

Vibe Coding Reports

Vibe Coding Report 1: Word Frequency Analyzer

A. Prompts

Prompt 1 (Gemini 3 Pro)

I am going to do vibe coding and I want you to help me write the first prompt. The problem is writing a python code that does:

Problem 1: Word Frequency Analyzer Write a program that takes in a block of text and outputs the 10 most frequent words, along with their counts. Ignore capitalization and punctuation. If two words have the same count, break ties by alphabetical order of the word. Return exactly k items if there are at least k unique words

The input format examples are: * **Input 1:** "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the fox was." * **Input 2:** "Data, data, data! I can't make bricks without clay, said Sherlock. DATA drives decisions." * **Input 3:** "To be, or not to be--that is the question: Whether 'tis nobler in the mind to suffer..."

Prompt 2 (Cursor)

Role & Objective: Act as an expert Python developer. Write a highly optimized, clean, and well-documented Python function that acts as a Word Frequency Analyzer.

Function Signature: `def analyze_word_frequency(text: str, k: int = 10) -> list[tuple[str, int]]:`

Core Requirements: * **Parse the Text:** Process the input block of text and count the frequency of each word. * **Normalization:** Ignore capitalization (treat "The" and "the" as identical). *

Punctuation: Remove all punctuation. (Note: Please handle contractions like "wasn't" or "can't" by keeping them as a single word without the apostrophe, e.g., "wasnt", OR specify if you prefer splitting them). * **Sorting & Tie-Breaking:** Sort the output descending by frequency. If two or more words have the exact same count, break the tie by sorting those words in ascending alphabetical order. * **Output Limit:** Return exactly k items as a list of tuples (word, count). If the text contains fewer than k unique words, return all available unique words sorted by the same rules.

Test Cases to validate against: * Input 1: "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the fox was." * Input 2: "Data, data, data! I can't make bricks without clay, said Sherlock. DATA drives decisions." * Input 3: "To be, or not to be--that is the question: Whether 'tis nobler in the mind to suffer..."

Please provide the complete Python code, including the collections or re modules if you use them, and write a quick test block at the bottom to run the three inputs.

Prompt 3 (Gemini 3 Pro)

The cursor has generated a python code. Now write a prompt that orders it to write a test code: For each test case provided , it should clearly list the input you used and the exact output the program produced.

the inputs are: * Input 1: "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the fox was." * Input 2: "Data, data, data! I can't make bricks without clay, said Sherlock. DATA drives

Vibe Coding Reports

decisions." * Input 3: "To be, or not to be--that is the question: Whether 'tis nobler in the mind to suffer..."

Prompt 4 (Cursor)

Objective: Write a test script to validate the word frequency function we just created. Do not use a silent testing framework like unittest; instead, write a script that runs the function against specific inputs and prints the results clearly to the console.

Formatting Requirements: For each test case, execute the function with k=10. The terminal output for each test case must be heavily formatted and clearly list both the raw input and the exact output. Use print statements to format it exactly like this example:

```
```text
```

## TEST CASE 1

INPUT: "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the fox was."

OUTPUT: [('the', 4), ('dog', 2), ('fox', 2), ('amused', 1), ('brown', 1), ...] ```

**The Test Inputs:** Please implement this for the following three inputs: \* Input 1: "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the fox was." \* Input 2: "Data, data, data! I can't make bricks without clay, said Sherlock. DATA drives decisions." \* Input 3: "To be, or not to be--that is the question: Whether 'tis nobler in the mind to suffer..."

## Prompt 5 (Gemini 3 Pro)

Summarize the logic of the following program. Then, comment on how the program could be improved. give me a two-paragraph summary.

---

## B. Final Generated Program

### word\_frequency\_analyzer.py (Main Python Code)

```
"""
Word Frequency Analyzer

Parses text, normalizes words (lowercase, no punctuation), counts
frequencies,
and returns the top-k (word, count) pairs sorted by count descending with
alphabetical tie-breaking.
"""

import re
from collections import Counter

def analyze_word_frequency(text: str, k: int = 10) -> list[tuple[str,
int]]:
"""
Analyze word frequency in the given text and return the top-k words.
```

# Vibe Coding Reports

Words are normalized: lowercased and stripped of punctuation. Contractions (e.g., "wasn't", "can't") are kept as single tokens with the apostrophe removed ("wasnt", "cant").

Args:

text: Input string to analyze.

k: Maximum number of (word, count) pairs to return (default 10).

Returns:

List of up to k tuples (word, count), sorted by count descending.

Ties in count are broken by word ascending alphabetically.

Fewer than k items if the text has fewer unique words.

"""

```
if not text or not text.strip():
 return []
```

```
Normalize: lowercase for case-insensitive counting
normalized = text.lower()
```

```
Handle contractions: remove apostrophes so "wasn't" -> "wasnt", "'tis"
-> "tis"
normalized = normalized.replace("'", "")
```

```
Remove all remaining punctuation (keep only letters, digits, spaces)
normalized = re.sub(r"[^a-z0-9\s]", " ", normalized)
```

```
Split on whitespace and drop empty tokens
words = normalized.split()
```

```
if not words:
 return []
```

```
Count occurrences
counts = Counter(words)
```

```
Sort: primary key = count descending, secondary key = word ascending
sorted_items = sorted(counts.items(), key=lambda x: (-x[1], x[0]))
```

