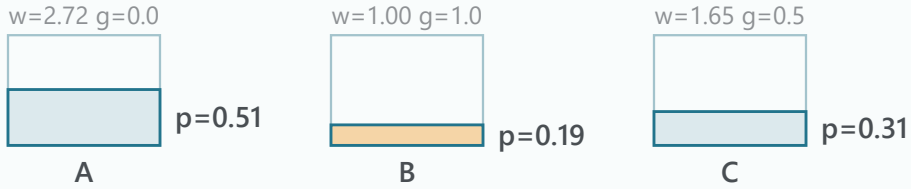
Step 8 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
▶          W = sum(w)
▶          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

t = 3 | W = 5.37 | p = [0.51, 0.19, 0.31] | w = [2.72, 1.00, 1.65] | total = 1.08

Code Trace: Multiplicative Weights



Payoffs revealed: g = [0, 1, 0.5]. Expected payoff Σ p·g = 0.34 → **total = 1.42**.

Step 9 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
▶          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

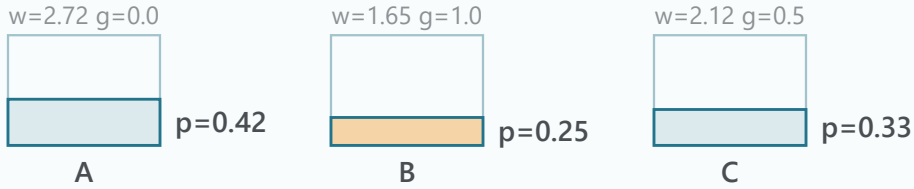
VARIABLES

`t = 3` `g = [0, 1, 0.5]`

`p = [0.51, 0.19, 0.31]`

`total = 1.42`

Code Trace: Multiplicative Weights



Update $w \leftarrow w \cdot e^{0.5 \cdot g}$: **w = [2.72, 1.65, 2.12]**. Experts that did well this round gain weight.

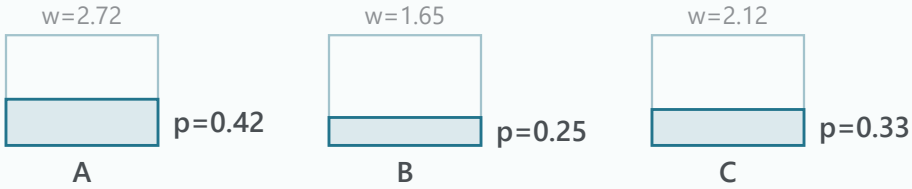
Step 10 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
▶          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`t = 3` `g = [0, 1, 0.5]` `w = [2.72, 1.65, 2.12]` `total = 1.42`

Code Trace: Multiplicative Weights



Round 4: W = 6.48, probabilities **p = [0.42, 0.25, 0.33]**.

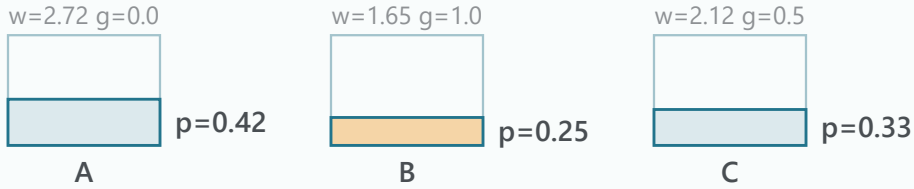
Step 11 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
▶          W = sum(w)
▶          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

t = 4 | W = 6.48 | p = [0.42, 0.25, 0.33] | w = [2.72, 1.65, 2.12] | total = 1.42

Code Trace: Multiplicative Weights



Payoffs revealed: g = [0, 1, 0.5]. Expected payoff Σ p·g = 0.42 → **total = 1.84**.

Step 12 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
▶          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

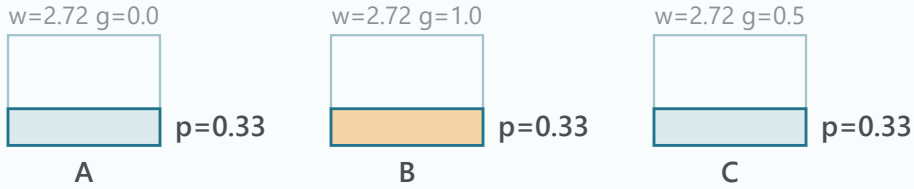
VARIABLES

`t = 4` `g = [0, 1, 0.5]`

`p = [0.42, 0.25, 0.33]`

`total = 1.84`

Code Trace: Multiplicative Weights



Update $w \leftarrow w \cdot e^{0.5 \cdot g}$: **w = [2.72, 2.72, 2.72]**. Experts that did well this round gain weight.

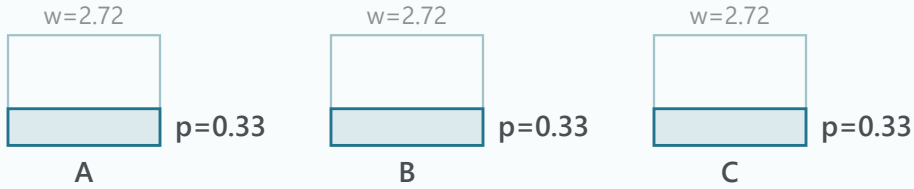
Step 13 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
▶          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`t = 4` `g = [0, 1, 0.5]` `w = [2.72, 2.72, 2.72]` `total = 1.84`

Code Trace: Multiplicative Weights



Round 5: W = 8.15, probabilities **p = [0.33, 0.33, 0.33]**.

Step 14 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
▶          W = sum(w)
▶          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

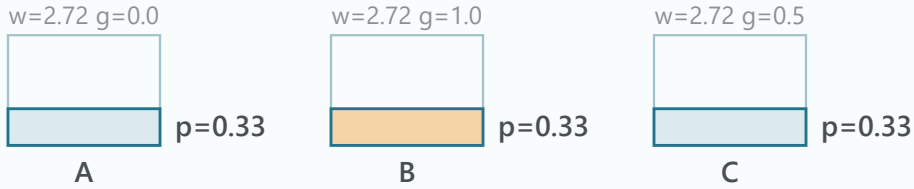
`t = 5` `W = 8.15`

`p = [0.33, 0.33, 0.33]`

`w = [2.72, 2.72, 2.72]`

`total = 1.84`

Code Trace: Multiplicative Weights



Payoffs revealed: g = [0, 1, 0.5]. Expected payoff Σ p·g = 0.50 → **total = 2.34**.

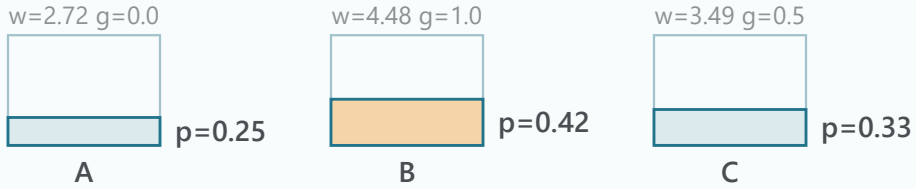
Step 15 / 17

```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
▶          total += sum(pi * gi for pi, gi in zip(p, g))
9          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

t = 5 | g = [0, 1, 0.5] | p = [0.33, 0.33, 0.33] | total = 2.34

Code Trace: Multiplicative Weights



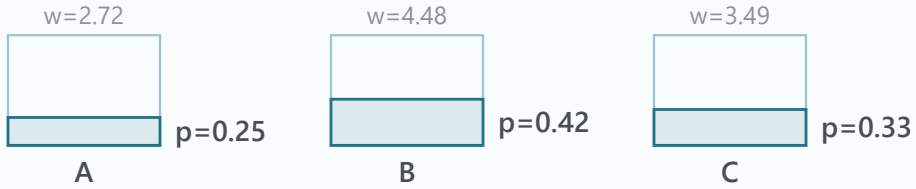
Update $w \leftarrow w \cdot e^{0.5 \cdot g}$: **w = [2.72, 4.48, 3.49]**. Experts that did well this round gain weight.


```
1  import math
2  def hedge(G, eta):
3      w = [1.0] * len(G[0])
4      total = 0.0
5      for g in G:                          # each round
6          W = sum(w)
7          p = [x / W for x in w]
8          total += sum(pi * gi for pi, gi in zip(p, g))
▶          w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
10     return total
```

VARIABLES

`t = 5` `g = [0, 1, 0.5]` `w = [2.72, 4.48, 3.49]` `total = 2.34`

Code Trace: Multiplicative Weights



return 2.34. Best fixed expert B: 3. Regret = 3 − 2.34 = **0.66** (follow the leader can do worse in 5 rounds).

Step 17 / 17

