

Game 2: Guess the Difficulty

Game Concept

This game challenges the player's **intuition** about word difficulty. Given a word, the player estimates **what percentage of all Wordle solutions it's harder than**. The real answer is calculated based on the word's **ODS (Overall Difficulty Score)** percentile across the dataset.

Files and Data Used

[final_difficulty_scores.csv](#)

This CSV has difficulty metrics for all words:

1	Word	LFS	PLFS	RLP	HLT	WSA	OS	ODS	ExpectedGuesses
2	ROSSA	0.187	0.548	1	0.287	0.238	0.68	0.46	4.11
3	JETTY	0.659	0.575	1	0.691	0.376	0.63	0.656	4.796
4	WIZZO	0.857	0.77	1	0.99	0.834	1	0.903	5.66
5	CUPPA	0.674	0.642	1	0.769	0.374	0.694	0.69	4.915
6	COHOE	0.508	0.537	1	0.794	0.323	1	0.675	4.863
7	GURKS	0.627	0.374	0	0.834	0.363	1	0.542	4.397
8	SQUAD	0.571	0.545	0	0.945	0.551	0.114	0.466	4.131
9	BEISA	0.307	0.483	0	0.307	0.163	1	0.37	3.795
10	SHRUG	0.619	0.511	0	0.882	0.232	0.635	0.494	4.229
11	FOSSA	0.304	0.566	1	0.333	0.589	0.689	0.553	4.436
12	FLUYT	0.779	0.709	0	0.969	0.367	1	0.652	4.782
13	CAMUS	0.511	0.36	0	0.525	0.285	0.667	0.403	3.91
14	SPEED	0.331	0.396	1	0.325	0.586	0.012	0.431	4.008
15	MAMIL	0.575	0.494	1	0.355	0.556	1	0.655	4.792
16	ARRAY	0.375	0.448	1	0.38	0.252	0.04	0.412	3.942
17	POLIO	0.545	0.437	1	0.253	0.704	0.574	0.581	4.534
18	BARNS	0.429	0.213	0	0.417	0.389	0.582	0.347	3.714
19	PANES	0.317	0.101	0	0.032	0.353	0.664	0.252	3.382
20	SOUTS	0.362	0.228	1	0.257	0.389	1	0.522	4.327
21	LIMAS	0.411	0.417	0	0.246	0.534	1	0.432	4.012
22	FETCH	0.692	0.621	0	0.547	0.249	0.345	0.437	4.03
23	QUECK	0.763	0.69	0	0.715	0.342	1	0.603	4.61
24	TWINK	0.766	0.626	0	0.564	0.272	0.429	0.475	4.162
25	GRAZE	0.524	0.55	0	0.903	0.216	0.672	0.482	4.187
26	CROCK	0.708	0.595	1	0.618	0.335	0.614	0.651	4.778

We load this into a `word_data` dictionary:

```
df = pd.read_csv(DATA_PATH)
df['Word'] = df['Word'].str.lower()

with open(WORDS_PATH, 'r') as f:
    text = f.read().lower()
    actual_word_list = re.findall(r"\b[a-z]{5}\b", text)

word_data = {
    row['Word']: {
        "LFS": row["LFS"],
        "PLFS": row["PLFS"],
        "RLP": row["RLP"],
        "HLT": row["HLT"],
        "WSA": row["WSA"],
        "OS": row["OS"],
        "ODS": row["ODS"],
        "ExpectedGuesses": row["ExpectedGuesses"]
    }
    for _, row in df.iterrows()
}
```

We also compute the distribution of difficulty scores across **actual Wordle solutions** to calculate percentiles:

```
actual_ods_scores = [
    word_data[w]["ODS"] for w in actual_word_list if w in word_data
]
```

Game Flow:

1. Randomly pick a word from the Wordle solutions.
2. Display:
"What % of words do you think 'TRUCK' is harder than?"
3. User enters a value between 0–100.
4. Compare their input to the actual **ODS percentile** of the word.
5. If the guess is within $\pm 10\%$, it's **correct**.
6. Show the actual answer and update the score.