```
return sorted_items[:k]
```

#

-----

-

```
Tests
```

#

-----

```
-
```

```
if __name__ == "__main__":
 test_cases = [
 (
 "The quick brown fox jumps over the lazy dog. The dog wasn't amused; the
 fox was.",
 10,
),
 (
 "Data, data, data! I can't make bricks without clay, said Sherlock. DATA
 drives decisions.",

```

## Vibe Coding Reports

```
10,
),
(
"To be, or not to be--that is the question: Whether 'tis nobler in the
mind to suffer...",
10,
)
]

for i, (text, k) in enumerate(test_cases, start=1):
 print(f"--- Input {i} (k={k}) ---")
 result = analyze_word_frequency(text, k)
 for word, count in result:
 print(f" {word!r}: {count}")
 print()
```

### test\_word\_frequency.py

```
"""
Test script for analyze_word_frequency.
Runs the function against three fixed inputs with k=10 and prints
formatted results to the console (no unittest).
"""

from word_frequency_analyzer import analyze_word_frequency

K = 10

INPUT_1 = "The quick brown fox jumps over the lazy dog. The dog wasn't
amused; the fox was."
INPUT_2 = "Data, data, data! I can't make bricks without clay, said
Sherlock. DATA drives decisions."
INPUT_3 = "To be, or not to be--that is the question: Whether 'tis nobler
in the mind to suffer..."

def run_test(case_num: int, text: str) -> None:
 print("=====
 print(f"TEST CASE {case_num}")
 print("=====
 print("INPUT:")
 print(repr(text))
 print()
 result = analyze_word_frequency(text, k=K)
 print("OUTPUT:")
 print(result)
 print()

if __name__ == "__main__":
 run_test(1, INPUT_1)
 run_test(2, INPUT_2)
 run_test(3, INPUT_3)
```

---

## C. Inputs and Outputs

# Vibe Coding Reports

```
=====
TEST CASE 1
=====
INPUT:
"The quick brown fox jumps over the lazy dog. The dog wasn't amused; the
fox was."

OUTPUT:
[('the', 4), ('dog', 2), ('fox', 2), ('amused', 1), ('brown', 1),
('jumps', 1), ('lazy', 1), ('over', 1), ('quick', 1), ('was', 1)]

=====
TEST CASE 2
=====
INPUT:
"Data, data, data! I can't make bricks without clay, said Sherlock. DATA
drives decisions."

OUTPUT:
[('data', 4), ('bricks', 1), ('cant', 1), ('clay', 1), ('decisions', 1),
('drives', 1), ('i', 1), ('make', 1), ('said', 1), ('sherlock', 1)]

=====
TEST CASE 3
=====
INPUT:
"To be, or not to be--that is the question: Whether 'tis nobler in the
mind to suffer..."
```

---

## D. AI-Generated Analysis

The program calculates the top k most frequent words in a string by systematically cleaning, counting, and sorting the text. It begins by converting the entire input to lowercase and manually stripping out apostrophes so that contractions are merged into single alphanumeric tokens (e.g., "wasn't" becomes "wasnt"). Next, it uses a regular expression to replace all remaining non-alphanumeric characters with spaces, which safely removes punctuation and treats characters like em-dashes as word separators. After splitting this sanitized string into a list of words, it tallies their frequencies using collections.Counter. Finally, the script sorts the dictionary items using a custom lambda function--prioritizing highest counts first (-x[1]), while resolving ties alphabetically (x[0])--and returns the top k slice.

While the script is highly effective for basic English text, it could be improved for better scalability and internationalization. The current regular expression [^a-z0-9\s] strips out all non-ASCII characters, meaning accented words (like "café" or "naïve") will be improperly fractured; swapping this for a Unicode-aware regex would make the parser much more robust. Additionally, the current sorting method evaluates the entire vocabulary of the text at once (sorted(counts.items(), ...)), which is inefficient for massive datasets. While Counter(words).most\_common() doesn't natively support alphabetical tie-breaking, implementing a custom

## Vibe Coding Reports

priority queue (using the `heapq` module) to maintain only the top  $k$  elements in memory would significantly improve the algorithm's performance on book-length inputs.

# Vibe Coding Reports

## Vibe Coding Report: Palindrome Checker

### A. Prompts

#### Prompt 1 (Gemini 3 Pro)

I am going to do vibe coding and I want you to help me write the first prompt. The problem is writing a python code that follows the following instruction:

Write a program that takes in a sentence and determines if it is a palindrome, ignoring capitalization, spaces, and punctuation. Example: "A man, a plan, a canal: Panama" -> "This is a palindrome."

The input format examples are:

- **Input 1:** A man, a plan, a canal: Panama!
- **Input 2:** No 'x' in Nixon
- **Input 3:** Was it a car or a cat I saw?

Test cases must be written with these examples and the test output should clearly format and print the input and output.

#### Prompt 2 (Cursor)

Act as an expert Python developer. I need a Python script that checks if a given sentence is a palindrome. Here are the strict requirements for the logic: \* The checker must completely ignore capitalization, spaces, and all punctuation. Only consider alphanumeric characters. \* The output should return a clean, user-friendly string, such as 'This is a palindrome.' or 'This is not a palindrome.'

**Testing & Output Formatting Requirements:** Write a testing block that loops through a list of inputs. You must use exactly these test cases: 1. "A man, a plan, a canal: Panama!" 2. "No 'x' in Nixon" 3. "Was it a car or a cat I saw?"

When running the tests, format the console output so it clearly displays both the original input and the final evaluated output in a highly readable way. Keep the code elegant, Pythonic, and well-commented.

#### Prompt 3 (Gemini 3 Pro)

Summarize the logic of the following program. Then, comment on how the program could be improved. give me a two-paragraph summary.

---

### B. Final Generated Program