```
1 import math
2 def hedge(G, eta):
3     w = [1.0] * len(G[0])
4     total = 0.0
5     for g in G:                          # each round
6         W = sum(w)
7         p = [x / W for x in w]
8         total += sum(pi * gi for pi, gi in zip(p, g))
9         w = [x*math.exp(eta*gi) for x, gi in zip(w, g)]
▶     return total
```

VARIABLES

`w = [2.72, 4.48, 3.49]`

`total = 2.34`

Change the Assumption: What If the Opponent Learns Too?

Rock–Paper–Scissors: a win pays 1, a tie ½, a loss 0. Player 1 learns with Hedge (fixed $\eta = 0.1$; results are simulated). If the opponent always plays (Rock 0.5, Paper 0.3, Scissors 0.2), where does Hedge go? **What changes if the opponent also learns with Hedge?**

Change the Assumption: What If the Opponent Learns Too?

Rock–Paper–Scissors: a win pays 1, a tie ½, a loss 0. Player 1 learns with Hedge (fixed $\eta = 0.1$; results are simulated). If the opponent always plays (Rock 0.5, Paper 0.3, Scissors 0.2), where does Hedge go? **What changes if the opponent also learns with Hedge?**

Fixed opponent: Hedge goes to the best response. Expected payoffs: Rock 0.45, **Paper 0.65**, Scissors 0.40. Hedge's probability of Paper is 0.82 at round 100 and 0.997 at round 300.

Change the Assumption: What If the Opponent Learns Too?

Rock–Paper–Scissors: a win pays 1, a tie ½, a loss 0. Player 1 learns with Hedge (fixed $\eta = 0.1$; results are simulated). If the opponent always plays (Rock 0.5, Paper 0.3, Scissors 0.2), where does Hedge go? **What changes if the opponent also learns with Hedge?**

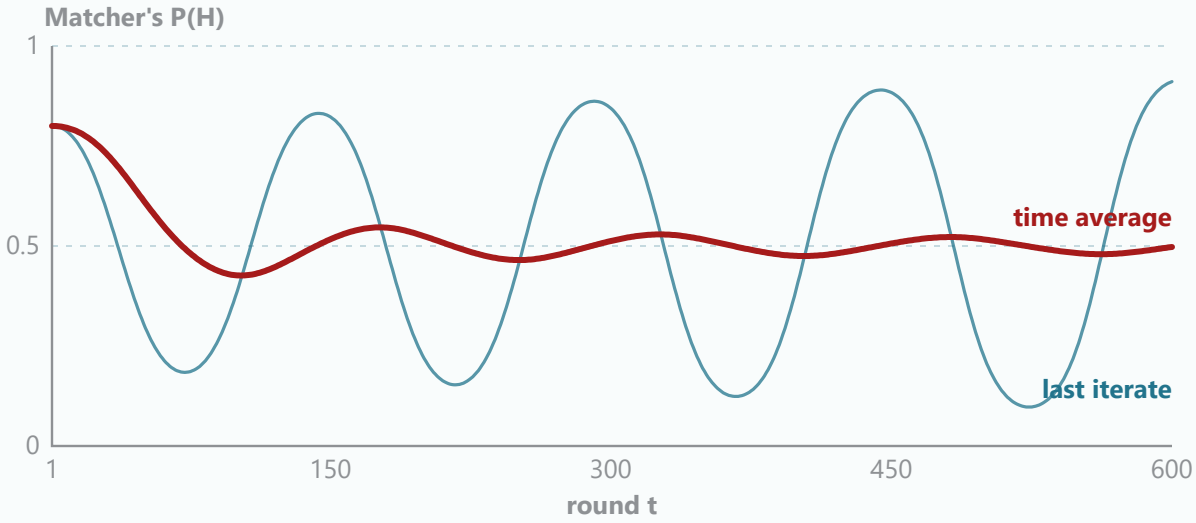
Fixed opponent: Hedge goes to the best response. Expected payoffs: Rock 0.45, **Paper 0.65**, Scissors 0.40. Hedge's probability of Paper is 0.82 at round 100 and 0.997 at round 300.

Learning opponent: the target moves. Both update at once; player 1 starts at (0.5, 0.3, 0.2), player 2 uniform.

round	100	200	800	1000
P(Rock) in that round	0.19	0.46	0.14	0.19
average P(Rock)	0.35	0.31	0.33	0.34

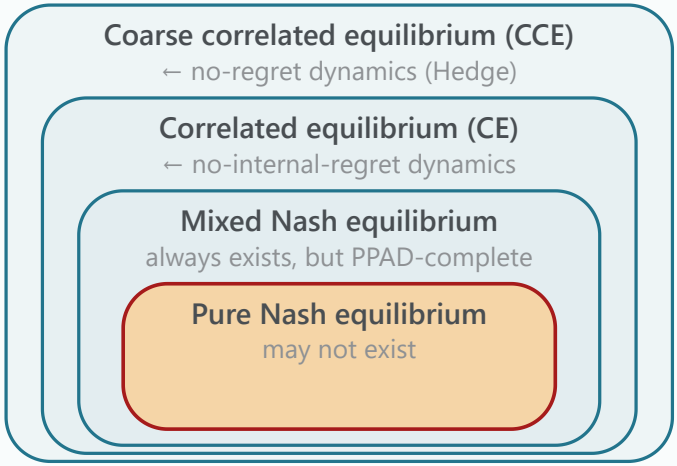
The last-round strategy keeps cycling; the time average approaches the equilibrium (⅓, ⅓, ⅓).

Hedge vs Itself: Matching Pennies



- Both players use Hedge ($\eta = 0.1$). Initial weights: matcher (H, T) = (0.8, 0.2), so P(H) = 0.8; mismatcher (0.5, 0.5). A win pays 1, a loss 0; each round both players update simultaneously with the expected payoffs of their actions.
- The probability in each round (the **last iterate**) never settles at equilibrium; it swings widely around it. The **time average** approaches the minimax strategy (½, ½) (0.498 after 600 rounds). This is the basis of the learning proof of the minimax theorem.

Which Equilibrium Does No-Regret Learning Reach?



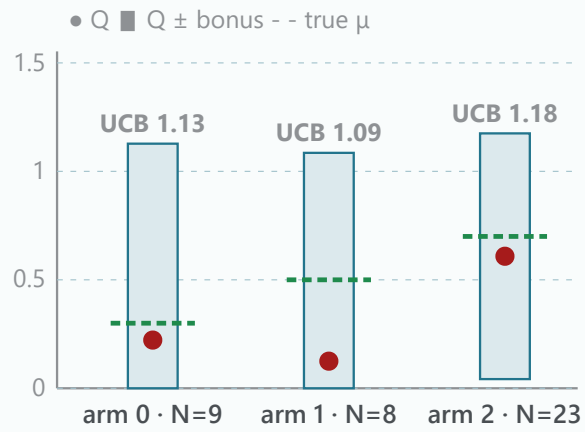
- A **mediator** tells everyone what to play (like a traffic light).
- **Correlated equilibrium** (CE): following your recommendation is best even *after seeing it*. With algorithms that minimize a stronger kind of regret (internal regret), the time average approaches the set of CE.
- **Coarse correlated equilibrium** (CCE) is weaker: committing to "I'll always play this" before seeing the recommendation does not pay. If everyone plays no-regret, the *time average* approaches a CCE.
- CE and CCE are easy to compute; mixed Nash equilibrium is PPAD-complete.

A traffic light at a crossing implements a **correlated equilibrium** of the game of Chicken: no one gains by ignoring the light. No-regret learning finds such equilibria without a central planner.

The Multi-Armed Bandit

- A K-armed slot machine: arm i pays a random reward with (unknown) mean μ_i.
- Each round **one** arm is pulled and only its reward is seen (unlike Hedge: partial feedback).
- Regret: $R_T = T \cdot \mu^* - \Sigma_t \mu_{a_t}$, where μ^* is the best arm's mean.

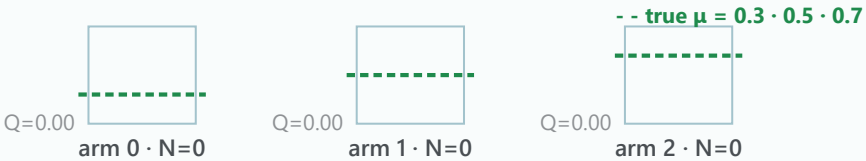
The exploration–exploitation dilemma: exploration gathers information about new arms; exploitation uses the arm that looks best so far. Pure exploitation gets stuck on the wrong arm; pure exploration wastes reward.



UCB1, μ = (0.3, 0.5, 0.7), 40 rounds: a rarely tried arm has a wide interval, so it gets tried again.

Applications: ad and news recommendation, A/B testing, clinical trials, price experimentation.

Code Trace: ε-Greedy (ε = 0.2)



Each round ε-greedy pulls a random arm with probability ε (explore), otherwise the arm with the largest Q (exploit). Call: `egreedy([0.3, 0.5, 0.7], 8, 0.2, random.Random(11))`. Estimates Q = 0, counts N = 0. The code numbers arms from 0: arms 0, 1, 2 = Arms 1, 2, 3.

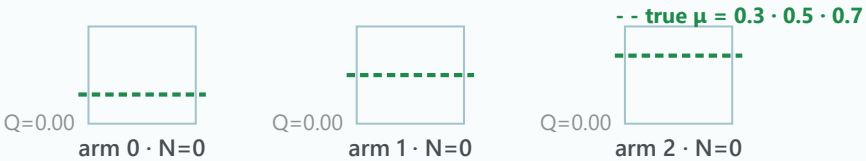
Step 1 / 18

```
1  def egreedy(mu, T, eps, rng):
▶      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:          # explore
5              a = rng.randrange(len(mu))
6          else:                           # exploit
7              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

VARIABLES

`eps = 0.2` `N = [0, 0, 0]` `Q = [0.00, 0.00, 0.00]`

Code Trace: ε-Greedy (ε = 0.2)



t = 1: 0.452 ≥ 0.2 → **exploit**: the arm with the largest Q is **0** (ties go to the first index).

Step 2 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:           # explore
5              a = rng.randrange(len(mu))
6          else:                            # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

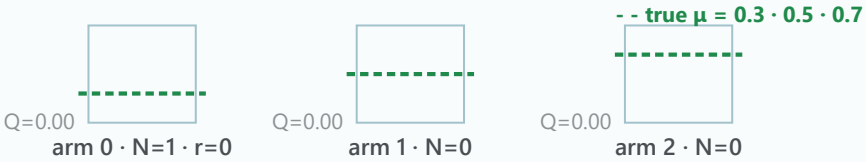
VARIABLES

t = 1 | u = 0.452 | a = 0

N = [0, 0, 0]

Q = [0.00, 0.00, 0.00]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 0**; N[0] = 1, Q[0] = 0.00 (incremental mean).

Step 3 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

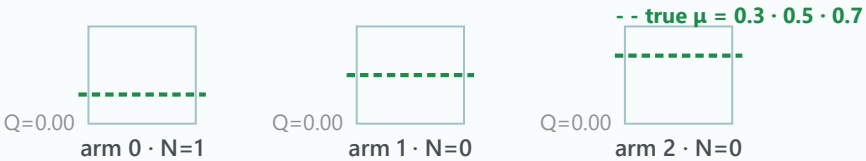
VARIABLES

`t = 1` `a = 0` `r = 0`

`N = [1, 0, 0]`

`Q = [0.00, 0.00, 0.00]`

Code Trace: ε-Greedy (ε = 0.2)



t = 2: 0.924 ≥ 0.2 → **exploit**: the arm with the largest Q is **0** (ties go to the first index).

Step 4 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

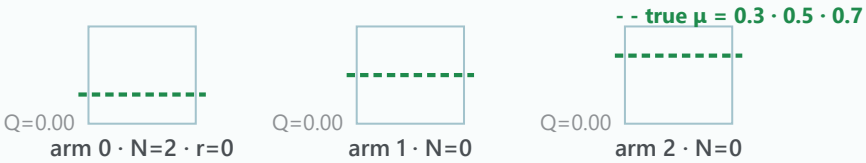
VARIABLES

`t = 2` `u = 0.924` `a = 0`

`N = [1, 0, 0]`

`Q = [0.00, 0.00, 0.00]`

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 0**; N[0] = 2, Q[0] = 0.00 (incremental mean).

Step 5 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

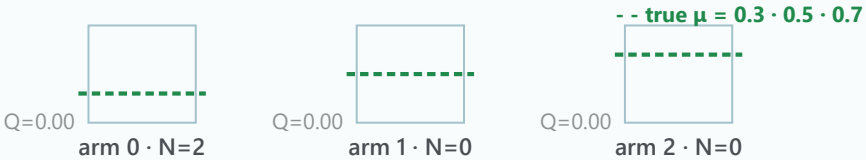
VARIABLES

`t = 2` `a = 0` `r = 0`

`N = [2, 0, 0]`

`Q = [0.00, 0.00, 0.00]`

Code Trace: ε-Greedy (ε = 0.2)



t = 3: 0.508 ≥ 0.2 → **exploit**: the arm with the largest Q is **0** (ties go to the first index).

Step 6 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

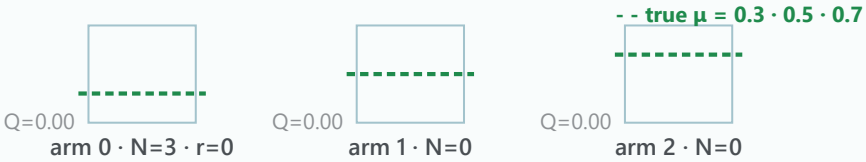
VARIABLES

t = 3 | u = 0.508 | a = 0

N = [2, 0, 0]

Q = [0.00, 0.00, 0.00]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 0**; N[0] = 3, Q[0] = 0.00 (incremental mean).

Step 7 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

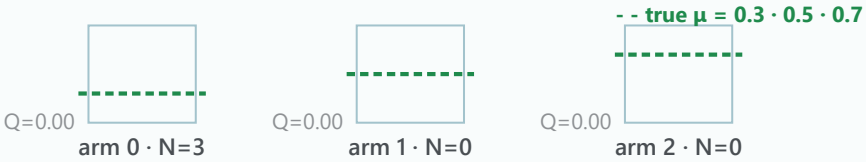
VARIABLES

t = 3 · a = 0 · r = 0

N = [3, 0, 0]

Q = [0.00, 0.00, 0.00]

Code Trace: ε-Greedy (ε = 0.2)



t = 4: random number 0.185 < 0.2 → **explore**, random arm **2**.

Step 8 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
▶              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

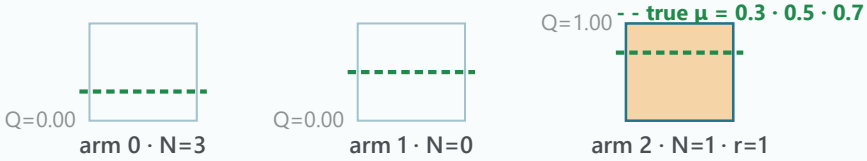
VARIABLES

`t = 4` `u = 0.185` `a = 2`

`N = [3, 0, 0]`

`Q = [0.00, 0.00, 0.00]`

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 1**; N[2] = 1, Q[2] = 1.00 (incremental mean).

