The Trick Within the Trick:

Psychology, Information Theory, and Penney's Game

Neshanth Anand | CS109 | Stanford University | Winter 2026

---

1. The Trick

Here is how the demo plays out: you open a webpage and are invited to read a short academic paper about Penney's Game, a well-known probability puzzle. Two optional reading resources are available: a Research Summary and a Full Research Paper. At some point, you pick a three-coin sequence (e.g. HTH). The bot has already chosen its counter-sequence before you finalize your choice. It beats you more often than not.

The performance is layered: the paper you are reading is itself the trick. It is a mathematical piece of misdirection. As you absorb each section, you believe you are gaining an edge. In reality, the bot is using your reading choices as a probabilistic signal to narrow its prediction of what you will choose and then applying Conway's algorithm to guarantee a structural advantage once it knows your sequence. The trick is not that the bot always wins; it is that the bot (probably) knew what you were going to pick before you did, using only two pieces of observable data: what you chose to read, and the first letter you typed. Inspired by magicians in the Now You See Me Trilogy and their tricks within a trick, I built this probability project to not only explain to you Penney's game but test an intersection of psychology + UX + Information Theory.

2. Penney's Game: The Structural Foundation

Penney's Game, introduced by Walter Penney in 1969, is a non-transitive coin-flipping game. Two players each choose a distinct sequence of three coin outcomes (H or T). A fair coin is flipped repeatedly, and the first player whose chosen sequence appears as three consecutive flips wins. What makes the game remarkable is that the win-probability relation is non-transitive: for every sequence a human might choose, there exists a counter-sequence that wins more than half the time.

Conway's algorithm constructs the optimal counter for any chosen sequence XYZ as follows:

```
counter(XYZ)  =  flip(Y) + X + Y
```

For example, if you choose HTH, the counter is T + H + T = THT. Empirical simulation confirms this counter wins approximately 75% of games. The bot never needs to reveal its counter; it simply plays it after you commit.

The win probability is not uniform across all matchups; it ranges from 66.7% to 87.5% depending on the structure of the chosen sequence. These probabilities are derived rigorously using the Law of Total Probability applied state-by-state across a Markov chain, where each state encodes the longest suffix of the flip stream that is a prefix of either sequence. Three structurally distinct cases emerge (corresponding to win rates of 87.5%, 75%, and 66.7%), driven by the depth and reachability of death states: positions from which one player wins with certainty. (I also fully derived how Penney's game works with LOTP, but it's too long, so see Appendix B)

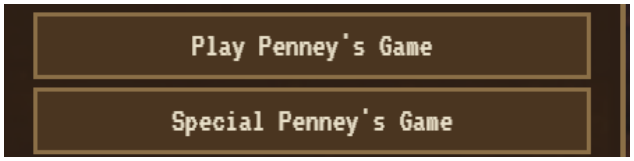
3. Modeling the Human: A Biased Prior

Alongside the standard Penney's Game, there is a twist in the special Penney's Game, where the bot uses information theory to gain its edge from as little as a single coin flip outcome of the human's chosen sequence. The bot's advantage deepens before the game even begins when playing the special Penney game. Humans are systematically bad at generating random sequences. Wagenaar (1972) and Lopes (1982) independently documented that when asked to behave randomly, humans alternate coin outcomes approximately 60% of the time.

Let $q = 0.6$ be the probability of alternation (H → T or T → H), and $p = 0.4$ be the probability of repetition (H → H or T → T). For any three-symbol sequence, the probability that a human selects it can be derived from this single parameter:

$$P(\text{human picks } s_1s_2s_3) \propto 0.5 \times (q \text{ if } s_1 \neq s_2 \text{ else } p) \times (q \text{ if } s_2 \neq s_3 \text{ else } p)$$

Normalizing across all eight sequences yields the human prior distribution H. Sequences like HTH and THT (due to alternation) are significantly overrepresented compared to HHH or TTT. The bot uses this distribution as its starting prior over what sequence you will pick.



4. Information Theory: Collapsing Uncertainty via Content Choice

4.1 Entropy as a Measure of Bot Uncertainty

Shannon entropy measures how uncertain the bot is about your choice:

$$H(X) = -\Sigma\, p(x) \times \log_2 p(x)$$

Before you do anything, the bot faces the full human prior H over eight sequences. This gives an entropy of roughly 2.75 bits — close to, but lower than, the theoretical maximum of 3 bits (uniform over 8 options). The bias toward alternating sequences compresses the distribution, so the bot is already somewhat informed. (I did make an assumption when coding this, see part 4.3)

4.2 The Content-Read Signal

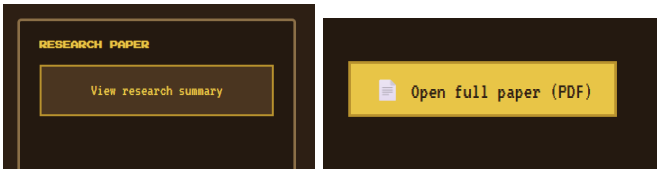
The demo offers the user two optional pieces of reading: a Research Summary and the Full Research Paper. These are presented as resources to help you make a better choice — and they genuinely are. But they also serve as the bot's only observable signal about your epistemic state.