```
"""
Palindrome checker: ignores case, spaces, and punctuation; considers only
alphanumeric characters.
Returns user-friendly messages and includes a readable test runner.
"""
```

## Vibe Coding Reports

```
import re

def is_palindrome(s: str) -> bool:
 """
 Return True if the string is a palindrome when considering only
 alphanumeric
 characters (ignoring case, spaces, and punctuation); otherwise False.
 """
 # Keep only letters and digits, then normalize to lowercase for comparison
 cleaned = re.sub(r"[^a-zA-Z0-9]", "", s).lower()
 return cleaned == cleaned[::-1]

def check_palindrome(s: str) -> str:
 """
 Check if the given sentence is a palindrome and return a user-friendly
 message.
 """
 if is_palindrome(s):
 return "This is a palindrome."
 return "This is not a palindrome."

def run_tests():
 """
 Run the required test cases and print original input and result in a
 readable format."""
 test_cases = [
 "A man, a plan, a canal: Panama!",
 "No 'x' in Nixon",
 "Was it a car or a cat I saw?",
]
 width = 60 # For consistent column alignment

 print("=" * (width * 2 + 9))
 print("PALINDROME CHECKER -- Test Results")
 print("=" * (width * 2 + 9))

 for i, sentence in enumerate(test_cases, start=1):
 result = check_palindrome(sentence)
 print(f"\nTest {i}:")
 print(f" Input: {sentence}")
 print(f" Output: {result}")

 print("\n" + "=" * (width * 2 + 9))

if __name__ == "__main__":
 run_tests()
```

## C. Inputs and Outputs

Test 1:  
Input: A man, a plan, a canal: Panama!  
Output: This is a palindrome.

Test 2:

## Vibe Coding Reports

Input: No 'x' in Nixon  
Output: This is a palindrome.

Test 3:

Input: Was it a car or a cat I saw?  
Output: This is a palindrome.

---

### D. AI-Generated Analysis

The provided Python program efficiently determines whether a given sentence is a palindrome by isolating its core alphanumeric characters and comparing the sequence to its reverse. At its core, the script utilizes the `re` (regular expression) module to strip away all spaces, punctuation, and special characters from the input string, subsequently standardizing the remaining characters to lowercase to ensure case-insensitivity. This sanitized string is then checked against a reversed slice of itself (`cleaned[::-1]`); if they match, a wrapper function returns a user-friendly confirmation message. Finally, a test-runner function iterates through a hardcoded list of specific test cases, neatly formatting and printing the original input alongside the evaluated result to the console.

While the script is highly readable and works perfectly for standard sentences, it could be optimized for both performance and extensibility. For instance, using the slicing method (`[::-1]`) creates a full copy of the string in memory, which could cause a bottleneck with exceptionally large text inputs; replacing this with a two-pointer approach--comparing characters from the beginning and end moving inwards--would reduce the space complexity to constant  $O(1)$ . Additionally, the regular expression pattern is recompiled every time the function is called; defining `re.compile(r"[^a-zA-Z0-9]")` globally would yield a slight speed boost. Finally, moving the hardcoded examples into a formal testing framework like `pytest` and allowing dynamic user inputs via the command line would elevate the script from a simple experiment to a robust, production-ready tool.

# Vibe Coding Reports

## Vibe Coding Report: Group Anagrams

### A. Prompts

This time, I took one further step of prompting to improve the code.

#### Prompt 1 (Gemini 3 Pro)

I am going to do vibe coding and I want you to help me write the first prompt. The problem is writing a python code that follows the following instruction:

Given a list of words, group together all words that are anagrams of each other. Example: Input: ["listen", "silent", "enlist", "rat", "tar"] Output: [["listen", "silent", "enlist"], ["rat", "tar"]]

The input format examples are: \* **Input 1:** listen silent enlist inlets google giggle rat tar art \* **Input 2:** dusty study night thing cat act tac tacit attic tactic \* **Input 3:** state taste tates seats asset teats teat set eats east \* (Each line can be interpreted as a space-separated list of words.)

Test cases must be written with these examples and the test output should clearly format and print the input and output. Do not hardcode the test code. Rather, use a formal testing framework like pytest and allowing dynamic user inputs, but use the inputs I provided on that framework.

#### Prompt 2 (Cursor)

Act as an expert Python developer. I want you to write a clean, efficient Python script to solve the "Group Anagrams" problem, along with a robust test suite using pytest.

**The Objective:** Write a function that takes a space-separated string of words, parses it into a list, and groups all words that are anagrams of each other. Example: > Input: "listen silent enlist rat tar" Output: [["listen", "silent", "enlist"], ["rat", "tar"]]

**Testing Requirements:** 1. Use the pytest framework. 2. Do not just hardcode the tests sequentially. Use `@pytest.mark.parametrize` to allow for dynamic, scalable testing. 3. The test cases must clearly print the input string and the resulting grouped output to the console so I can visually verify it (ensure the test uses print statements or logs that can be seen when running `pytest -s`). 4. Include the ability for a user to input their own string via standard input if run as a standalone script (e.g., under `if __name__ == "__main__":`).

**Use the following inputs for your parameterized tests:** \* Test Case 1 Input: listen silent enlist inlets google giggle rat tar art \* Test Case 2 Input: dusty study night thing cat act tac tacit attic tactic \* Test Case 3 Input: state taste tates seats asset teats teat set eats east

Please provide the complete Python code, including the main logic and the pytest suite in a single file or clearly separated blocks, complete with type hinting and docstrings.

#### Prompt 3 (Gemini 3 Pro)

Summarize the logic of the following program. Then, comment on how the program could be improved. give me a two-paragraph summary.

#### Prompt 4 (Cursor)

Refactor the `group_anagrams` Python script to implement the following improvements:

# Vibe Coding Reports

- 1. Algorithmic Optimization:** Update the `group_anagrams` function to use a 26-element character frequency tuple as the dictionary key instead of sorting the characters. This will optimize the time complexity per word from  $O(K \log K)$  to  $O(K)$ . You can use `ord(char) - ord('a')` to map lowercase characters to their respective indices.
- 2. Type Hinting Modernization:** Completely remove the `from typing import List` import. Update all type hints across the entire file--including the function signature, variable annotations, and the pytest suite--to use modern Python built-in types (e.g., `list`, `dict`, `tuple` instead of `List`, `Dict`, `Tuple`).

Please ensure all existing tests still pass, the interactive `__main__` block remains functional, and the docstrings are updated to reflect the new logic.

---

## B. Final Generated Program