Step 9 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

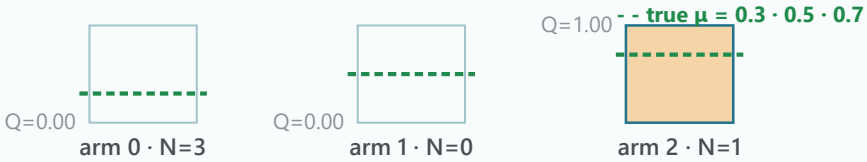
VARIABLES

t = 4 · a = 2 · r = 1

N = [3, 0, 1]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



t = 5: 0.614 ≥ 0.2 → **exploit**: the arm with the largest Q is **2**.


```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

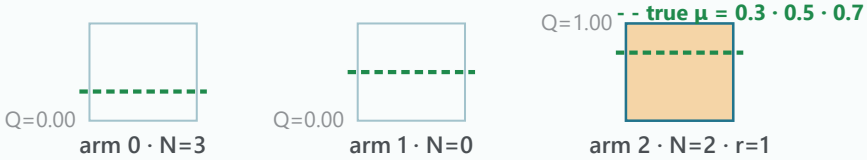
VARIABLES

t = 5 | u = 0.614 | a = 2

N = [3, 0, 1]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 1**; N[2] = 2, Q[2] = 1.00 (incremental mean).

Step 11 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

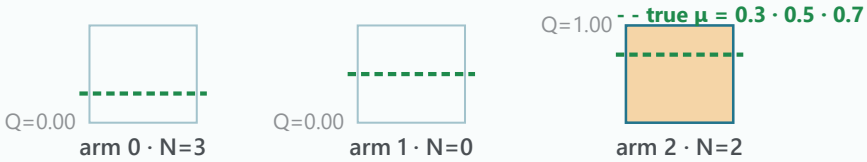
VARIABLES

t = 5 · a = 2 · r = 1

N = [3, 0, 2]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



t = 6: 0.447 ≥ 0.2 → **exploit**: the arm with the largest Q is **2**.


```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

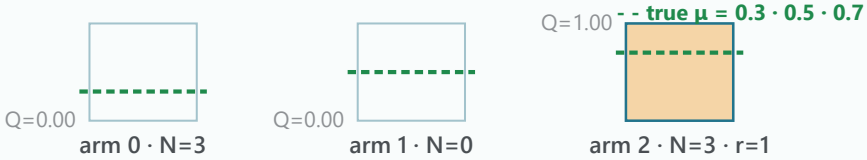
VARIABLES

t = 6 | u = 0.447 | a = 2

N = [3, 0, 2]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 1**; N[2] = 3, Q[2] = 1.00 (incremental mean).

Step 13 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

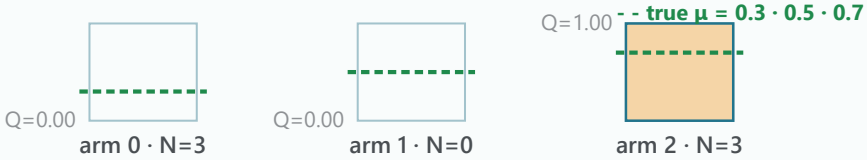
VARIABLES

t = 6 · a = 2 · r = 1

N = [3, 0, 3]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



t = 7: 0.539 ≥ 0.2 → **exploit**: the arm with the largest Q is **2**.


```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:          # explore
5              a = rng.randrange(len(mu))
6          else:                           # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

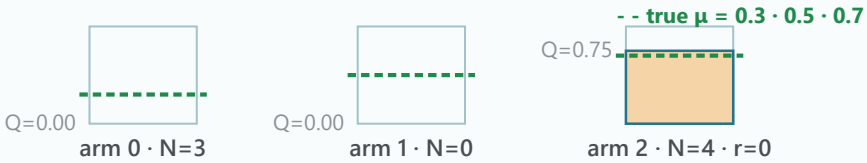
VARIABLES

t = 7 · u = 0.539 · a = 2

N = [3, 0, 3]

Q = [0.00, 0.00, 1.00]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 0**; N[2] = 4, Q[2] = 0.75 (incremental mean).

Step 15 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

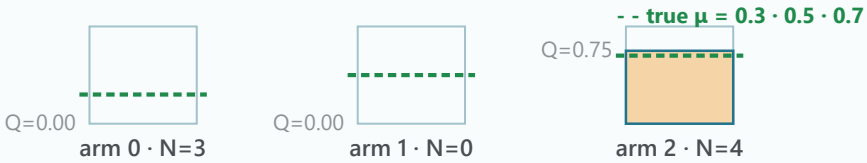
VARIABLES

`t = 7` `a = 2` `r = 0`

`N = [3, 0, 4]`

`Q = [0.00, 0.00, 0.75]`

Code Trace: ε-Greedy (ε = 0.2)



t = 8: 0.634 ≥ 0.2 → **exploit**: the arm with the largest Q is **2**.


```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
▶          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
▶              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

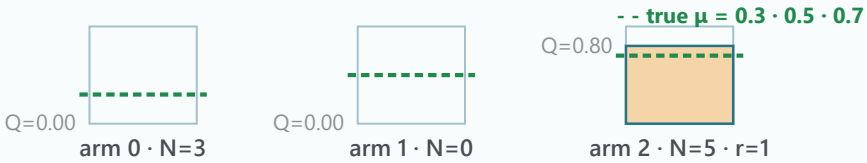
VARIABLES

t = 8 | u = 0.634 | a = 2

N = [3, 0, 4]

Q = [0.00, 0.00, 0.75]

Code Trace: ε-Greedy (ε = 0.2)



Reward **r = 1**; N[2] = 5, Q[2] = 0.80 (incremental mean).

Step 17 / 18

```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:            # explore
5              a = rng.randrange(len(mu))
6          else:                             # exploit
7              a = Q.index(max(Q))
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
11     return N, Q
```

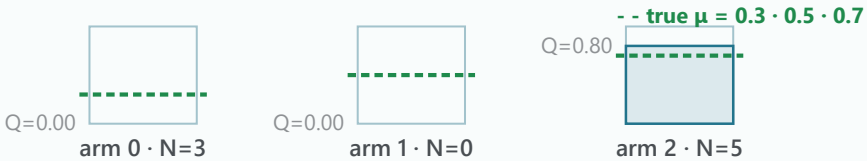
VARIABLES

t = 8 · a = 2 · r = 1

N = [3, 0, 5]

Q = [0.00, 0.00, 0.80]

Code Trace: ε-Greedy (ε = 0.2)



return N = [3, 0, 5]. The first 3 rounds got stuck on arm 0 (all rewards 0, ties go to the first arm); a single exploration step found the best arm (2).

Step 18 / 18

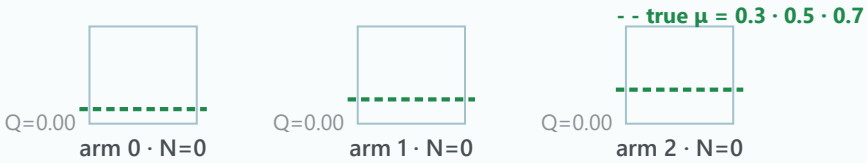
```
1  def egreedy(mu, T, eps, rng):
2      N, Q = [0] * len(mu), [0.0] * len(mu)
3      for t in range(1, T + 1):
4          if rng.random() < eps:           # explore
5              a = rng.randrange(len(mu))
6          else:                            # exploit
7              a = Q.index(max(Q))
8          r = 1 if rng.random() < mu[a] else 0
9          N[a] += 1
10         Q[a] += (r - Q[a]) / N[a]
▶      return N, Q
```

VARIABLES

N = [3, 0, 5]

Q = [0.00, 0.00, 0.80]

Code Trace: UCB1 (Upper Confidence Bound)



Call: `ucb([0.3, 0.5, 0.7], 8, random.Random(3))`. Estimates and counts start at zero. Score Q + √(2 ln t / N): t is the current round, N the arm's pull count. Rarely pulled arms get a large bonus.

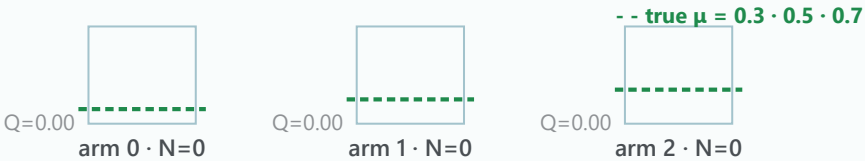
Step 1 / 18

```
1  import math
2  def ucb(mu, T, rng):
▶      K = len(mu)
▶      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

VARIABLES

K = 3 | N = [0, 0, 0] | Q = [0.00, 0.00, 0.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 1: an untried arm has bonus ∞ → the first untried arm **0** is chosen (each arm once).

Step 2 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

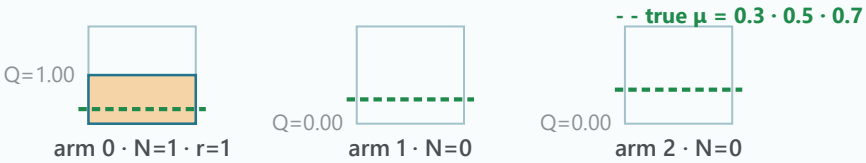
VARIABLES

`t = 1` `b = [inf, inf, inf]` `a = 0`

`N = [0, 0, 0]`

`Q = [0.00, 0.00, 0.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 1**; Q[0] = 1.00, N[0] = 1.

Step 3 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

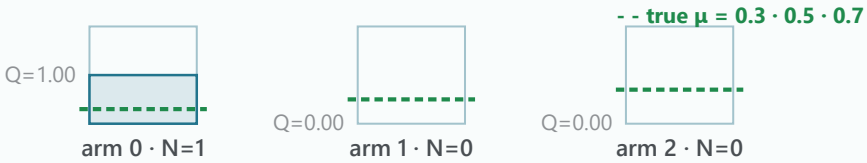
VARIABLES

t = 1 | a = 0 | r = 1

N = [1, 0, 0]

Q = [1.00, 0.00, 0.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 2: an untried arm has bonus ∞ → the first untried arm **1** is chosen (each arm once).

Step 4 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

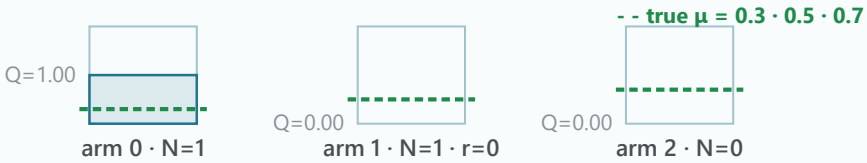
VARIABLES

`t = 2` `b = [1.18, inf, inf]` `a = 1`

`N = [1, 0, 0]`

`Q = [1.00, 0.00, 0.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 0**; Q[1] = 0.00, N[1] = 1.

Step 5 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

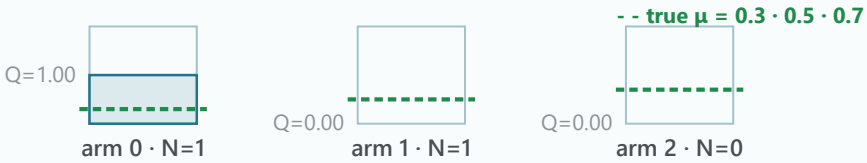
VARIABLES

t = 2 | a = 1 | r = 0

N = [1, 1, 0]

Q = [1.00, 0.00, 0.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 3: an untried arm has bonus ∞ → the first untried arm **2** is chosen (each arm once).

Step 6 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

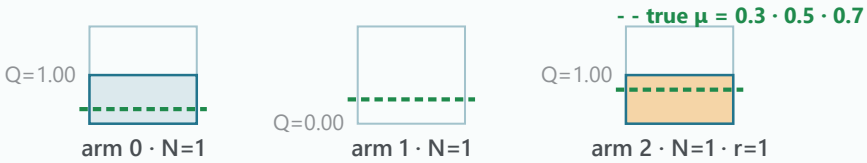
VARIABLES

`t = 3` `b = [1.48, 1.48, inf]`

`a = 2` `N = [1, 1, 0]`

`Q = [1.00, 0.00, 0.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 1**; Q[2] = 1.00, N[2] = 1.

Step 7 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

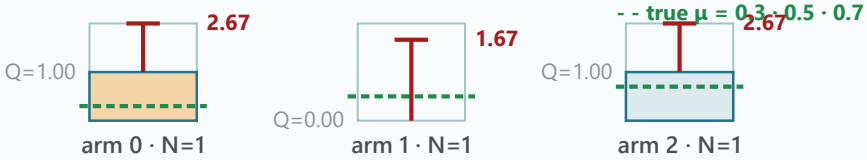
VARIABLES

t = 3 | a = 2 | r = 1

N = [1, 1, 1]

Q = [1.00, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 4: bonus √(2 ln t / N) = [1.67, 1.67, 1.67]; UCB = Q + b = [2.67, 1.67, 2.67] → largest: arm **0**.