GUI Components (Tkinter)

We structure the tab like this:

```
guess_tab = tk.Frame(notebook, bg="#1e272e")
notebook.add(guess_tab, text="🎯 Guess Difficulty")
```



Displayed Labels and Entry

```
word_display_label = tk.Label(guess_tab, ...)
guess_entry = tk.Entry(guess_tab, ...)
game_result_label = tk.Label(guess_tab, ...)
score_label = tk.Label(guess_tab, ...)
```

Game Logic Code

1) Launching a New Game

```
def launch_game():
    current_game_word[0] = choice(actual_word_list)
    word_display_label.config(
        text=f"What % of words do you think '{current_game_word[0]}' is harder than?"
    )
    guess_entry.delete(0, tk.END)
    game_result_label.config(text="")
```

We store the selected word in `current_game_word[0]`

This variable is used in `submit_guess()` to calculate the result

2 Submitting the Guess

```
def submit_guess():
    if current_game_word[0] is None:
        return
    guess_text = guess_entry.get().strip()
    if not guess_text.isdigit() or not (0 <= int(guess_text) <= 100):
        messagebox.showwarning("Invalid", "Enter a number between 0 and 100.")
        return
    guess = int(guess_text)
    word = current_game_word[0]
    current_game_word[0] = None
    if word not in word_data:
        game_result_label.config(text="Word not found in dataset.")
        return
    actual_percentile = percentileofscore(actual_ods_scores, word_data[word]["ods"])
    diff = abs(actual_percentile - guess)
    score["total"] += 1
    if diff <= 10:
        score["correct"] += 1
        game_result_label.config(text=f"✅ Correct! Actual: {actual_percentile:.1f}%", fg="🟢#2ecc71")
    else:
        game_result_label.config(text=f"❌ Off by {diff:.1f}%. Actual: {actual_percentile:.1f}%", fg="🔴#e74c3c")
    score_label.config(text=f"Score: {score['correct']} / {score['total']} ({100*score['correct']/score['total']:.1f}%)")
    root.after(3000, launch_game)
```

We use `scipy.stats.percentileofscore()` to compute what % of words the selected word is harder than, have a $\pm 10\%$ tolerance gives some margin for error.

Visual Theme

All fonts and colors are themed to match the main app:

- Background: `#1e272e`
- Fonts: Segoe UI, bold, centered
- Button color: `#2980b9` and `#f39c12` for contrast
- Color-coded feedback: green = ✅, red = ❌

Why It Works

This game blends **data literacy** with **intuition**. The difficulty scores are statistical, but players are using feel, memory, and experience to make their best guess — and learning more about word difficulty over time.

Future Enhancements

- Make the wordle grading system feel more intuitive/satisfactory.



Game 3: Rank by Difficulty

Game Concept

This game tests the player's relative sense of difficulty between multiple words. Instead of estimating a percentile (like in Game 2), the player must arrange four words in ascending order of difficulty, based on their ODS (Overall Difficulty Score).

This format encourages critical comparison and helps users learn which words are deceptively tough or easier than they appear.

Game Flow

- 1) Randomly choose 4 real Wordle words.
- 2) Display them **in alphabetical order** (to avoid giving a hint).
- 3) Let the player click each word once to build an order (they can't unclick).
- 4) After all 4 are clicked:
 - Compare the order to the true ascending ODS order.
 - Show / and correct order.
 - Automatically start a new round after 10 seconds.

Key Variables

```
ranking_words = [ ]      # The 4 words shown
selected_order = [ ]     # Words clicked by user
word_buttons = [ ]       # Tkinter button widgets for words
```

Word Selection Logic

We ensure the 4 random words are valid, exist in `word_data`, and have associated ODS scores.

```
ranking_words = sample([w for w in actual_word_list if w in word_data], 4)
ranking_words.sort() # Display alphabetically first
```

We create a button for each:

```
for w in ranking_words:
    btn = tk.Button(ranking_frame, text=w.upper(), font=("Segoe UI", 14), width=10,
                    command=lambda word=w: move_word(word), bg="#34495e", fg="white")
    btn.pack(pady=4)
    word_buttons.append(btn)

confirm_btn.pack(pady=8)
```

Click Handling: `move_word()`

When the player clicks a word:

```
def move_word(word):
    if word in selected_order:
        return
    lbl = tk.Label(selected_frame, text=word.upper(), font=("Segoe UI", 13), bg="#2d3436", fg="white", width=10)
    lbl.pack(pady=2)
    selected_order.append(word)
    if len(selected_order) == 4:
        confirm_btn.config(command=submit_ranking)
```