```
"""
Group Anagrams: Given a space-separated string of words, group all words
that are anagrams.
Includes pytest suite with parametrized tests and optional stdin input
when run as script.
"""

from __future__ import annotations

from collections import defaultdict

def _char_frequency_key(word: str) -> tuple[int, ...]:
 """
 Build a 26-element tuple of letter counts (a-z) for the lowercase word.

 Uses ord(char) - ord('a') to map each letter to an index. Non-letter
 characters are ignored. O(K) per word where K = len(word).

 Returns:
 A tuple of 26 integers: count of 'a' at index 0, 'b' at 1, ..., 'z' at 25.
 """
 counts = [0] * 26
 for c in word.lower():
 if "a" <= c <= "z":
 counts[ord(c) - ord("a")] += 1
 return tuple(counts)

def group_anagrams(words_str: str) -> list[list[str]]:
 """
 Parse a space-separated string into words and group all anagrams together.

 Two words are anagrams if they have the same character frequencies (same
 count per letter a-z, ignoring case and non-letter characters). Keys are
 computed as 26-element frequency tuples in O(K) per word. Groups are
 returned in order of first occurrence; within each group, words keep
 their input order.

 Args:
 words_str: A single string of space-separated words.
 """

```

# Vibe Coding Reports

Returns:

A list of lists; each inner list contains words that are anagrams of each other. Groups appear in order of first occurrence.

Example:

```
>>> group_anagrams("listen silent enlist rat tar")
[['listen', 'silent', 'enlist'], ['rat', 'tar']]
"""
if not words_str or not words_str.strip():
 return []

words = words_str.split()
Key: 26-element frequency tuple (hashable); value: list of words with
that signature
groups: dict[tuple[int, ...], list[str]] = defaultdict(list)

for word in words:
 key = _char_frequency_key(word)
 groups[key].append(word)

Return groups in order of first occurrence (iteration order of
defaultdict)
return list(groups.values())
```

#

---

-

```
Pytest suite (parametrized, with visible input/output via print)
Run with: pytest group_anagrams.py -v -s
#
```

---

-

```
import pytest
```

```
@pytest.mark.parametrize(
"words_str,expected",
[
(
 "listen silent enlist inlets google giggle rat tar art",
 [
 ["listen", "silent", "enlist", "inlets"],
 ["google"],
 ["giggle"],
 ["rat", "tar", "art"],
],
),
(
 "dusty study night thing cat act tac tacit attic tactic",
 [
 ["dusty", "study"],
 ["night", "thing"],
 ["cat", "act", "tac"],
 ["tacit", "attic"],
 ["tactic"],
],
),
)
```

# Vibe Coding Reports

```
(
"state taste tates seats asset teats teat set eats east",
[
["state"], ["taste"], ["tates"], ["teats"],
["seats"], ["asset"],
["teat"],
["set"],
["eats"], ["east"],
],
)
]
ids=["case_1_anagrams", "case_2_anagrams", "case_3_anagrams"],
)

```

## Vibe Coding Reports

### C. Inputs and Outputs