```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

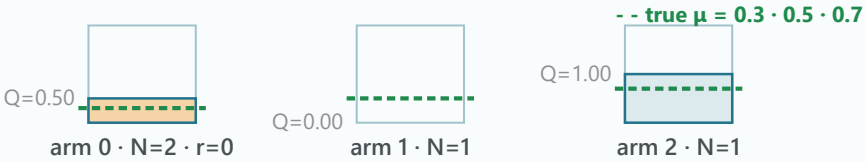
VARIABLES

t = 4 | b = [1.67, 1.67, 1.67]

a = 0 | N = [1, 1, 1]

Q = [1.00, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 0**; Q[0] = 0.50, N[0] = 2.

Step 9 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

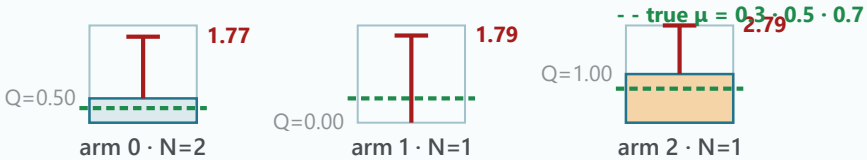
VARIABLES

`t = 4` `a = 0` `r = 0`

`N = [2, 1, 1]`

`Q = [0.50, 0.00, 1.00]`

Code Trace: UCB1 (Upper Confidence Bound)



t = 5: bonus √(2 ln t / N) = [1.27, 1.79, 1.79]; UCB = Q + b = [1.77, 1.79, 2.79] → largest: arm **2**.

Step 10 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

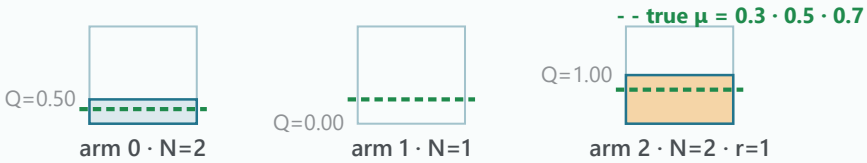
VARIABLES

`t = 5` `b = [1.27, 1.79, 1.79]`

`a = 2` `N = [2, 1, 1]`

`Q = [0.50, 0.00, 1.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 1**; Q[2] = 1.00, N[2] = 2.

Step 11 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

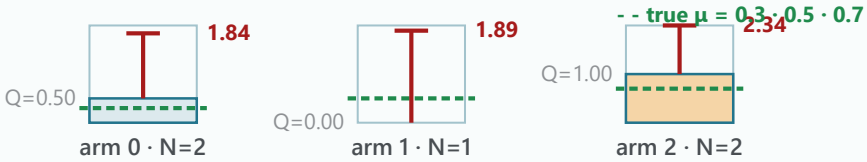
VARIABLES

t = 5 | a = 2 | r = 1

N = [2, 1, 2]

Q = [0.50, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 6: bonus √(2 ln t / N) = [1.34, 1.89, 1.34]; UCB = Q + b = [1.84, 1.89, 2.34] → largest: arm **2**.

Step 12 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

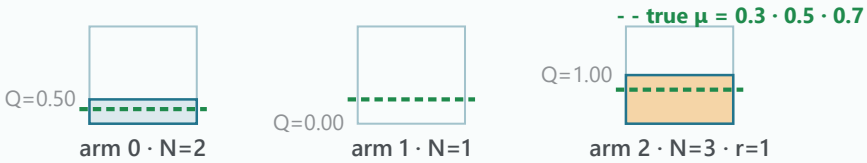
VARIABLES

`t = 6` `b = [1.34, 1.89, 1.34]`

`a = 2` `N = [2, 1, 2]`

`Q = [0.50, 0.00, 1.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 1**; Q[2] = 1.00, N[2] = 3.

Step 13 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

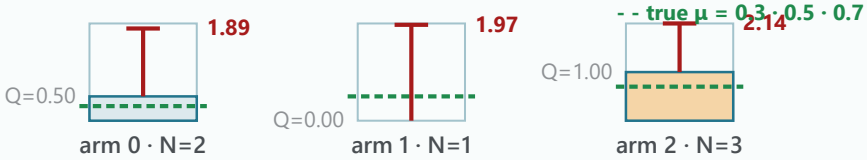
VARIABLES

t = 6 | a = 2 | r = 1

N = [2, 1, 3]

Q = [0.50, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 7: bonus √(2 ln t / N) = [1.39, 1.97, 1.14]; UCB = Q + b = [1.89, 1.97, 2.14] → largest: arm **2**.

Step 14 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

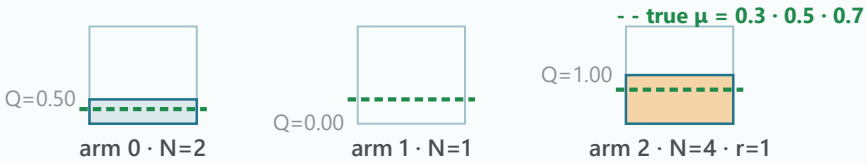
VARIABLES

`t = 7` `b = [1.39, 1.97, 1.14]`

`a = 2` `N = [2, 1, 3]`

`Q = [0.50, 0.00, 1.00]`

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 1**; Q[2] = 1.00, N[2] = 4.

Step 15 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

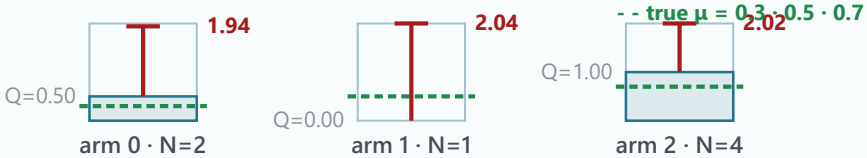
VARIABLES

t = 7 | a = 2 | r = 1

N = [2, 1, 4]

Q = [0.50, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



t = 8: bonus √(2 ln t / N) = [1.44, 2.04, 1.02]; UCB = Q + b = [1.94, 2.04, 2.02] → largest: arm **1**.

Step 16 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
▶          b = [math.inf if n == 0 else
▶               math.sqrt(2*math.log(t)/n) for n in N]
▶          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

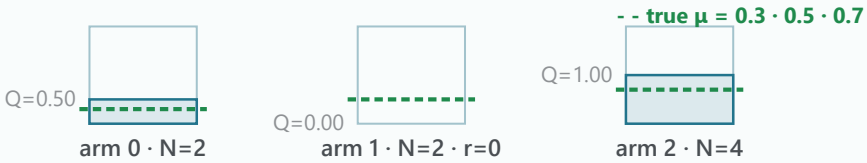
VARIABLES

t = 8 | b = [1.44, 2.04, 1.02]

a = 1 | N = [2, 1, 4]

Q = [0.50, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



Reward **r = 0**; Q[1] = 0.00, N[1] = 2.

Step 17 / 18

```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
▶          r = 1 if rng.random() < mu[a] else 0
▶          N[a] += 1
▶          Q[a] += (r - Q[a]) / N[a]
12     return N, Q
```

VARIABLES

t = 8 | a = 1 | r = 0

N = [2, 2, 4]

Q = [0.50, 0.00, 1.00]

Code Trace: UCB1 (Upper Confidence Bound)



return N = [2, 2, 4]. A rarely tried arm keeps a large bonus: at t = 8, arm 1 (one try, Q = 0) was tried again — exploration comes for free.

Step 18 / 18

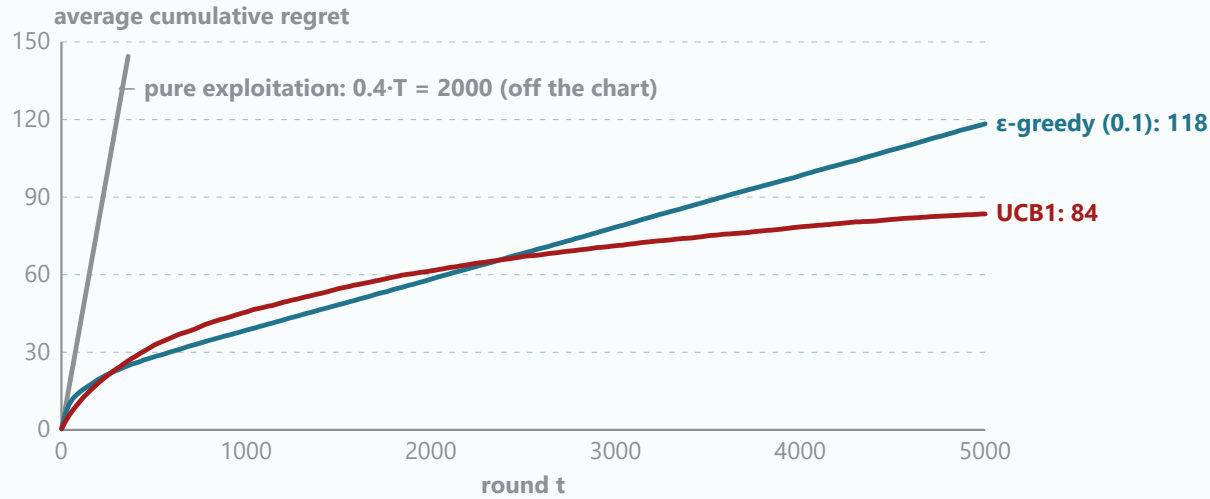
```
1  import math
2  def ucb(mu, T, rng):
3      K = len(mu)
4      N, Q = [0] * K, [0.0] * K
5      for t in range(1, T + 1):
6          b = [math.inf if n == 0 else
7               math.sqrt(2*math.log(t)/n) for n in N]
8          a = max(range(K), key=lambda i: Q[i] + b[i])
9          r = 1 if rng.random() < mu[a] else 0
10         N[a] += 1
11         Q[a] += (r - Q[a]) / N[a]
▶      return N, Q
```

VARIABLES

N = [2, 2, 4]

Q = [0.50, 0.00, 1.00]

Comparison: 5000 Rounds, 100 Runs



- Arms μ = (0.3, 0.5, 0.7), seeds 0–99. Pure exploitation gets stuck on arm 0 every time: regret 0.4·T, **linear**.
- ε-greedy keeps exploring 10% of the time, so its regret grows linearly. UCB1's regret is above ε-greedy's from roughly round 270 to round 2,370 and stays below afterwards, because UCB1's regret is **logarithmic**, i.e. it grows very slowly (Auer et al., 2002).
- ε is a tuning parameter: 0.2 in the short trace so that exploration shows up, 0.1 in the long comparison, 0.15 in the bandit game. A smaller ε means less exploration and later correction.

Back to the Game: The Multi-Armed Bandit

Theory's answer: exploration has a cost

Arms μ = (0.3, 0.5, 0.7), 100 rounds:

Strategy	Reward	Regret
always random (explore only)	50	20
stick with the 0.5 arm	50	20
ε-greedy, ε = 0.15 (best found)	67	3
always the best arm	70	0

ε-greedy explores 15% of rounds forever; UCB1 explores rarely tried arms, with logarithmic regret.

The opening question

One failed pull of Arm 3 says little. At t = 9 UCB1 picks Arm 3: $0 + \sqrt{(2 \ln 9 / 1)} \approx 2.10$ vs. $0.67 + \sqrt{(2 \ln 9 / 3)} \approx 1.88$ for Arm 1.

In practice: in a four-armed bandit task, human choices fit softmax on value differences better than a UCB-like uncertainty bonus (Daw et al., 2006).