Validation Logic: `submit_ranking()`

We compute the correct order by sorting by ODS:

```
correct_order = sorted(ranking_words, key=lambda w: word_data[w]["ODS"])
```

Then compare:

```
else:
    correct_display = " → ".join(w.upper() for w in correct_order)
    ranking_output.config(text=f"❌ Incorrect. Correct: {correct_display}", fg="#e74c3c")
    selected_order.clear()
```

After displaying feedback, we clear:

```
selected_order.clear()
for widget in selected_frame.winfo_children():
    widget.destroy()
root.after(7000, setup_ranking_game)
```

Future Ideas

- Add a streak counter

Show **ODS values** after a round to help players learn

Game 4: Difficulty Duel

Game Concept

This is a 1-vs-1 elimination game. Each round, you are shown two random Wordle solution words side by side. You must choose which word you believe is harder to guess, based on underlying difficulty factors.

It's similar in spirit to Difficulty Ladder, but:

- It's binary instead of ranked.
- It's fast-paced and very intuitive.
- A great intro to comparative difficulty intuition.

Game Flow

- Pick 2 random Wordle words from the list
- Show both as clickable buttons
- User selects which one they believe is harder
- Game checks ODS to determine if they were correct
- Feedback is shown + score is updated
- New duel starts automatically after a short delay

Word Selection Logic

We draw two words from the Wordle list:

```
def generate_duel():
    candidates = sample([w for w in actual_word_list if w in word_data], 2)
    duel_words[0], duel_words[1] = candidates
    duel_result_label.config(text="", fg="white")
    duel_a_btn.config(text=duel_words[0].upper(), state="normal")
    duel_b_btn.config(text=duel_words[1].upper(), state="normal")
```

Each word is shown as a clickable button:

```
duel_a_btn = tk.Button(duel_btn_frame, text="", font=("Segoe UI", 14, "bold"),
                        width=12, bg="#34495e", fg="white", command=lambda: make_duel_guess(duel_words[0]))
duel_a_btn.grid(row=0, column=0, padx=20)

duel_b_btn = tk.Button(duel_btn_frame, text="", font=("Segoe UI", 14, "bold"),
                        width=12, bg="#34495e", fg="white", command=lambda: make_duel_guess(duel_words[1]))
duel_b_btn.grid(row=0, column=1, padx=20)
```

Guess Logic: `make_duel_guess(choice)`

This function handles user selection:

```
def make_duel_guess(choice):
    word_a, word_b = duel_words
    harder = word_a if word_data[word_a]["ODS"] > word_data[word_b]["ODS"] else word_b
    correct = (choice == harder)

    duel_score["total"] += 1
    if correct:
        duel_score["correct"] += 1
        duel_result_label.config(text="✅ Correct!", fg="#2ecc71")
    else:
        duel_result_label.config(text=f"❌ Wrong! Tougher word was: {harder.upper()}", fg="#e74c3c")

    duel_score_label.config(
        text=f"Score: {duel_score['correct']} / {duel_score['total']} ({100*duel_score['correct']/duel_score['total']:.1f}%"
    )
    duel_a_btn.config(state="disabled")
    duel_b_btn.config(state="disabled")
    root.after(2000, generate_duel)
```

After each round:

- Buttons are disabled
- Feedback is displayed
- New duel starts in 2 seconds:

Future Ideas

- Show **ODS values** after selection
- Add **streak tracking**
- Introduce difficulty tiers (Easy vs Hard duels)

Game 5: Odd One Out

Game Concept

You're shown four 5-letter words. Three are real obscure Wordle words (with high OS = Obscurity Score), and one is a **fake** generated word that *looks plausible*, but isn't actually real.

The player must rely on:

- Vibe matching
- Pattern recognition
- Knowledge of obscure words

Data and Fake Word Generation

We pull from:

- `final_difficulty_scores.csv`: for real obscure words with `OS = 1.0`
- A **2-letter n-gram model** trained on these real obscure words to generate realistic-sounding fakes

Step 1: Load obscure words

```
# Prepare real obscure words
real_obscure_words = [w for w in word_data if word_data[w]["OS"] == 1.0]
```

Step 2: Build a 2-letter → next-letter model

```
ngram_counts = defaultdict(list)
for word in real_obscure_words:
    for i in range(len(word)-2):
        prefix = word[i:i+2]
        next_letter = word[i+2]
        ngram_counts[prefix].append(next_letter)
```

This lets us create a fake by starting with 2 letters and then building from context.