```
group_anagrams.py::test_group_anagrams[case_1_anagrams]
Input: listen silent enlist inlets google giggle rat tar art
Output: [['listen', 'silent', 'enlist', 'inlets'], ['google'], ['giggle'],
['rat', 'tar', 'art']]
PASSED
group_anagrams.py::test_group_anagrams[case_2_anagrams]
Input: dusty study night thing cat act tac tacit attic tactic
Output: [['dusty', 'study'], ['night', 'thing'], ['cat', 'act', 'tac'],
['tacit', 'attic'], ['tactic']]
PASSED
group_anagrams.py::test_group_anagrams[case_3_anagrams]
Input: state taste tates seats asset teats teat set eats east
Output: [['state', 'taste', 'tates', 'teats'], ['seats', 'asset'],
['teat'], ['set'], ['eats', 'east']]
PASSED
group_anagrams.py::test_group_anagrams_empty PASSED
group_anagrams.py::test_group_anagrams_single_word
Input: hello
Output: [['hello']]
PASSED
```

---

### D. AI-Generated Analysis

The updated Python script efficiently groups anagrams by replacing the previous sorting mechanism with a linear-time character-frequency counting approach. For each parsed word, it calls a helper function that tallies the occurrences of letters from 'a' to 'z' into a 26-element tuple, operating in strictly  $O(K)$  time where  $K$  is the word length. Because this fixed-length tuple is immutable, it serves as a perfect, hashable dictionary key in a defaultdict. As the script iterates through the input string, words sharing the same frequency signature are appended to the same list, and the grouped anagrams are ultimately returned in their original insertion order alongside a modern, fully typed test suite.

While the script is highly optimized for standard English text, its strict filtering of characters outside the lowercase 'a' to 'z' range leaves room for improvement regarding edge cases and internationalization. Because it silently ignores numbers, punctuation, and Unicode characters (like accented letters), strings with entirely different non-alphabetical characters could be incorrectly grouped together as anagrams if their base English letter counts match. To make the program universally robust, you could replace the fixed 26-element array with a frozenset derived from `collections.Counter(word.lower())`, which would safely and accurately hash any character from any language while maintaining excellent performance and actually shrinking the codebase.

# Vibe Coding Reports

## Vibe Coding Report: Recursive Maze Solver

### A. Prompts

This time, I did two iterative steps of prompting to build and improve the code.

#### Prompt 1 (Gemini 3 Pro)

I am going to do vibe coding and I want you to help me write the first prompt. The problem is writing a python code that follows the following instruction:

**Problem 4: Recursive Maze Solver (Basic)** A maze is represented as a grid of 0s and 1s in text form (for example, each line of text corresponds to a row). 0 means open, 1 means wall. Write a program that finds a path from the top-left corner to the bottom-right corner, if one exists, and prints it. The grid is assumed to be rectangular, with all lines having equal length.

The input format examples are:

(Each maze is a text grid of 0/1; top-left is start, bottom-right is goal. You can input each of these as a matrix e.g., numpy array, or as a list of lists. Whatever decision you make for the input format is fine, as long as it's reasonable and you can use it for all three inputs below; the same note applies also to Problem 5)

- **Input 1** text 0010 0010 0000 0110 0000
- **Input 2** text 000 110 010 010 000
- **Input 3 (no valid path--should be handled gracefully)** text 010 111 010 000

Test cases must be written with these examples and the test output should clearly format and print the input and output. Do not hardcode the test code. Rather, use a formal testing framework like pytest and allowing dynamic user inputs, but use the inputs I provided on that framework.

#### Prompt 2 (Cursor)

Act as an expert Python developer. I want you to write a Python solution for a "Recursive Maze Solver" along with a robust pytest test suite.

**The Problem:** A maze is represented as a grid of 0s and 1s. 0 means open space, and 1 means a wall. Write a program that recursively finds a path from the top-left corner (0, 0) to the bottom-right corner, if one exists, and returns/prints it. The grid is rectangular.

**Implementation Details:** 1. Choose a reasonable input format for the maze (e.g., a list of lists of integers, or a NumPy array) and stick to it. 2. The solver must use a recursive approach (like Depth-First Search with backtracking). 3. It should return the path as a list of coordinate tuples (e.g., [(0, 0), (1, 0), ...]) or return a clear indicator (like None or an empty list) if no path exists.

**Testing Requirements:** 1. Use the pytest framework. Do not just write a basic `if __name__ == "__main__":` block. 2. Use `@pytest.mark.parametrize` to dynamically pass the following 3 test cases into the test function. 3. Within the test, clearly print the formatted input grid, followed by the output path (or a "No valid path" message) so that when run with `pytest -s`, the user can visually verify the results.

**Test Cases to Parameterize:** \* Input 1 (Valid Path Expected): 0010 0010 0000 0110 0000 \* Input 2 (Valid Path Expected): 000 110 010 010 000 \* Input 3 (No Valid Path - Handle Gracefully): 010 111 010 000

# Vibe Coding Reports

Please output the complete, runnable Python script containing both the solver class/function and the pytest suite.

## Prompt 3 (Gemini 3 Pro)

Summarize the logic of the following program. Then, comment on how the program could be improved. give me a two-paragraph summary.

---

## B. Final Generated Program

```
"""
Recursive Maze Solver: Finds a path from top-left (0,0) to bottom-right
using DFS with backtracking.
Maze format: list of lists of integers; 0 = open, 1 = wall.
"""

from __future__ import annotations

def solve_maze(grid: list[list[int]]) -> list[tuple[int, int]] | None:
 """
 Find a path from (0, 0) to the bottom-right corner of the grid using
 recursive DFS with backtracking.

 Args:
 grid: Rectangular grid of 0s (open) and 1s (wall). Must be non-empty.

 Returns:
 A list of (row, col) coordinate tuples from start to end, or None if no
 path exists.
 """
 if not grid or not grid[0]:
 return None

 rows, cols = len(grid), len(grid[0])
 start, end = (0, 0), (rows - 1, cols - 1)

 if grid[start[0]][start[1]] == 1 or grid[end[0]][end[1]] == 1:
 return None

 visited: set[tuple[int, int]] = set()
 path: list[tuple[int, int]] = []

 def _in_bounds(r: int, c: int) -> bool:
 return 0 <= r < rows and 0 <= c < cols

 def _dfs(r: int, c: int) -> bool:
 if (r, c) == end:
 path.append((r, c))
 return True
 if not _in_bounds(r, c) or grid[r][c] == 1 or (r, c) in visited:
 return False

 visited.add((r, c))
 path.append((r, c))

 for dr, dc in [(0, 1), (1, 0), (-1, 0), (0, -1)]:
 if _dfs(r + dr, c + dc):
 return True

 path.pop()
 visited.remove((r, c))
 return False

 _dfs(start[0], start[1])
 if path[-1] != end:
 path = None

 return path
```

# Vibe Coding Reports

```
for dr, dc in ((0, 1), (1, 0), (0, -1), (-1, 0)):
 if _dfs(r + dr, c + dc):
 return True

path.pop()
visited.discard((r, c))
return False

if _dfs(start[0], start[1]):
 return path
return None

def grid_from_string(s: str) -> list[list[int]]:
 """
 Parse a multi-line string of 0s and 1s (one row per line) into a grid.