One run is misleading: rewards are random, so compare averages over many runs.

From Bandits to Reinforcement Learning

- A bandit is a one-state problem. In **reinforcement learning** an action determines not only the reward but also the next **state** (where the agent is, e.g. the position on a board).
- **Q-learning** (Watkins, 1989): updates the state–action value from an observed transition

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a) \right]$$

α: the learning rate (how much to trust new information); γ: the discount factor for future reward.

Numerical example

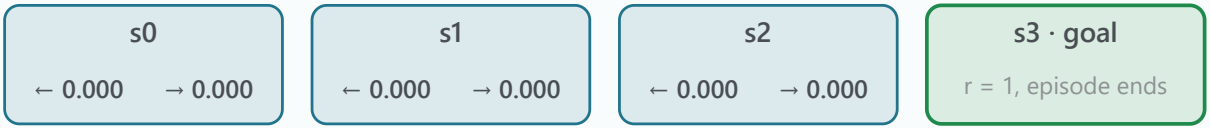
Q(s, a) = 2, r = 1, γ = 0.9, max Q(s′, ·) = 4, α = 0.5:

```
Q ← 2 + 0.5·(1 + 3.6 − 2) = 3.3
```

With multiple agents

Other agents are learning too, so from each agent's view the rules of the game keep changing (the environment is non-stationary); single-agent convergence guarantees no longer hold. Self-play (AlphaZero) is one way out.

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Call `q_learn(ep, 0.5, 0.9)` : 3 episodes starting at s0. All Q values are 0; s3 is the goal (+1).

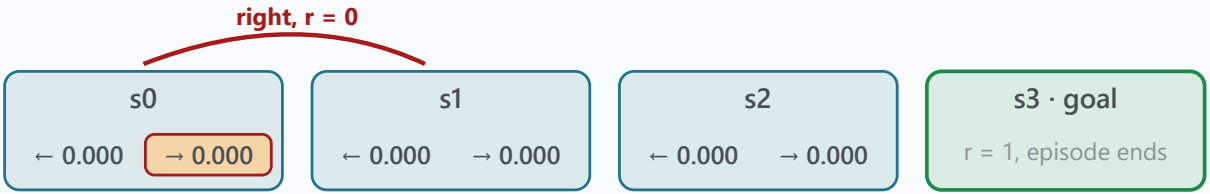
Step 1 / 13

```
1 def q_learn(episodes, alpha, gamma):
▶     Q = [[0.0, 0.0] for s in range(4)]  # [left,
  right]
3     for ep in episodes:
4         for s, a, r, s2 in ep:
5             best = 0 if s2 == 3 else max(Q[s2])
6             target = r + gamma * best
7             Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`alpha = 0.5` `gamma = 0.9` `Q = all 0`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 1: s0 → right → s1, r = 0. best = max Q(s1, ·) = 0.000; target = 0.000; Q(s0,right) = 0.000 + 0.5·(0.000 − 0.000) = **0.000**.

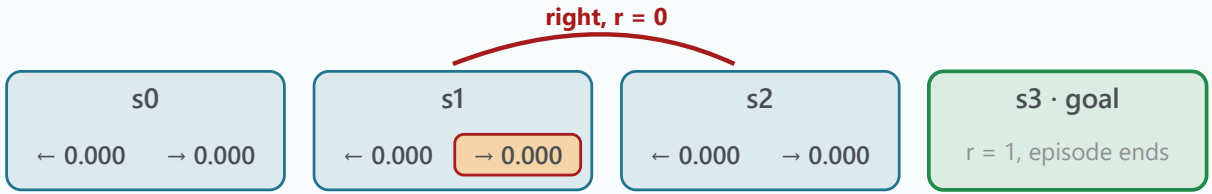
Step 2 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 1` `s = 0` `a = 1 (right)` `r = 0` `s2 = 1` `best = 0.0` `target = 0.0` `Q[s][a] = 0.0`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 1: s1 → right → s2, r = 0. best = max Q(s2, ·) = 0.000; target = 0.000; Q(s1,right) = 0.000 + 0.5·(0.000 − 0.000) = **0.000**.

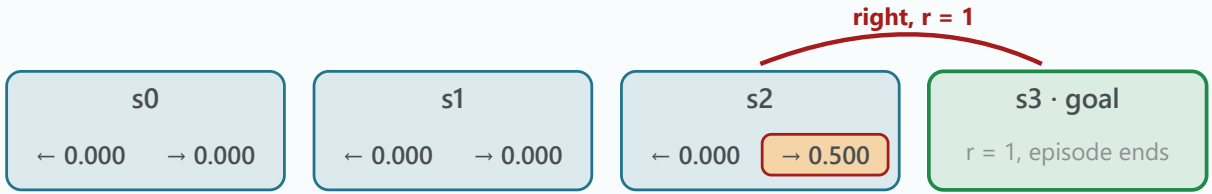
Step 3 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 1` `s = 1` `a = 1 (right)`
`r = 0` `s2 = 2` `best = 0.0`
`target = 0.0` `Q[s][a] = 0.0`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 1: s2 → right → s3, r = 1. s3 is the goal → best = 0; target = 1.000; Q(s2,right) = 0.000 + 0.5·(1.000 − 0.000) = **0.500**.

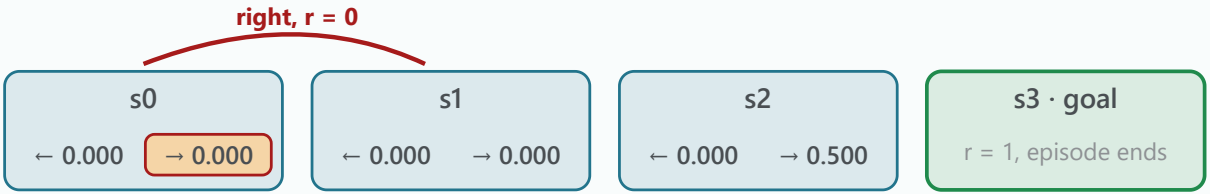
Step 4 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 1` `s = 2` `a = 1 (right)` `r = 1` `s2 = 3` `best = 0` `target = 1.0` `Q[s][a] = 0.5`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 2: s0 → right → s1, r = 0. best = max Q(s1, ·) = 0.000; target = 0.000; Q(s0,right) = 0.000 + 0.5·(0.000 − 0.000) = **0.000**.

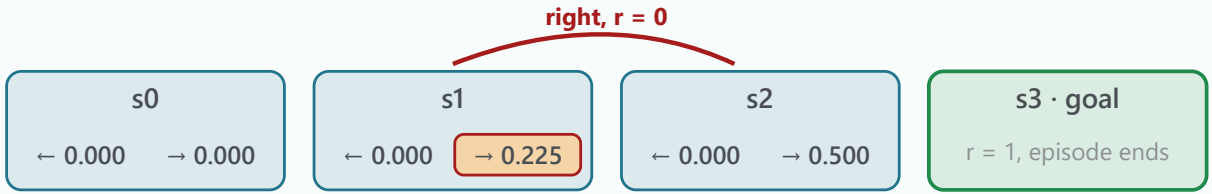
Step 5 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 2` `s = 0` `a = 1 (right)` `r = 0` `s2 = 1` `best = 0.0` `target = 0.0` `Q[s][a] = 0.0`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 2: s1 → right → s2, r = 0. best = max Q(s2, ·) = 0.500; target = 0.450; Q(s1,right) = 0.000 + 0.5·(0.450 − 0.000) = **0.225**.

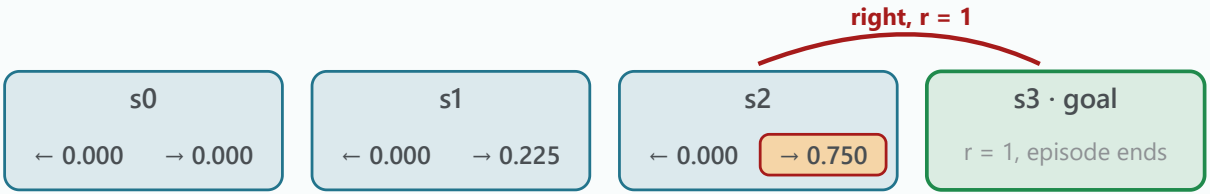
Step 6 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 2` `s = 1` `a = 1 (right)` `r = 0` `s2 = 2` `best = 0.5` `target = 0.45` `Q[s][a] = 0.225`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 2: s2 → right → s3, r = 1. s3 is the goal → best = 0; target = 1.000; Q(s2,right) = 0.500 + 0.5·(1.000 − 0.500) = **0.750**.

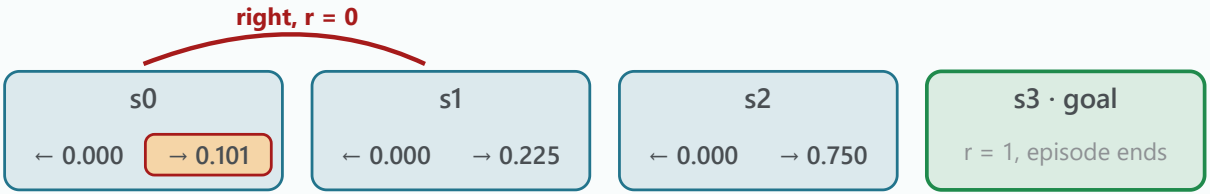
Step 7 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 2` `s = 2` `a = 1 (right)` `r = 1` `s2 = 3` `best = 0` `target = 1.0` `Q[s][a] = 0.75`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 3: s0 → right → s1, r = 0. best = max Q(s1, ·) = 0.225; target = 0.203; Q(s0,right) = 0.000 + 0.5·(0.203 − 0.000) = **0.101**.

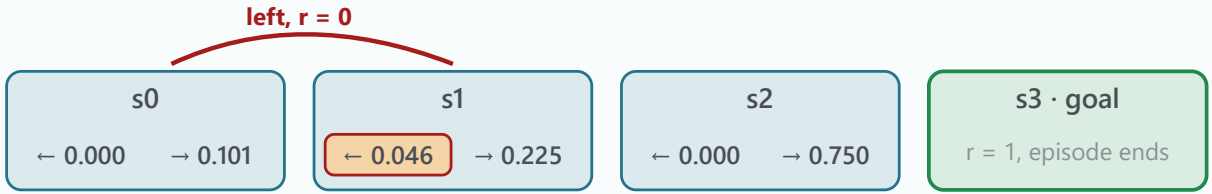
Step 8 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 3` `s = 0` `a = 1 (right)` `r = 0` `s2 = 1` `best = 0.225` `target = 0.2025` `Q[s][a] = 0.1013`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 3: s1 → left → s0, r = 0. best = max Q(s0, ·) = 0.101; target = 0.091; Q(s1,left) = 0.000 + 0.5·(0.091 − 0.000) = **0.046**.

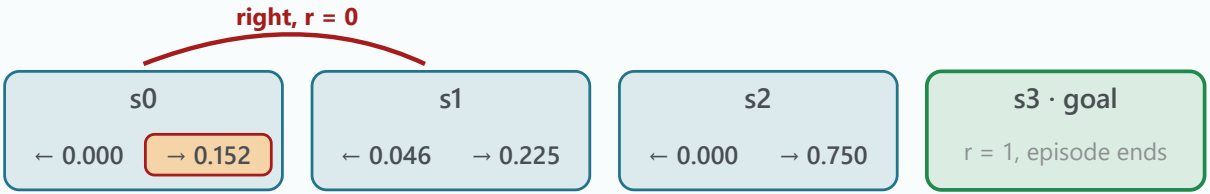
Step 9 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

ep = 3 | s = 1 | a = 0 (left) | r = 0 | s2 = 0 | best = 0.1013 | target = 0.0911 | Q[s][a] = 0.0456

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 3: s0 → right → s1, r = 0. best = max Q(s1, ·) = 0.225; target = 0.203; Q(s0,right) = 0.101 + 0.5·(0.203 − 0.101) = **0.152**.

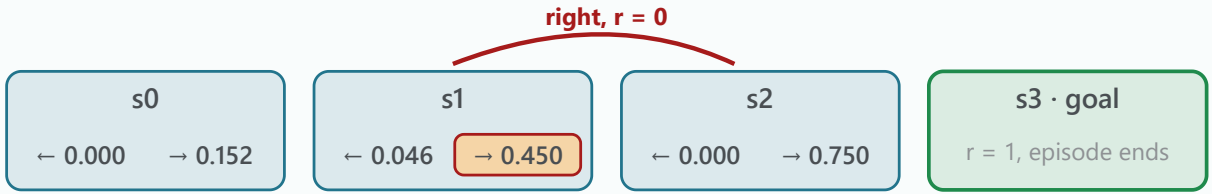
Step 10 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 3` `s = 0` `a = 1 (right)` `r = 0` `s2 = 1` `best = 0.225` `target = 0.2025` `Q[s][a] = 0.1519`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 3: s1 → right → s2, r = 0. best = max Q(s2, ·) = 0.750; target = 0.675; Q(s1,right) = 0.225 + 0.5·(0.675 − 0.225) = **0.450**.

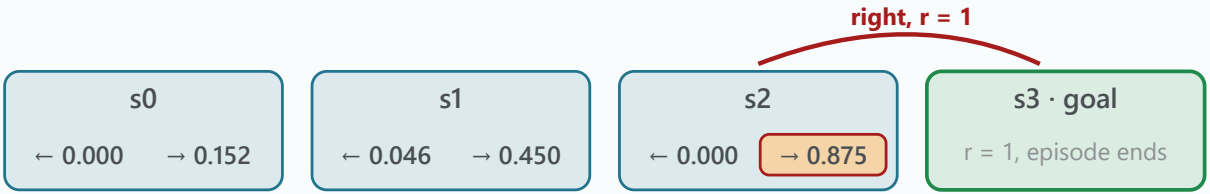
Step 11 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 3` `s = 1` `a = 1 (right)` `r = 0` `s2 = 2` `best = 0.75` `target = 0.675` `Q[s][a] = 0.45`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



Episode 3: s2 → right → s3, r = 1. s3 is the goal → best = 0; target = 1.000; Q(s2,right) = 0.750 + 0.5·(1.000 − 0.750) = **0.875**.

Step 12 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left, right]
3     for ep in episodes:
          for s, a, r, s2 in ep:
              best = 0 if s2 == 3 else max(Q[s2])
              target = r + gamma * best
              Q[s][a] += alpha * (target - Q[s][a])
8     return Q
```