Fake Word Generator

```
def generate_fake_word():  
    while True:  
        word = random.choice(real_obscure_words)  
        base = word[:2]  
        result = base  
        while len(result) < 5:  
            options = ngram_counts.get(result[-2:], [])  
            if not options:  
                break  
            result += random.choice(options)  
        if result not in word_data and len(result) == 5:  
            return result.lower()
```

It ensures:

- 1) 5-letter fake
- 2) Not a known word
- 3) Built from real bigram transitions

Round Setup

Each round:

- 1) 3 real obscure words are picked
- 2) 1 fake word is generated
- 3) All 4 are shuffled

```
real_choices = random.sample(real_obscure_words, 3)  
fake = generate_fake_word()  
fakeout_words = real_choices + [fake]  
random.shuffle(fakeout_words)  
fakeout_answer = fake
```

Each word is shown as a button:

```
btn = tk.Button(fakeout_frame, text=w.upper(), font=("Segoe UI", 14),  
                width=10, bg="#34495e", fg="white",  
                command=lambda word=w: handle_fakeout_guess(word))
```

Guess Handling

```
def handle_fakeout_guess(selected):
    global fakeout_answer
    for btn in fakeout_buttons:
        btn.config(state="disabled")

    fakeout_score["total"] += 1
    if selected == fakeout_answer:
        fakeout_score["correct"] += 1
        fakeout_result_label.config(text="✅ Correct! That was the fake.", fg="✅#2ecc71")
    else:
        fakeout_result_label.config(text=f"❌ Nope! The fake was: {fakeout_answer.upper()}", fg="❌#e74c3c")
```

Score Display

```
fakeout_score_label.config(
    text=f"Score: {fakeout_score['correct']} / {fakeout_score['total']} ({100*fakeout_score['correct']/fakeout_score['total']:.1f}%)"
)
```

Why It Works

- 1) The fake words are generated using real patterns, so they feel **believable**
- 2) Obscure words often look strange themselves — the challenge is fun and fair
- 3) Players learn word structure + obscure vocabulary while playing

Future Ideas

Game 6: Wordlocked

Game Concept

You're given a **set of logic constraints**, and your goal is to find **any 5-letter word** that satisfies **all constraints**. It's a twist on logic puzzles and Wordle — like solving a reverse-engineered clue-based word game.

Dataset: `constraint_puzzles.json`

Each puzzle is **pre-generated** and stored in JSON format with:

```
[
  {
    "constraints": [
      "Exactly 1 vowel",
      "Starts with a consonant",
      "Contains the letter H",
      "Contains the letter P",
      "Letter 2 is Y"
    ],
    "solution": "hyper",
    "answers": [
      "hyped",
      "hyper",
      "hypes",
      "hypha",
      "hypos"
    ]
  },
  {
    "constraints": [
      "Exactly 2 vowels",
      "Starts with a vowel",
      "Contains the letter A",
      "Contains the letter S",
      "Letter 2 is X"
    ],
    "solution": "exams",
    "answers": [
      "axels",
      "axils",
      "axles",
      "axons",
      "exams"
    ]
  }
],
```

constraints: Human-readable rules

answers: All valid words satisfying all constraints, using `final_difficulty_scores.csv`

This approach ensures:

- 1) Every puzzle is **pre-verified**
- 2) No runtime guessing logic required
- 3) Only **valid, fair challenges** with enough possible solutions