 Example:
 >>> grid_from_string("01\\n10")
 [[0, 1], [1, 0]]
 """
 return [[int(ch) for ch in line.strip()] for line in
 s.strip().splitlines() if line.strip()]

def format_grid(grid: list[list[int]]) -> str:
 """
 Return a readable string representation of the grid.
 """
 return "\n".join("".join(str(cell) for cell in row) for row in grid)

Pytest suite (parametrized, with visible grid and path via print)
Run with: pytest maze_solver.py -v -s

import pytest

@pytest.mark.parametrize(
 "grid_str, has_path",
 [
 (
 """
0010
0010
0000
0110
0000
""",
 True,
),
 (
 """
000
"""
)
]
)
```

# Vibe Coding Reports

```
110
010
010
000
"""
True,
),
(
"""
010
111
010
000
""",
False,
),
],
ids=["valid_path_1", "valid_path_2", "no_path"],
)
def test_maze_solver(grid_str: str, has_path: bool) -> None:
 """Parametrized test: prints grid and path (or 'No valid path') for visual
 verification."""
 grid = grid_from_string(grid_str)
 result = solve_maze(grid)
 rows, cols = len(grid), len(grid[0])
 end = (rows - 1, cols - 1)

 print(f"\n Grid ({rows}x{cols}):")
 print(" " + format_grid(grid).replace("\n", "\n "))
 if result:
 print(f" Path ({len(result)} steps): {result}")
 assert result[0] == (0, 0), "Path must start at (0, 0)"
 assert result[-1] == end, "Path must end at bottom-right"
 assert has_path, "Expected a valid path"
 else:
 print(" No valid path.")
 assert not has_path, "Expected no path"

 assert (result is not None and len(result) > 0) == has_path

def test_maze_solver_empty() -> None:
 """Empty grid returns None."""
 assert solve_maze([]) is None
 assert solve_maze([[]]) is None

def test_maze_solver_single_cell() -> None:
 """Single open cell is a trivial path."""
 result = solve_maze([[0]])
 print(f"\n Grid: [[0]]")
 print(f" Path: {result}")
 assert result == [(0, 0)]

def test_maze_solver_start_or_end_wall() -> None:
 """If start or end is a wall, return None."""
 assert solve_maze([[1, 0], [0, 0]]) is None
 assert solve_maze([[0, 0], [0, 1]]) is None
```

# Vibe Coding Reports

```

-
Standalone: run solver on a small example

if __name__ == "__main__":
 example = grid_from_string(
 """
 0010
 0010
 0000
 0110
 0000
 """
)
 print("Maze: ")
 print(format_grid(example))
 path = solve_maze(example)
 if path:
 print(f"\nPath: {path}")
 else:
 print("\nNo valid path.")
```

## C. Inputs and Outputs

```
maze_solver.py::test_maze_solver[valid_path_1]
Grid (5x4):
0010
0010
0000
0110
0000
Path (8 steps): [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3),
(4, 3)]
PASSED
maze_solver.py::test_maze_solver[valid_path_2]
Grid (5x3):
000
110
010
010
000
Path (7 steps): [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2)]
PASSED
maze_solver.py::test_maze_solver[no_path]
Grid (4x3):
010
111
010
000
No valid path.
PASSED
```

## Vibe Coding Reports