VARIABLES

`ep = 3` `s = 2` `a = 1 (right)` `r = 1` `s2 = 3` `best = 0` `target = 1.0` `Q[s][a] = 0.875`

Code Trace: Q-Learning (α = 0.5, γ = 0.9)



return Q. Reward propagated back from the goal: Q(s2,right) = 0.875 → Q(s1,right) = 0.450 → Q(s0,right) = 0.152. In every cell Q(right) > Q(left): the greedy policy is "always go right".

Step 13 / 13

```
1 def q_learn(episodes, alpha, gamma):
2     Q = [[0.0, 0.0] for s in range(4)]  # [left,
  right]
3     for ep in episodes:
4         for s, a, r, s2 in ep:
5             best = 0 if s2 == 3 else max(Q[s2])
6             target = r + gamma * best
7             Q[s][a] += alpha * (target - Q[s][a])
▶     return Q
```

VARIABLES

Q[0] = [0.0, 0.152]

Q[1] = [0.046, 0.45]

Q[2] = [0.0, 0.875]

Beyond Q-Learning: Methods and Milestones

- **Thompson sampling** (1933): keeps a distribution over each arm's possible mean (the posterior), draws a sample from it each round and pulls the arm with the largest sample.
- **Win-Stay, Lose-Shift** (WSLS; Nowak–Sigmund, 1993): after a 3 or 5 repeat the move, after a 0 or 1 switch; in the Prisoner's Dilemma it recovers cooperation two rounds after a single error.
- **Others:** EXP3 (Hedge that sees only the pulled arm's reward; its regret grows slower than T even if an adversary picks the rewards), SARSA and DQN (a neural network instead of a Q table), CFR (a regret-based method for poker).

Year	System	Method
1992	TD-Gammon (backgammon)	self-play
2017–19	Libratus, Pluribus (poker)	CFR (regret)
2018	AlphaZero (chess, Go)	self-play

Games are a good laboratory for RL: clear rules, a clear reward, and millions of repetitions; in poker, regret-based methods came out on top.

Replicator Dynamics

- In a large population, individuals are matched at random; each plays a single strategy and passes it on to its offspring. No rationality, just **selection**.
- f_i: the average payoff of strategy i (its **fitness**); $\bar{f}$: the population's average payoff. Strategies that earn above average spread.
- The discrete form is enough for this course: **new share = old share × f_i / $\bar{f}$** (a baseline of 3 is added to every payoff to keep fitness positive).
- **Evolutionarily stable strategy (ESS)**: a small group of mutants cannot invade (Maynard Smith–Price, 1973).

Individual \ Opponent	Hawk	Dove
Hawk	(−1 , −1)	(4 , 0)
Dove	(0 , 4)	(2 , 2)

Hawk–Dove: resource V = 4, cost of fighting C = 6. Two hawks fight: (V − C)/2 = −1; a hawk against a dove gets V = 4 and the dove 0; two doves share: V/2 = 2.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1					
2					
3					
4					
5					
6					
7					

$x = 0.100$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 6.50$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.80$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2					
3					
4					
5					
6					
7					

$x = 0.131$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 6.35$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.74$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3					
4					
5					
6					
7					

$x = 0.168$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 6.16$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.66$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3	0.210	5.95	4.58	4.87	0.257
4					
5					
6					
7					

$x = 0.210$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 5.95$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.58$. $f_H > \bar{f}$ → hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3	0.210	5.95	4.58	4.87	0.257
4	0.257	5.72	4.49	4.80	0.306
5					
6					
7					

$x = 0.257$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 5.72$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.49$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3	0.210	5.95	4.58	4.87	0.257
4	0.257	5.72	4.49	4.80	0.306
5	0.306	5.47	4.39	4.72	0.354
6					
7					

$x = 0.306$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 5.47$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.39$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3	0.210	5.95	4.58	4.87	0.257
4	0.257	5.72	4.49	4.80	0.306
5	0.306	5.47	4.39	4.72	0.354
6	0.354	5.23	4.29	4.62	0.401
7					

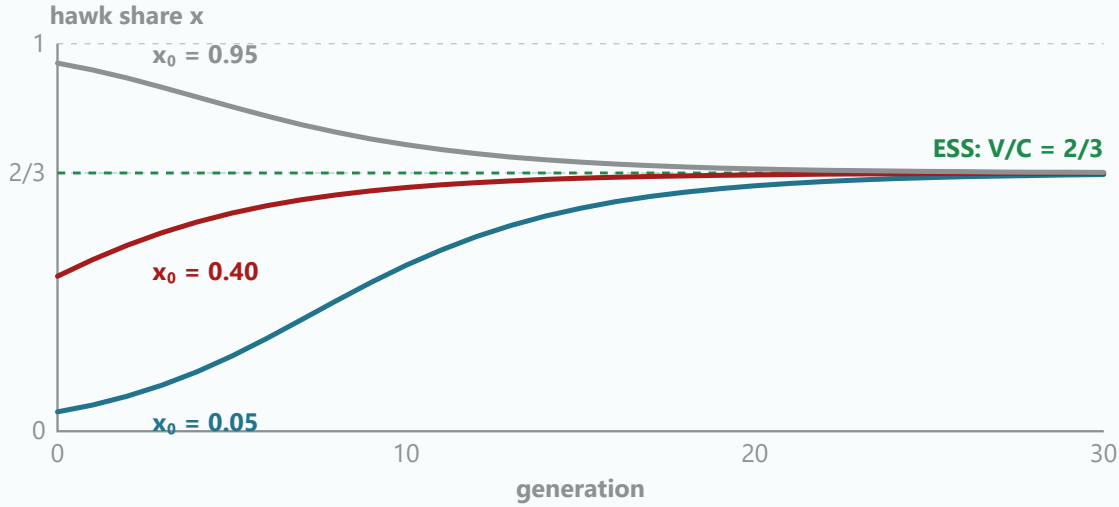
$x = 0.354$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 5.23$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.29$. $f_H > \bar{f} \rightarrow$ hawks increase.

Replicator Dynamics Step by Step: Hawk–Dove

generation	x (hawk share)	f_H	f_D	$\bar{f}$	new x = $x \cdot f_H / \bar{f}$
0	0.100	6.50	4.80	4.97	0.131
1	0.131	6.35	4.74	4.95	0.168
2	0.168	6.16	4.66	4.92	0.210
3	0.210	5.95	4.58	4.87	0.257
4	0.257	5.72	4.49	4.80	0.306
5	0.306	5.47	4.39	4.72	0.354
6	0.354	5.23	4.29	4.62	0.401
7	0.401	5.00	4.20	4.52	0.443

$x = 0.401$: $f_H = 3 + (-1) \cdot x + 4 \cdot (1 - x) = 5.00$, $f_D = 3 + 0 \cdot x + 2 \cdot (1 - x) = 4.20$. $f_H > \bar{f}$ → hawks increase.

From Every Interior Start ($0 < x_0 < 1$) to the ESS: $x^* = 2/3$



- Rest point $f_H = f_D$: $7 - 5x = 5 - 2x \Rightarrow$ **$x^* = 2/3$**, the same as the game's symmetric mixed Nash equilibrium.
- In Rock–Paper–Scissors the replicator dynamics **cycle** around the equilibrium; in the Stag Hunt they reach one of the two pure equilibria, depending on the start.

The Repeated Prisoner's Dilemma

Player 1 \ Player 2	C (cooperate)	D (defect)
C (cooperate)	(3, 3)	(0, **5**)
D (defect)	(**5**, 0)	(**1**, **1**)

Points: temptation 5 > mutual cooperation 3 > mutual defection 1 > sucker 0. C: stay silent.

- One-shot, or a known finite number of rounds (backward induction): defect every round.
- If play continues with probability δ (**the shadow of the future**), grim trigger can sustain cooperation:

Comply: 3 every round ⇒ $3 + 3\delta + \ldots = \mathbf{3/(1-\delta)}$.

The Repeated Prisoner's Dilemma

Player 1 \ Player 2	C (cooperate)	D (defect)
C (cooperate)	(3, 3)	(0, **5**)
D (defect)	(**5**, 0)	(**1**, **1**)

Points: temptation 5 > mutual cooperation 3 > mutual defection 1 > sucker 0. C: stay silent.

- One-shot, or a known finite number of rounds (backward induction): defect every round.
- If play continues with probability δ (**the shadow of the future**), grim trigger can sustain cooperation:

Comply: 3 every round ⇒ $3 + 3\delta + \ldots = \mathbf{3/(1-\delta)}$.

Deviate: 5 now, then 1 forever ⇒ $\mathbf{5 + \delta/(1-\delta)}$.

The Repeated Prisoner's Dilemma

Player 1 \ Player 2	C (cooperate)	D (defect)
C (cooperate)	(3 , 3)	(0 , **5**)
D (defect)	(**5** , 0)	(**1** , **1**)

Points: temptation 5 > mutual cooperation 3 > mutual defection 1 > sucker 0. C: stay silent.

- One-shot, or a known finite number of rounds (backward induction): defect every round.
- If play continues with probability δ (**the shadow of the future**), grim trigger can sustain cooperation:

Comply: 3 every round ⇒ $3 + 3\delta + \ldots = \mathbf{3/(1-\delta)}$.

Deviate: 5 now, then 1 forever ⇒ $\mathbf{5 + \delta/(1-\delta)}$.

Comply ≥ deviate; times $(1-\delta)$: $3 \geq 5(1-\delta) + \delta \Leftrightarrow \mathbf{\delta \geq (5-3)/(5-1) = 1/2}$.

The Repeated Prisoner's Dilemma

Player 1 \ Player 2	C (cooperate)	D (defect)
C (cooperate)	(3, 3)	(0, **5**)
D (defect)	(**5**, 0)	(**1**, **1**)

Points: temptation 5 > mutual cooperation 3 > mutual defection 1 > sucker 0. C: stay silent.

