# Project Report: Wordle Game and Automated Solver

Team Flan
CS Final Project

December 19, 2025

**Abstract**

This report documents the design and implementation of a text-based clone of the game Wordle, developed in C. In addition to the playable game, the project includes a statistical solver capable of guessing the target word autonomously. The project focuses on modular design, string manipulation, and entropy-based search algorithms.

## Contents

# 1 Introduction

The goal of this project was to recreate the mechanics of the viral word game "Wordle" within a Unix-style command-line environment. The project consists of three main components: a shared logic library, a user interface, and an AI solver.

# 2 System Architecture

We organized the code into modular files to separate the backend logic from the frontend presentation.

- `wordle.c`: This file acts as the "engine." It handles dictionary loading, memory management, and the core rules of the game .
- `wordle_ui.c`: This is the playable version of the game. It imports `wordle.c` and adds a visual layer using ANSI escape codes for coloring .
- `solver.c`: An algorithmic solver that uses the logic from `wordle.c` to simulate games and determine optimal guesses .

# 3 Core Logic Implementation

## 3.1 Data Structures

To manage the state of the game, we avoided complex objects and stuck to simple arrays and structs. We defined a struct `attempt_response` to hold the feedback for each letter:

```
typedef struct {
  char letter;
  int state;
} attempt_response;
```

The `state` integer corresponds to an enum representing `NUH` (Gray), `GOOD` (Green), or `EXISTS` (Yellow) .

## 3.2 Dictionary Management

The dictionary is loaded from a text file named `dictionary.txt` . We set a hard limit of 6,969 words for the dictionary size . The `init_wordle()` function handles the file I/O using `fscanf` and randomly selects a target word using `srand(time(0))` .

## 3.3 The Feedback Algorithm

The most challenging logic in Wordle is handling duplicate letters correctly (e.g., if the target is "APPLE" and the guess is "PUPPY"). A naive check would mark all P's as yellow, which is incorrect.

Our solution, implemented in the `feedback` function, uses a two-pass approach:

1. We count the frequency of each letter in both the target and the guess .

2. If a letter appears more times in the guess than in the target, we mark the excess instances as 'ignored' (or gray) starting from the right side of the word .

3. Finally, we iterate through the cleaned guess to assign the `GOOD`, `EXISTS`, or `NUH` states .

# 4 User Interface

The UI runs in the terminal. To make the game visually distinct, we used ANSI escape sequences.

- **Coloring:** Macros like `GREEN ("\x1b[42m")` and `YELLOW ("\x1b[43m")` change the background color of the text during the print phase .
- **Screen Refreshing:** To prevent the terminal from getting cluttered, we used `"\033[A\033[K"` to move the cursor up and clear lines, creating a smoother frame update when the user enters an invalid word .

# 5 The Automated Solver

The highlight of the project is `solver.c`. It does not cheat by peeking at the target; instead, it uses information theory to narrow down the possibilities.

## 5.1 Strategy: Entropy Reduction

The solver starts with the full pool of words (approx. 7,000). For every potential guess, it calculates a score based on how well that guess splits the remaining pool.

- The function `entropy_score` simulates the outcome of a guess against every other valid word .
- It groups the results into pattern codes (using base-3 math for the 5 slots) .
- It sums the squares of these group sizes. A lower score implies the guess distributes the remaining candidates more evenly, providing more information .

## 5.2 Performance Optimization

Calculating entropy for 7,000 words against 7,000 words is computationally expensive ($O(n^2)$). To speed up the first turn, we hardcoded the starting word to "brain" . We found this word provides a statistically strong opening without waiting for the CPU to calculate it every time.

## 5.3 Filtering

After making a guess and receiving feedback (Green/Yellow/Gray), the solver calls `filter_pool`. This function iterates through the current candidate list and removes any word that wouldn't have produced that specific feedback pattern . This rapidly reduces the pool size, often going from thousands to single digits in 3 turns.