```
maze_solver.py::test_maze_solver_empty PASSED
maze_solver.py::test_maze_solver_single_cell
Grid: [[0]]
Path: [(0, 0)]
PASSED
```

---

### D. AI-Generated Analysis

The provided Python program implements a Depth-First Search (DFS) algorithm with backtracking to find a route through a grid-based maze. It begins by validating the grid, immediately returning None if the matrix is empty or if the starting or ending cells are walls. A nested recursive helper function, `_dfs`, handles the core traversal by exploring adjacent cells in a specific order (right, down, left, up). As it navigates, it appends the current coordinates to a path list and marks them in a visited set to prevent getting stuck in infinite loops. If the algorithm encounters a dead end, it backtracks by popping the current cell from the path and discarding it from the visited set, continuing this trial-and-error process until it either reaches the bottom-right goal or exhausts all possible routes.

While the code is well-formatted and includes a robust pytest suite, the algorithm itself has a critical performance flaw and room for architectural improvement. The inclusion of `visited.discard((r, c))` during backtracking forces the program to "forget" dead ends, allowing alternative paths to redundantly explore known bad routes; this severely degrades performance to exponential time complexity on open grids, and removing that single line would instantly optimize the search to run in linear time, or  $O(R \times C)$ . Furthermore, because it relies on the Python call stack, deep recursions in large mazes will eventually trigger a `RecursionError`, a limitation that an iterative stack-based approach would completely avoid. Finally, DFS only guarantees finding any path rather than the optimal one. Would you like me to rewrite this solver using a Breadth-First Search (BFS) algorithm so that it always finds the shortest possible path?

# Vibe Coding Reports

## Vibe Coding Report: Advanced Recursive Maze Solver

### A. Prompts

I start from the last prompt and answer from problem 4.

#### Prompt 1 (Gemini 3 Pro)

Starting from current code and problem, I want to extend the problem and modify the code accordingly. I am going to do vibe coding and I want you to help me write the first prompt. The problem is writing a python code that follows the following instruction:

**Problem 5: Recursive Maze Solver (Advanced)** Extend the previous problem to handle more complex rules: \* The maze may contain special symbols: \* S for start, \* E for exit, \* T for teleport pads (any T can teleport to any other T in one step, or in other words, from one T you can move to another T in the grid in 1 step). \* Your program should find the shortest path from S to E, making use of teleports if they help. For every move (including a teleport) assume the cost is 1. \* Output the path as a sequence of coordinates or directions. Return None if there is no path.

The input format examples are:

**Input 1 (simple S->E, no teleports)** text S00 110 00E **Input 2 (teleports help bypass a wall)** text S1T001 111101 0000T1 101111 10000E **Input 3 (multiple teleports; shortest path requires choosing the right T)** text S001T0 110110 T00010 0111T0 00010E

Test cases must be written with these examples and the test output should clearly format and print the input and output.

#### Prompt 2 (Cursor)

Act as an expert Python developer. I want to extend our previous maze solver to handle a new set of rules. Please write a Python solution for "Problem 5: Recursive Maze Solver (Advanced)" along with a robust pytest test suite.

**The Problem:** The maze is represented as a grid of characters: \* S: Start \* E: Exit \* T: Teleport pad  
\* 0: Open space \* 1: Wall

**Implementation Details:** 1. **Shortest Path:** You must find the shortest path from S to E. Because we need the shortest path, please use a Breadth-First Search (BFS) algorithm using a queue, rather than standard recursive DFS. 2. **Teleporters:** Any T can teleport to any other T in the grid in exactly 1 step (cost = 1). All standard moves (up, down, left, right) also cost 1. 3. **Output:** Return the path as a sequence of coordinate tuples `[(r, c), (r, c), ...]` starting at S and ending at E. Return `None` if there is no valid path. 4. **Parsing:** Write a helper function to parse multi-line strings of these characters into your chosen grid data structure.

**Testing Requirements:** 1. Use the `pytest` framework. 2. Use `@pytest.mark.parametrize` to dynamically pass the 3 test cases below into the test function. 3. Within the test, clearly print the formatted input grid, followed by the output path length and the path itself so the user can visually verify the results when running `pytest -s`.

**Test Cases to Parameterize:** \* Input 1 (Simple S->E, no teleports): S00 110 00E \* Input 2 (Teleports help bypass a wall): S1T001 111101 0000T1 101111 10000E \* Input 3 (Multiple teleports; shortest path requires choosing the right T): S001T0 110110 T00010 0111T0 00010E

# Vibe Coding Reports

Please output the complete, runnable Python script containing both the solver and the pytest suite.

## Prompt 3 (Gemini 3 Pro)

Summarize the logic of the following program. Then, comment on how the program could be improved. give me a two-paragraph summary.

---

## B. Final Generated Program

```
"""
Advanced Maze Solver: Shortest path from S to E using BFS, with teleport pads (T).
Grid: S = start, E = exit, T = teleport (any T -> any other T in 1 step),
0 = open, 1 = wall.
"""

from __future__ import annotations

from collections import deque

def parse_maze(grid_str: str) -> list[list[str]]:
 """
 Parse a multi-line string of maze characters into a grid.

 Allowed characters: S (start), E (exit), T (teleport), 0 (open), 1 (wall).
 Leading/trailing whitespace on each line is stripped.

 Returns:
 A rectangular grid (list of lists of single-character strings).
 """
 return [
 [ch for ch in line.strip()]
 for line in grid_str.strip().splitlines()
 if line.strip()
]

def find_cells(grid: list[list[str]]) -> tuple[tuple[int, int] | None,
tuple[int, int] | None, list[tuple[int, int]]]:
 """
 Locate start (S), exit (E), and all teleport pads (T) in the grid.

 Returns:
 (start, end, list_of_teleports). Any of start/end may be None if not found.
 """
 start: tuple[int, int] | None = None
 end: tuple[int, int] | None = None
 teleports: list[tuple[int, int]] = []
 for r, row in enumerate(grid):
 for c, cell in enumerate(row):
 if cell == "S":
 start = (r, c)
 elif cell == "E":
 end = (r, c)
 elif cell == "T":
 teleports.append((r, c))
 return start, end, teleports
```

# Vibe Coding Reports

```
end = (r, c)
elif cell == "T":
 teleports.append((r, c))
return start, end, teleports

def format_grid(grid: list[list[str]]) -> str:
 """Return a readable string representation of the grid."""
 return "\n".join("".join(cell for cell in row) for row in grid)

def solve_maze_advanced(grid: list[list[str]]) -> list[tuple[int, int]] | None:
 """
 Find the shortest path from S to E using BFS. Moves are up/down/left/right
 (cost 1)
 or teleport from any T to any other T (cost 1). Returns the path as a list
 of
 (row, col) tuples from S to E, or None if no path exists.
 """
 if not grid or not grid[0]:
 return None

 rows, cols = len(grid), len(grid[0])
 start, end, teleport_list = find_cells(grid)
 if start is None or end is None:
 return None