- One-shot, or a known finite number of rounds (backward induction): defect every round.
- If play continues with probability δ (**the shadow of the future**), grim trigger can sustain cooperation:

Comply: 3 every round ⇒ $3 + 3\delta + \ldots = \mathbf{3/(1-\delta)}$.

Deviate: 5 now, then 1 forever ⇒ $\mathbf{5 + \delta/(1-\delta)}$.

Comply ≥ deviate; times $(1-\delta)$: $3 \geq 5(1-\delta) + \delta \Leftrightarrow \mathbf{\delta \geq (5-3)/(5-1) = 1/2}$.

$\delta = 0.9$: comply 30, deviate 14 ⇒ cooperate. $\delta = 0.3$: comply ≈ 4.29, deviate ≈ 5.43 ⇒ defect.

The Repeated Prisoner's Dilemma

Player 1 \ Player 2	C (cooperate)	D (defect)
C (cooperate)	(3, 3)	(0, **5**)
D (defect)	(**5**, 0)	(**1**, **1**)

Points: temptation 5 > mutual cooperation 3 > mutual defection 1 > sucker 0. C: stay silent.

- One-shot, or a known finite number of rounds (backward induction): defect every round.
- If play continues with probability δ (**the shadow of the future**), grim trigger can sustain cooperation:

Comply: 3 every round ⇒ $3 + 3\delta + \ldots = \mathbf{3/(1-\delta)}$.

Deviate: 5 now, then 1 forever ⇒ $\mathbf{5 + \delta/(1-\delta)}$.

Comply ≥ deviate; times $(1-\delta)$: $3 \geq 5(1-\delta) + \delta \Leftrightarrow \mathbf{\delta \geq (5-3)/(5-1) = 1/2}$.

$\delta = 0.9$: comply 30, deviate 14 ⇒ cooperate. $\delta = 0.3$: comply ≈ 4.29, deviate ≈ 5.43 ⇒ defect.

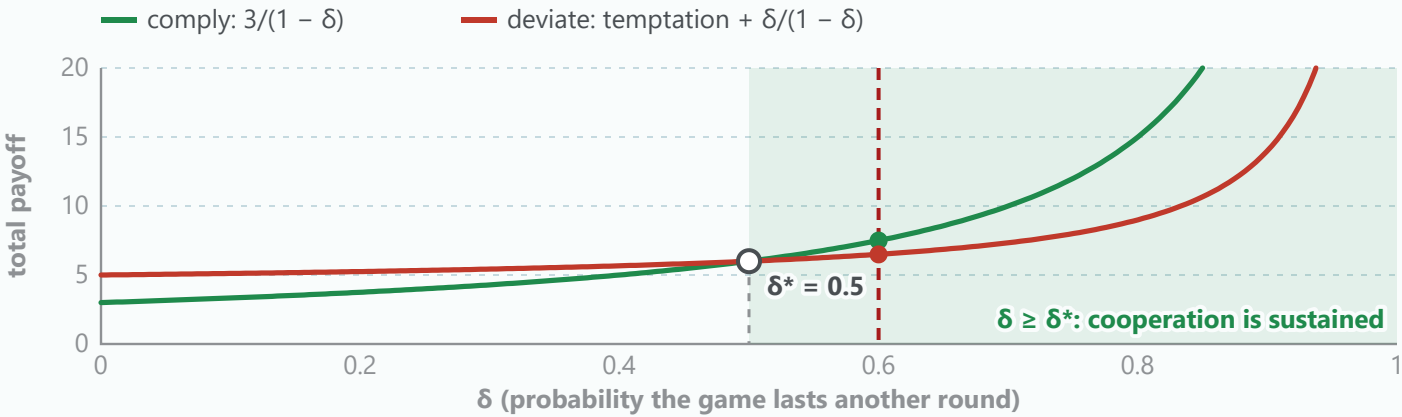
The punishment is credible: (D, D) is a stage-game equilibrium. **Folk theorem**: for large δ, any average payoffs above what defecting guarantees can be sustained.

Grim Trigger: For Which δ Does Cooperation Last?

Comply: **3/(1 − δ)** · deviate: **temptation + δ/(1 − δ)** · comply ≥ deviate ⇔ **δ ≥ δ*** · δ* = (temptation − 3)/(temptation − 1)

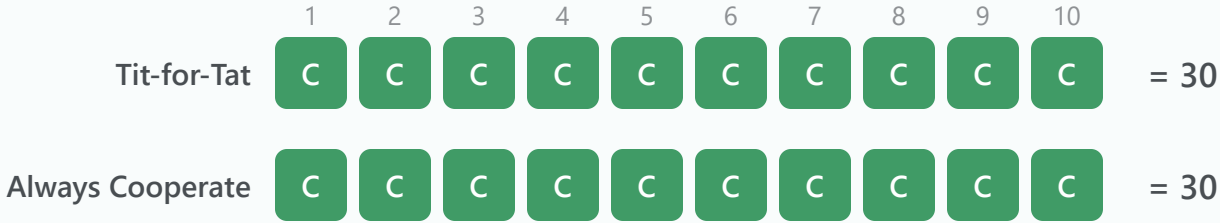
δ = 0.6 **temptation =** 5

comply = **7.5** deviate = 5 + δ/(1 − δ) = **6.5** threshold δ* = **1/2 = 0.5** **Cooperation is sustainable**



The more the future is worth (the larger δ), the stronger the deterrent of the grim trigger; as temptation rises, so does the threshold δ* needed for cooperation.

10-Round Matches: Who Plays What?

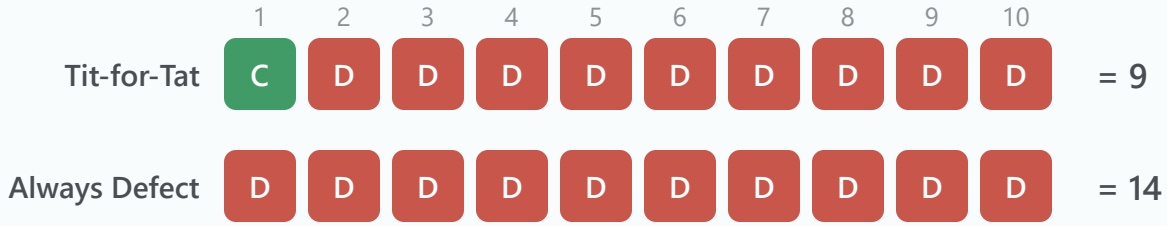


TFT answers cooperation with cooperation: 30 points each. (Total: 30 – 30)

Step 1 / 5

C: staying silent, i.e. cooperating (green); D: defecting (red). TFT: cooperate in round 1, then repeat the opponent's last move.

10-Round Matches: Who Plays What?



TFT is exploited in round 1, then always defects: 9 to 14. TFT can **never win** a match (at best it ties). (Total: 9 – 14)

C: staying silent, i.e. cooperating (green); D: defecting (red). TFT: cooperate in round 1, then repeat the opponent's last move.

10-Round Matches: Who Plays What?



Suspicious TFT opens with defection; the two TFTs "echo" each other → alternating C/D, 25 each. (Total: 25 – 25)

C: staying silent, i.e. cooperating (green); D: defecting (red). TFT: cooperate in round 1, then repeat the opponent's last move.

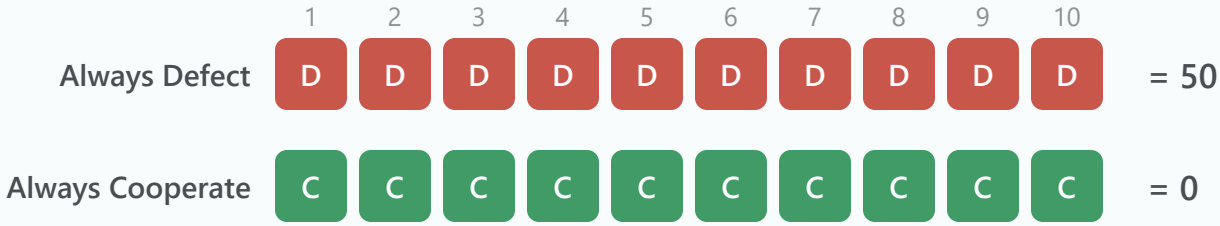
10-Round Matches: Who Plays What?



Grim never forgives a single defection: from round 2 on, mutual defection. (Total: 13 – 13)

C: staying silent, i.e. cooperating (green); D: defecting (red). TFT: cooperate in round 1, then repeat the opponent's last move.

10-Round Matches: Who Plays What?



Always Defect fully exploits the unconditional cooperator: 50 to 0. (Total: 50 – 0)

C: staying silent, i.e. cooperating (green); D: defecting (red). TFT: cooperate in round 1, then repeat the opponent's last move.

Code Trace: `match(tft, stft, 6)`



Call `match(tft, stft, 6)`: histories empty, scores 0.

Step 1 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
▶      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`T = 6` `h1 = []` `h2 = []` `p1 = 0` `p2 = 0`

Code Trace: `match(tft, stft, 6)`



t = 0: With empty histories, TFT opens with **C** and Suspicious TFT with **D**.


```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
   tft = lambda me, opp: opp[-1] if opp else "C"
   stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
       for t in range(T):
           a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 0` `a = C` `b = D`

`h1 = []` `h2 = []` `p1 = 0`

`p2 = 0`

Code Trace: `match(tft, stft, 6)`



PAY["CD"] = (0, 5) → scores **0 – 5**.

Step 3 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 0` `u = 0` `v = 5`

`h1 = [C]` `h2 = [D]` `p1 = 0`

`p2 = 5`

Code Trace: `match(tft, stft, 6)`



t = 1: Both copy the opponent's last move: TFT → **D**, STFT → **C**.


```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
▶  tft = lambda me, opp: opp[-1] if opp else "C"
▶  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
▶      for t in range(T):
▶          a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

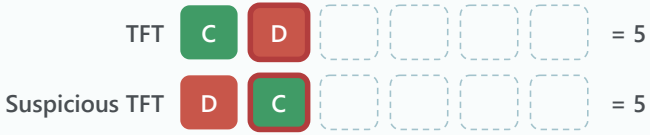
VARIABLES

`t = 1` `a = D` `b = C`

`h1 = [C]` `h2 = [D]` `p1 = 0`

`p2 = 5`

Code Trace: `match(tft, stft, 6)`



PAY["DC"] = (5, 0) → scores **5 – 5**.

Step 5 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

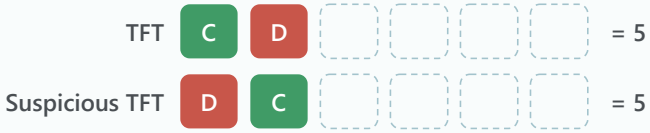
VARIABLES

`t = 1` `u = 5` `v = 0`

`h1 = [C, D]` `h2 = [D, C]`

`p1 = 5` `p2 = 5`

Code Trace: `match(tft, stft, 6)`



t = 2: Both copy the opponent's last move: TFT → **C**, STFT → **D**.


```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
▶  tft = lambda me, opp: opp[-1] if opp else "C"
▶  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
▶      for t in range(T):
▶          a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

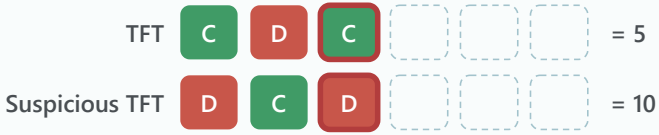
VARIABLES

`t = 2` `a = C` `b = D`

`h1 = [C, D]` `h2 = [D, C]`

`p1 = 5` `p2 = 5`

Code Trace: `match(tft, stft, 6)`



PAY["CD"] = (0, 5) → scores **5 – 10**.

Step 7 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 2` `u = 0` `v = 5`

`h1 = [C, D, C]`

`h2 = [D, C, D]` `p1 = 5`

`p2 = 10`

Code Trace: `match(tft, stft, 6)`



t = 3: Both copy the opponent's last move: TFT → **D**, STFT → **C**.