If you read neither, the bot maintains the full prior over all eight sequences (pool size 8, entropy ≈2.75 bits). If you open the Research Summary, it hints that four sequences are strategically sound — HHT, HTT, THH, and TTH — and the bot collapses its candidate pool to these four (pool size 4, entropy = 2.00 bits). If you go further and read the Full Research Paper, which reveals the two best sequences, the pool collapses to just HTH and THT (pool size 2, entropy = 1.00 bit).

At each stage, entropy drops:

$$H_1 \approx 2.75 \text{ bits} \rightarrow H_2 = 2.00 \text{ bits} \rightarrow H_3 = 1.00 \text{ bit}$$

The content the user chose to read is a discrete, unambiguous signal. Unlike a passive scroll tracker, this is an active decision: clicking to open the summary or the full paper. The bot observes which documents were opened, not how much of them was read.



4.3 The Perfect Curiosity Assumption

The model assumes that when a user opens a piece of content, they read it, understood it, and updated their behavior accordingly. This is called a deterministic read model: the content-open event is treated as a hard signal of epistemic state. This is not a psychological claim about all humans: it is a claim about what the bot can know given the only observable it has.

We justify it as a minimax argument. If the human read the summary and the bot assumes they did not, the bot underperforms by using too broad a pool. If the human opened but did not fully absorb the content, the bot still plays a reasonable counter; the expected regret is asymmetric, and the assumption is the safer one to optimize against.

The assumption breaks in two directions: false positive signals (opening content without processing it) and partial updates (reading but not fully acting on it). But since content-open events are the only observable, collapsing the pool on that signal is the maximum information extraction available to the bot.

5. One Letter Is Enough

After the bot has observed which content you chose to read, a single additional signal eliminates almost all remaining uncertainty: the first letter you type.

Given a collapsed pool of k candidates, typing H or T immediately restricts the viable sequences to those beginning with that letter. In the most collapsed state (pool = 2, sequences {HTH, THT}), typing H reduces the candidate set to {HTH}. Now, the bot knows your sequence with certainty before you have finished typing. At that point, it counters with THT.

Even in earlier states, conditional entropy after observing the first letter is substantially lower than the prior entropy. The bot does not need to wait for you to submit your full sequence; it can commit to a counter after receiving two data points: what you chose to read, and the first letter you typed.

```
H(X | pool=2, first='H')  =  0 bits
```

This is the mechanic that makes the trick feel impossible. The illusion is that you chose freely, at the end of the process. Now the bot basically can just guess the optimal counter more often than not, just by knowing the first toss!

6. The Full System

The complete pipeline is as follows. (1) The user is offered two optional readings: a Research Summary and the Full Research Paper. (2) The bot observes which documents the user opens and collapses its candidate pool accordingly, reducing entropy at each stage. (3) The user begins typing their sequence. (4) On the first letter, the bot's conditional entropy drops to near zero. (5) The bot applies Conway's counter and plays the game. (6) The structural win probability of Conway's counter ranges from 66.7% to 87.5% depending on the human's choice.

The system combines three distinct CS109 concepts: a Bayesian prior over human behavior (derived from empirical alternation bias), information-theoretic entropy reduction via a sequential observation model, and Markov chain analysis of the non-transitive game itself. None of these components alone would be sufficient. The alternation bias gives us a prior. The content-read model gives us likelihood updates. Conway's algorithm gives us the exploitable game structure.

The result is a trick within a trick. You think you are learning about probability. You are also being modeled by it. See Appendix A for proof of concept.

I hope you had fun. I know I did. This is super exciting stuff for me, getting so much information from virtually so little (in the user's perspective) because of UX, psychology, and a little misdirection :)

Image Source: Now You See Me: Now You Don't

Implementation Notes

The core probability logic was implemented in Python (ProofOfConcept.py) and validated before connecting to the frontend. The interactive web demo was built in HTML/CSS/JavaScript with assistance from GPT-4 for the frontend scaffolding. The visual bar scene was created in Canva. The alternation bias parameter q = 0.6 is drawn from Wagenaar (1972); the entropy calculations, Conway counter logic, and content-read-to-pool-collapse mapping were implemented independently. GPT-4 was used as a coding assistant for the UI, not for the probability design.

References

Penney, W. (1969). Problem 95: Pentecoinage. Journal of Recreational Mathematics, 2(4), 241.

Wagenaar, W. A. (1972). Generation of random sequences by human subjects. Psychological Bulletin, 77(1), 65–72.

Lopes, L. L. (1982). Doing the impossible: A note on induction and the experience of randomness. Journal of Experimental Psychology, 8(6), 626–636.