Constraints: **generate_constraints.py**

```
def is_valid(word, constraints):
    for c in constraints:
        if c == "Starts with a vowel" and word[0] not in VOWELS:
            return False
        elif c == "Starts with a consonant" and word[0] in VOWELS:
            return False
        elif c == "Ends with a vowel" and word[-1] not in VOWELS:
            return False
        elif c == "Ends with a consonant" and word[-1] in VOWELS:
            return False
        elif c == "All letters are unique" and len(set(word)) != 5:
            return False
        elif c == "Contains repeated letters" and len(set(word)) == 5:
            return False
        elif c == "Exactly 1 vowel" and sum(1 for ch in word if ch in VOWELS) != 1:
            return False
        elif c == "Exactly 2 vowels" and sum(1 for ch in word if ch in VOWELS) != 2:
            return False
        elif c == "Exactly 3 vowels" and sum(1 for ch in word if ch in VOWELS) != 3:
            return False
        elif c == "At least 4 vowels" and sum(1 for ch in word if ch in VOWELS) < 4:
            return False
        elif c.startswith("Contains the letter "):
            letter = c[-1].lower()
            if letter not in word:
                return False
        elif c.startswith("Letter"):
            parts = c.split()
            idx = int(parts[1]) - 1
            expected = parts[-1].lower()
            if word[idx] != expected:
                return False
    return True
```

These constraints are created to contain a large variety of questions, and subsequently words.

Puzzle Creation

```
# ----- Puzzle Generation ----- #

puzzles = []

while len(puzzles) < 200:
    constraints = build_constraints()
    candidates = [w for w in word_list if is_valid(w, constraints)]

    if not candidates:
        continue

    valid_under_threshold = [w for w in candidates if word_os.get(w, 1) < 0.6]
    if not valid_under_threshold:
        continue

    best = min(valid_under_threshold, key=lambda w: word_os[w])
    puzzles.append({
        "constraints": constraints,
        "solution": best,
        "answers": sorted(candidates)
    })

print(f"✅ Generated {len(puzzles)} puzzles.")

# Save to JSON
with open("data/constraint_puzzles.json", "w") as f:
    json.dump(puzzles, f, indent=2)
```

We create 200 such prestored puzzles, each with at least one answer below OS 0.6, to make sure we create puzzles within reasonable difficulty ranges.

Timer Logic

```
✓ def start_wordlocked_timer(t=60):
    wordlocked_timer_label.config(text=f"⌚ Time left: {t}s")
    ✓ if t > 0:
        wordlocked_timer_id[0] = root.after(1000, lambda: start_wordlocked_timer(t - 1))
    ✓ else:
        handle_wordlocked_submission(timeout=True)
```


You get 60 seconds to answer each puzzle.

If time runs out, it counts as a miss and **adds 60 seconds** penalty.

Submission Handling

```
def handle_wordlocked_submission(event=None, timeout=False):
    puzzle = wordlocked_puzzles[wordlocked_index[0]]
    accepted_answers = puzzle["answers"]
    solution = min(accepted_answers, key=lambda w: word_data.get(w, {}).get("OS", 1.0))
    user_guess = wordlocked_input.get().strip().lower()
```

If correct:

- 1) Adds the actual time taken
- 2) Feedback shows green  with timestamp

If wrong:

- 1) Adds **60s penalty**
- 2) Shows one correct answer: the lowest OS word (most obscure among answers)

```
wordlocked_score["time_sum"] += 60.0
wordlocked_feedback.config(
    text=f"❌ Wrong! One valid answer: {solution.upper()} (+60s)", fg=❌ "#e74c3c"
```

Round Progression

```
wordlocked_index[0] += 1
if wordlocked_index[0] >= len(wordlocked_puzzles):
    wordlocked_index[0] = 0
    random.shuffle(wordlocked_puzzles)
```

Each puzzle is auto-replaced after 4 seconds (feedback delay).

Constraint Display

```
wordlocked_constraint_label.config(
    text="\n".join(f"• {c}" for c in constraints)
)
```

- Multiple constraints are supported
- Displayed as bullet points
- Ensures readability with wrap and center alignment

Game 7: Word Builder Blitz

Game Concept

- 1) You're given 8 shuffled letters, including at least 2 vowels.
- 2) You must form **valid 5-letter words** using any combination of those letters
- 3) The letters can be reused **as long as frequency constraints are met**
- 4) After 60 seconds, the game shows all valid solutions, refreshes with a new letter pool

Word Source

We use a filtered subset of `final_difficulty_scores.csv`:

```
builder_word_set = set(df[df["os"] < 0.8]["Word"].tolist())
```

This ensures we only include words with OS values under 0.8 to avoid overly obscure/rare solutions.