```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
▶  tft = lambda me, opp: opp[-1] if opp else "C"
▶  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
▶      for t in range(T):
▶          a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 3` `a = D` `b = C`

`h1 = [C, D, C]`

`h2 = [D, C, D]` `p1 = 5`

`p2 = 10`

Code Trace: `match(tft, stft, 6)`



PAY["DC"] = (5, 0) → scores **10 – 10**.

Step 9 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 3` `u = 5` `v = 0`

`h1 = [C, D, C, D]`

`h2 = [D, C, D, C]` `p1 = 10`

`p2 = 10`

Code Trace: `match(tft, stft, 6)`



t = 4: Both copy the opponent's last move: TFT → **C**, STFT → **D**.

Step 10 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
   tft = lambda me, opp: opp[-1] if opp else "C"
   stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
       for t in range(T):
           a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 4` `a = C` `b = D`

`h1 = [C, D, C, D]`

`h2 = [D, C, D, C]` `p1 = 10`

`p2 = 10`

Code Trace: `match(tft, stft, 6)`



PAY["CD"] = (0, 5) → scores **10 – 15**.

Step 11 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 4` `u = 0` `v = 5`

`h1 = [C, D, C, D, C]`

`h2 = [D, C, D, C, D]`

`p1 = 10` `p2 = 15`

Code Trace: `match(tft, stft, 6)`



t = 5: Both copy the opponent's last move: TFT → **D**, STFT → **C**.

Step 12 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
   tft = lambda me, opp: opp[-1] if opp else "C"
   stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
       for t in range(T):
           a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 5` `a = D` `b = C`

`h1 = [C, D, C, D, C]`

`h2 = [D, C, D, C, D]`

`p1 = 10` `p2 = 15`

Code Trace: `match(tft, stft, 6)`



PAY["DC"] = (5, 0) → scores **15 – 15**.

Step 13 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
▶          u, v = PAY[a + b]
▶          h1.append(a); h2.append(b)
▶          p1, p2 = p1 + u, p2 + v
12     return p1, p2
```

VARIABLES

`t = 5` `u = 5` `v = 0`

`h1 = [C, D, C, D, C, D]`

`h2 = [D, C, D, C, D, C]`

`p1 = 15` `p2 = 15`

Code Trace: `match(tft, stft, 6)`



return (15, 15). A single opening defection starts an endless "echo": CD/DC alternating, 2.5 points per round on average (mutual cooperation would give 3).

Step 14 / 14

```
1  PAY = {"CC": (3, 3), "CD": (0, 5),
2         "DC": (5, 0), "DD": (1, 1)}
3  tft = lambda me, opp: opp[-1] if opp else "C"
4  stft = lambda me, opp: opp[-1] if opp else "D"
5  def match(s1, s2, T):
6      h1, h2, p1, p2 = [], [], 0, 0
7      for t in range(T):
8          a, b = s1(h1, h2), s2(h2, h1)
9          u, v = PAY[a + b]
10         h1.append(a); h2.append(b)
11         p1, p2 = p1 + u, p2 + v
▶      return p1, p2
```

VARIABLES

- h1 = [C, D, C, D, C, D]
- h2 = [D, C, D, C, D, C]
- p1 = 15
- p2 = 15

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	
Always Defect							
Tit-for-Tat							
Grim Trigger							
Suspicious TFT							

Always Cooperate plays 10 rounds against everyone (including its twin): total **117**; it scores most against Always Cooperate (30).

Step 1 / 6

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	
Always Defect	50	10	14	14	10	**98**	
Tit-for-Tat							
Grim Trigger							
Suspicious TFT							

Always Defect plays 10 rounds against everyone (including its twin): total **98**; it scores most against Always Cooperate (50).

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	
Always Defect	50	10	14	14	10	**98**	
Tit-for-Tat	30	9	30	30	25	**124**	
Grim Trigger							
Suspicious TFT							

Tit-for-Tat plays 10 rounds against everyone (including its twin): total **124**; it scores most against Always Cooperate (30).

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	
Always Defect	50	10	14	14	10	**98**	
Tit-for-Tat	30	9	30	30	25	**124**	
Grim Trigger	30	9	30	30	13	**112**	
Suspicious TFT							

Grim Trigger plays 10 rounds against everyone (including its twin): total **112**; it scores most against Always Cooperate (30).

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	
Always Defect	50	10	14	14	10	**98**	
Tit-for-Tat	30	9	30	30	25	**124**	
Grim Trigger	30	9	30	30	13	**112**	
Suspicious TFT	32	10	25	13	10	**90**	

Suspicious TFT plays 10 rounds against everyone (including its twin): total **90**; it scores most against Always Cooperate (32).

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

A Mini Axelrod Tournament: Step by Step

Strategy \ opponent	All-C	All-D	TFT	Grim	STFT	Total	Rank
Always Cooperate	30	0	30	30	27	**117**	2
Always Defect	50	10	14	14	10	**98**	4
Tit-for-Tat	30	9	30	30	25	**124**	1
Grim Trigger	30	9	30	30	13	**112**	3
Suspicious TFT	32	10	25	13	10	**90**	5

Ranking: Tit-for-Tat (124), Always Cooperate (117), Grim Trigger (112), Always Defect (98), Suspicious TFT (90). **Tit-for-Tat comes first** without winning a single match! Always Defect is second to last.

Points scored in 10 rounds by the row strategy against the column opponent. All-C: Always Cooperate, All-D: Always Defect, STFT: Suspicious TFT.

Axelrod's Tournaments (1980) and Their Lessons

Axelrod's two computer tournaments (14 and 62 programs) were both won by Rapoport's four-line **Tit-for-Tat**.

Nice	**Retaliatory**	**Forgiving**	**Clear**
Never be the first to defect.	Punish defection.	Cooperate again when they do.	Be easy to read.

- With **noise** (mistaken moves), two TFTs retaliate endlessly; *generous TFT* sometimes forgives and does better.
- Success depends on the pool of opponents; in an evolutionary (replicator) tournament, winners multiply.



Two TFTs, one error by A in round 4: an endless C/D echo. 29 – 24 over 10 rounds (30 – 30 without it).

SUMMARY

Discussion: Learning, People and Ethics

Does learning always lead to the optimum?

No. Some dynamics cycle (Matching Pennies, Shapley's example); the environment and opponents change; a wrong model leads to bad convergence. The reliable target is usually *no regret*.

Do people learn like algorithms?

Similar: trial and error, learning from the past. Different: people act on risk perception, emotions and social norms; algorithms follow a defined reward.

Ethical risks of adaptive systems

Data bias, recommendations that manipulate users (clickbait), privacy, algorithmic price coordination (tacit collusion among learning pricing bots).

Summary

Method	Description	Outcome
Best-response dynamics	Best response to the last action	Equilibrium in potential games; otherwise cycles
Fictitious play	Best response to past frequencies	Frequencies reach equilibrium in zero-sum games
Hedge	$w \leftarrow w \cdot e^{\eta g}$	$R_T = O(\sqrt{T \ln n})$; time average → CCE
ε-greedy / UCB	Balancing exploration and exploitation	UCB: logarithmic regret
Q-learning	$Q \leftarrow Q + \alpha(r + \gamma \cdot \max Q' - Q)$	Reward propagates back from the goal
Replicator	Successful strategies spread	Hawk–Dove: $x^* = V/C$
Repeated PD	The shadow of the future	Cooperation if $\delta \geq 1/2$; TFT is strong

Sample Problems

1. Hedge with $\eta = \ln 2$, two actions, payoffs (1, 0) and (1, 0) in the first two rounds. What is p_1 in round 3?
2. UCB1 at $t = 10$: arm 1 has $Q = 0.6$, $N = 5$; arm 2 has $Q = 0.2$, $N = 1$. Which arm is chosen?
3. In Hawk–Dove ($V = 4$, $C = 6$, baseline 3) with $x = 0.5$, does the hawk share increase in the next generation?
4. With temptation 5, mutual cooperation 3 and mutual defection 1, what is the smallest δ for which Grim sustains cooperation? What if the temptation were 7?
5. How does fictitious play differ from best-response dynamics? Which one "approaches" equilibrium in Matching Pennies?
6. In Q-learning, $Q(s, a) = 1$, $r = 0$, $\gamma = 0.9$, $\max Q(s', \cdot) = 2$, $\alpha = 0.1$. What is $Q(s, a)$ after the update?

▶ **Show answers**

SUMMARY

Try It Yourself

Nashpy (Python library)

Fictitious play (`fictitious_play`) and replicator dynamics (`replicator_dynamics`) on any payoff matrix.

References

- D. Fudenberg, D. Levine, *The Theory of Learning in Games*, MIT Press, 1998.
- N. Cesa-Bianchi, G. Lugosi, *Prediction, Learning, and Games*, Cambridge, 2006.
- T. Lattimore, C. Szepesvári, *Bandit Algorithms*, Cambridge, 2020.
- P. Auer, N. Cesa-Bianchi, P. Fischer, "Finite-time analysis of the multiarmed bandit problem", *Machine Learning* 47, 2002.
- C. Watkins, P. Dayan, "Q-learning", *Machine Learning* 8, 1992; A. Blum, Y. Mansour, "From external to internal regret", *JMLR* 8, 2007.
- R. Axelrod, *The Evolution of Cooperation*, Basic Books, 1984.
- J. Hofbauer, K. Sigmund, *Evolutionary Games and Population Dynamics*, Cambridge, 1998.
- T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*, 2016 (Lectures 16–18); course page: **CS364A (Stanford, 2013)**.
- N. Nisan, T. Roughgarden, É. Tardos, V. Vazirani (eds.), *Algorithmic Game Theory*, Cambridge, 2007 (Chapter 4: learning, regret minimization and equilibria).
- A. R. Karlin, Y. Peres, *Game Theory, Alive*, AMS, 2017 (Chapter 7: evolutionary equilibria; Chapter 18: adaptive decision making); D. Easley, J. Kleinberg, *Networks, Crowds, and Markets*, 2010 (Chapter 7: evolutionary game theory).
- Tool: **Nashpy** (fictitious play, replicator dynamics).

References (2/2)

- J. Robinson, “An iterative method of solving a game”, *Annals of Mathematics* 54(2), 1951; K. Miyasawa, “On the convergence of the learning process in a 2×2 non-zero-sum two-person game”, Princeton Research Memo 33, 1961; L. S. Shapley, “Some topics in two-person games”, *Advances in Game Theory*, 1964.
- D. Monderer, L. S. Shapley, “Potential games”, *Games and Economic Behavior* 14(1), 1996.
- D. Foster, R. Vohra, “Calibrated learning and correlated equilibrium”, *Games and Economic Behavior* 21, 1997; S. Hart, A. Mas-Colell, “A simple adaptive procedure leading to correlated equilibrium”, *Econometrica* 68(5), 2000.
- J. Maynard Smith, G. R. Price, “The logic of animal conflict”, *Nature* 246, 1973.
- W. R. Thompson, “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples”, *Biometrika* 25, 1933.
- P. Auer, N. Cesa-Bianchi, Y. Freund, R. E. Schapire, “The nonstochastic multiarmed bandit problem”, *SIAM Journal on Computing* 32(1), 2002.
- N. D. Daw, J. P. O’Doherty, P. Dayan, B. Seymour, R. J. Dolan, “Cortical substrates for exploratory decisions in humans”, *Nature* 441, 2006.
- C. Watkins, *Learning from Delayed Rewards*, PhD thesis, University of Cambridge, 1989; V. Mnih et al., “Human-level control through deep reinforcement learning”, *Nature* 518, 2015.
- M. Zinkevich, M. Johanson, M. Bowling, C. Piccione, “Regret minimization in games with incomplete information”, *NeurIPS*, 2007.
- M. Nowak, K. Sigmund, “A strategy of win-stay, lose-shift that outperforms tit-for-tat in the Prisoner’s Dilemma game”, *Nature* 364, 1993.

The End

Chapter 13: Learning and Adaptation

Previous: Chapter 12: Behavioral and Collective Decisions · Next: Chapter 14: Future Directions