Appendix A: Proof Of Concept Python Code

The core logic and workings behind the site (feel free to run this in your Python IDE):

```
import random
import math

# ═══════════════════════════════════════════════════════════════
#  Penney's Game — CS109 Probability Concepts
#  (proof of concept — not connected to frontend)
# ═══════════════════════════════════════════════════════════════


# ── 1. EMPIRICAL HUMAN DISTRIBUTION ──────────────────────────
# From Wagenaar (1972) / Lopes (1982):
# When asked to "act random", humans alternate ~60% of the time.
# We derive P(human picks each pattern) from this one parameter.

q = 0.6   # P(next flip alternates) — empirically measured
p = 0.4   # P(next flip repeats)

PATTERNS = ['HHH','HHT','HTH','HTT','THH','THT','TTH','TTT']

def human_pattern_prob(pattern):
    """P(human chooses this pattern) derived from alternation bias."""
    prob = 0.5  # first flip unbiased
    for i in range(len(pattern) - 1):
        prob *= q if pattern[i] != pattern[i+1] else p
    return prob

# Normalize to sum to 1
raw   = {pat: human_pattern_prob(pat) for pat in PATTERNS}
total = sum(raw.values())
HUMAN_DIST = {pat: raw[pat] / total for pat in PATTERNS}


# ── 2. INFORMATION THEORY — NARROWING BY CONTENT READ STATE ──────────
# As the player reads more of the paper, the bot's uncertainty drops.
```

```
# pool_size: 8 (naive) → 4 (read rational section) → 2 (read best two)

def narrow_options(pool_size, first_letter):
    """Return remaining candidate sequences given what player has read."""
    if pool_size == 8:
        pool = PATTERNS                          # all 8, weighted by HUMAN_DIST
    elif pool_size == 4:
        pool = ['HHT','HTT','THH','TTH']            # 4 rational (non-dominated)
options
    elif pool_size == 2:
        pool = ['HTH','THT']                     # the 2 best sequences for human

    return [s for s in pool if s[0] == first_letter]



# ── 3. ENTROPY ──────────────────────────────────────────────────
# Shannon entropy: H(X) = -∑ p(x) * log2(p(x))
# = expected surprise / uncertainty over the distribution
# Higher H = bot is more uncertain about what you'll pick

def entropy(dist):
    """H(X) = -sum( p * log2(p) ) over all patterns with p > 0."""
    return -sum(p * math.log2(p) for p in dist.values() if p > 0)

# Precompute entropy for each scroll state
STATE_DISTS = {
    'naive (8 options)':         HUMAN_DIST,
    'read paper (4 options)':    {p: (0.25 if p in ['HHT','HTT','THH','TTH'] else 0)
for p in PATTERNS},
    'read best two (2 options)': {p: (0.5  if p in ['HTH','THT']             else 0)
for p in PATTERNS},
}



# ── 4. PENNEY COUNTER ──────────────────────────────────────────
# Conway's rule: for sequence XYZ → counter = (not Y)(X)(Y)

def get_counter(seq):
```

```
    flip = {'H': 'T', 'T': 'H'}
    return flip[seq[1]] + seq[0] + seq[1]


# ── 4. SIMULATE ONE GAME ──────────────────────────────────────
# Flip coins until someone's 3-pattern appears. Return who won.

def sim_game(user_seq, comp_seq):
    flips = []
    while True:
        flips.append(random.choice(['H', 'T']))
        last3 = ''.join(flips[-3:])
        if last3 == user_seq: return 'user'
        if last3 == comp_seq: return 'comp'


# ── DEMO ──────────────────────────────────────────────────────
if __name__ == '__main__':

    print("── Human pick probabilities (from q=0.6 alternation bias) ──")
    for pat, prob in HUMAN_DIST.items():
        print(f"  {pat}: {prob:.1%}")

    seq = input("\nEnter your 3-coin sequence (e.g. HTH): ").upper()

    counter = get_counter(seq)
    print(f"\nYour sequence : {seq}")
    print(f"Bot's counter : {counter}")

    winner = sim_game(seq, counter)
    print(f"Simulated 1 game → {winner} wins")

    print("\n── Entropy per scroll state (bot's uncertainty about your pick) ──")
    for state_name, dist in STATE_DISTS.items():
        h = entropy(dist)
        print(f"  {state_name:30s}  H(X) = {h:.4f} bits")
```

```
first = seq[0]
print(f"\n── After you type first letter '{first}' — conditional entropy ──")
for size in [8, 4, 2]:
    remaining = narrow_options(size, first)
    # uniform over remaining candidates after observing first letter
    n = len(remaining)
    cond_dist = {pat: (1/n if pat in remaining else 0) for pat in PATTERNS}
    h = entropy(cond_dist)
    print(f"  Pool={size} → {remaining}  H = {h:.4f} bits")
```

Appendix B: Deriving Penney's Game With LOTP

This is long, but it was a way for me to wrap my head around the concept! [Used GPT to generate LaTex Equations and summarize this for my decoy write-up paper on the site!]

https://drive.google.com/file/d/1AMJws65-vbv7PHrmzQKxeACit7k3QBDh/view?usp=sharing