Letter Pool Generation

```
def generate_letter_pool():
    vowels = 'aeiou'
    all_letters = 'abcdefghijklmnopqrstuvwxyz'
    pool = sample(vowels, 2) + sample(all_letters, 6)
    return sample(pool, 8)
```

- 1) **2 vowels** are guaranteed
- 2) Random consonants are added
- 3) We shuffle the full pool to prevent predictability

Valid Word Filtering

We then filter all eligible 5-letter words based on this pool:

```
def find_valid_words():
    valid = []
    for word in builder_word_set:
        if len(word) == 5 and all(word.count(c) <= builder_letters.count(c) for c in set(word)):
            valid.append(word)
    return valid
```

It checks that each character in the word doesn't exceed what's available in the pool
All valid 5-letter answers are stored in `current_valid_words`.

Why It Works

- 1) Encourages fast thinking under pressure
- 2) Word frequency filtering keeps it accessible
- 3) Clear and colorful UI makes it engaging
- 4) Automatic feedback loop keeps the player focused

Future Upgrades

- 1) Difficulty tiers (OS buckets: 0.2–0.5, 0.5–0.8)
- 2) Multiplayer blitz mode with real-time score updates
- 3) Bonus points for rare words
- 4) Combo chains for back-to-back valid guesses

Game 8: Word Morph

Game Concept

- 1) You are given a **START word** and an **END word**.
- 2) Your task is to **fill in the intermediate steps**, where each step changes exactly **one letter, every word in the chain must be a valid Wordle word**
- 3) All puzzles are **pre-solved** and **preloaded** from a JSON file to ensure fairness and solvability.

We create the JSON file with the puzzles with the following code:

```
filtered_words = set(df[df["os"] < 0.5]["Word"])
```

We make sure all the words used to solve the puzzle have an os of less than 0.5, to ensure that unfairly obscure words don't tamper with the puzzles

Building the Word Morph Graph

```
def build_graph(words):
    graph = defaultdict(set)
    buckets = defaultdict(list)
    for word in words:
        for i in range(5):
            pattern = word[:i] + "_" + word[i+1:]
            buckets[pattern].append(word)
    for group in buckets.values():
        for i in range(len(group)):
            for j in range(i+1, len(group)):
                w1, w2 = group[i], group[j]
                graph[w1].add(w2)
                graph[w2].add(w1)
    return graph
```

To allow you to **efficiently look up valid next-step words**, you build a graph where:

- 1) **Nodes** = 5-letter words.
- 2) **Edges** = connect two words that differ by exactly **one letter**.

Finding All Shortest Paths Between Word Pairs

```
# ----- BFS to get all shortest paths ----- #
def all_shortest_paths(start, end, graph):
    queue = deque([[start]])
    visited = {start: 0}
    results = []
    shortest_length = None

    while queue:
        path = queue.popleft()
        current = path[-1]

        if shortest_length and len(path) > shortest_length:
            continue

        if current == end:
            if shortest_length is None:
                shortest_length = len(path)
            results.append(path)
            continue

        for neighbor in graph[current]:
            if neighbor not in visited or visited[neighbor] >= len(path):
                visited[neighbor] = len(path)
                queue.append(path + [neighbor])

    return results
```

- 1) Uses **BFS** (Breadth-First Search) to find all shortest paths.
- 2) Keeps track of all paths equal to the minimum length.
- 3) Ensures no duplicate visits unless they're part of a path of equal or better length.

Generating Puzzle Pairs

```
# ----- Generate diverse start-end pairs ----- #
def generate_morph_pairs(words, graph, count=100, min_len=4, max_len=10):
    words_list = list(words)
    random.shuffle(words_list)
    pairs = []
    seen = set()
    attempts = 0

    while len(pairs) < count and attempts < 5000:
        w1, w2 = random.sample(words_list, 2)
        key = tuple(sorted((w1, w2)))
        if key in seen:
            attempts += 1
            continue
        seen.add(key)
```

- 1) Randomly sample word pairs.
- 2) Avoid repeats using a **seen** set.
- 3) Only keep pairs with a **shortest path length between 4 and 10**, so puzzles aren't too trivial or long.
- 4) Store **all shortest paths** for each valid pair.

Validation Logic

```
def check_morph_attempt():
    guess_path = [current_morph["start"]] + [e.get().strip().lower() for e in morph_entries] + [current_morph["end"]]
    valid_paths = [path for path in current_morph["paths"]]

    if guess_path in valid_paths:
        result = "✅ Correct! You solved the morph chain!"
        color = "#00b894"
    else:
        sample = " → ".join(word.upper() for word in valid_paths[0])
        result = f"❌ Incorrect. Try again or reveal one valid path:\n{sample}"
        color = "#d63031"
```

- 1) Player inputs are compared word-by-word against the known path.
- 2) Case-insensitive comparison.
- 3) A solution reveal option is offered.

Future Ideas

- 1) Add difficulty levels (based on number of steps or word obscurity)
- 2) Timed mode
- 3) Leaderboards for quickest morphs
- 4) Daily Morph Puzzle

Game 9: Reverse Wordle

Game Concept

You're given a sequence of 4 guesses with colored feedback — like seeing someone else's Wordle board. Your job? **Guess the final solution word**, using only the past feedback as clues. You get **30 seconds per puzzle**, and **only one answer** is valid.

Core Design Goals

- 1) Replicate the look and logic of a Wordle grid (guess + feedback)
- 2) Use pre-solved puzzles for guaranteed unique answers
- 3) Ensure all solutions are relatively **fair and guessable** (OS < 0.5)
- 4) Provide a **countdown timer** and track correct/incorrect attempts

We create the puzzles through `generate_puzzles.py`

Simulate Wordle Feedback

```
# Wordle feedback simulation
def get_feedback(solution, guess):
    feedback = ['B'] * 5
    s_count = Counter(solution)
    for i in range(5):
        if guess[i] == solution[i]:
            feedback[i] = 'G'
            s_count[guess[i]] -= 1
    for i in range(5):
        if feedback[i] == 'B' and s_count[guess[i]] > 0:
            feedback[i] = 'Y'
            s_count[guess[i]] -= 1
    return ''.join(feedback)
```

- 1) Mimics the **Wordle logic**
- 2) Handles **duplicate letter logic** using a counter for accuracy.

This is the **engine that powers the game logic** — both for generating and verifying puzzles.

Create Puzzle Data

```
STARTERS = ["slate", "crane", "stare", "arise", "trace", "reast", "glint", "blame", "pride"]

puzzles = []
random.shuffle(candidate_solutions)
```

The script tries to solve each word by simulating **four realistic Wordle guesses**.

Picks guesses from a mix of the **STARTERS** list (highly strategic first guesses), **actual_words** (so it feels natural and human-like).

Each guess is tracked as a **[word, feedback]** pair.

Ensure Unique Solution

```
# Check if it's a uniquely solvable path
compatible = [w for w in dictionary if all(matches(w, g, f) for g, f in guesses)]
if len(compatible) == 1:
    puzzles.append({"solution": target, "guesses": guesses})
```

This is the **core validation** step:

We simulate every word in the dictionary to see which words would match **all the guess-feedback pairs**.

If **only one word** fits the entire sequence, it's considered a valid and solvable puzzle.

Rendering the Puzzle Grid

```
for guess, feedback in puzzle['guesses'][:-1]:
    row = tk.Frame(grid_frame, bg="█" * "#1e272e")
    row.pack(pady=2)
    for i, letter in enumerate(guess):
        bg = feedback_colors.get(feedback[i], "█" * "#bdc3c7")
        tk.Label(row, text=letter.upper(), font=("Courier", 18, "bold"),
                 bg=bg, fg="white", width=2, height=1).pack(side="left", padx=2)
```

- 1) Each **guess** is a 5-letter string
- 2) Each **feedback** string (e.g. "BYBBY") is applied letter by letter
- 3) Each letter appears in its own colored **tk.Label**

Timer & Countdown

```
def countdown(t=30):
    countdown_label.config(text=f"⌚ {t} seconds")
    if t > 0:
        timer_id[0] = root.after(1000, lambda: countdown(t - 1))
    else:
        submit_btn.config(state="disabled")
        time_expired()
```

- 1) Uses `root.after()` for scheduling updates every second
- 2) Cancels old timers when new rounds are launched
- 3) If timer hits 0, it automatically submits an incorrect guess

Puzzle Refresh

```
grid_index[0] += 1
if grid_index[0] >= len(grid_puzzles):
    grid_index[0] = 0
    random.shuffle(grid_puzzles)
root.after(2500, render_grid_puzzle)
```

Ensures continuous gameplay with reshuffled puzzles after a full loop.