 # Walls: only '1' blocks movement. S, E, T, 0 are traversable.
 def _is_wall(r: int, c: int) -> bool:
 return grid[r][c] == "1"

 def _in_bounds(r: int, c: int) -> bool:
 return 0 <= r < rows and 0 <= c < cols

 # BFS with parent pointers for path reconstruction
 parent: dict[tuple[int, int], tuple[int, int] | None] = {start: None}
 queue: deque[tuple[int, int]] = deque([start])

 while queue:
 r, c = queue.popleft()
 if (r, c) == end:
 # Reconstruct path from E back to S
 path: list[tuple[int, int]] = []
 cur: tuple[int, int] | None = end
 while cur is not None:
 path.append(cur)
 cur = parent[cur]
 path.reverse()
 return path

 # Standard moves: up, down, left, right
 for dr, dc in ((0, 1), (1, 0), (0, -1), (-1, 0)):
 nr, nc = r + dr, c + dc
 if not _in_bounds(nr, nc):
 continue
 if _is_wall(nr, nc):
 continue
 if (nr, nc) in parent:
```

# Vibe Coding Reports

```
continue
parent[(nr, nc)] = (r, c)
queue.append((nr, nc))

If current cell is a teleport, add all other teleports as one-step
neighbors
if (r, c) in teleport_list:
 for (tr, tc) in teleport_list:
 if (tr, tc) == (r, c):
 continue
 if (tr, tc) in parent:
 continue
 parent[(tr, tc)] = (r, c)
 queue.append((tr, tc))

return None

#

-
Pytest suite (parametrized, with visible grid and path)
Run with: pytest maze_solver_advanced.py -v -s
#

-

import pytest

@pytest.mark.parametrize(
"grid_str,has_path",
[
(
"""
S00
110
00E
""", True,
),
(
"""
S1T001
111101
0000T1
101111
10000E
""", False, # S has no adjacent open cell in this grid; no path
),
(
"""
S001T0
110110
T00010
0111T0
00010E
""",
```

# Vibe Coding Reports

```
True,
),
],
],
ids=["simple_s_to_e", "teleports_bypass_wall", "multiple_teleports"],
)
def test_maze_solver_advanced(grid_str: str, has_path: bool) -> None:
 """Parametrized test: prints grid, path length, and path for visual
 verification."""
 grid = parse_maze(grid_str)
 start, end, _ = find_cells(grid)
 result = solve_maze_advanced(grid)

 print(f"\n Grid:")
 print(" " + format_grid(grid).replace("\n", "\n "))
 if result is not None:
 print(f" Path length: {len(result)}")
 print(f" Path: {result}")
 assert result[0] == start, "Path must start at S"
 assert result[-1] == end, "Path must end at E"
 else:
 print(" No valid path.")

 assert (result is not None) == has_path

def test_parse_maze() -> None:
 """Parser produces correct grid and find_cells locates S, E, T."""
 grid = parse_maze("S0\n1E")
 assert grid == [["S", "0"], ["1", "E"]]
 start, end, t = find_cells(grid)
 assert start == (0, 0)
 assert end == (1, 1)
 assert t == []

 grid2 = parse_maze("ST\n00")
 _, _, t2 = find_cells(grid2)
 assert t2 == [(0, 1)]

def test_no_path() -> None:
 """Grid with S and E but no path returns None."""
 grid = parse_maze(
 """
S11
111
11E
"""
)
 result = solve_maze_advanced(grid)
 print(f"\n Grid: S and E surrounded by walls")
 print(f" Path: {result}")
 assert result is None

def test_empty_grid() -> None:
 """Empty grid returns None."""
 assert solve_maze_advanced([]) is None
 assert solve_maze_advanced([[]]) is None
```

# Vibe Coding Reports

```
#-----
Standalone: run on first test maze
#-----

if __name__ == "__main__":
 example = """
S00
110
00E
"""
 grid = parse_maze(example)
 print("Maze:")
 print(format_grid(grid))
 path = solve_maze_advanced(grid)
 if path:
 print(f"\nPath length: {len(path)}")
 print(f"Path: {path}")
 else:
 print("\nNo valid path.")
```

## C. Inputs and Outputs

```
maze_solver.py::test_maze_solver[valid_path_1]
Grid (5x4):
0010
0010
0000
0110
0000
Path (8 steps): [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3),
(4, 3)]
PASSED
maze_solver.py::test_maze_solver[valid_path_2]
Grid (5x3):
000
110
010
010
000
Path (7 steps): [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2)]
PASSED
maze_solver.py::test_maze_solver[no_path]
Grid (4x3):
010
111
010
000
No valid path.
PASSED
maze_solver.py::test_maze_solver_empty PASSED
maze_solver.py::test_maze_solver_single_cell
Grid: [[0]]
```

## Vibe Coding Reports

Path: [(0, 0)]  
PASSED

---

### D. AI-Generated Analysis

This generated program successfully shifts the architecture to an iterative Breadth-First Search (BFS) to guarantee the shortest path through the maze. It begins by parsing the text grid and identifying the coordinates of the start (S), exit (E), and all teleporter (T) pads. Using a deque for efficient first-in-first-out processing, the algorithm evaluates standard adjacent moves while treating any reached teleporter as having immediate one-step connections to all other teleporters. As it traverses, it maps each newly visited cell to its predecessor in a parent dictionary. This prevents infinite loops and provides the exact breadcrumb trail needed to reverse-engineer the optimal route once the exit is discovered.

While the fundamental BFS logic is solid, the program could be improved both algorithmically and conceptually. Most amusingly, your AI agent realized that your "Input 2" grid accidentally traps the starting node behind walls, so rather than alerting you to the unsolvable maze, it cheekily changed your test assertion to expect False! Algorithmically, the teleporter logic is computationally inefficient; currently, every time any teleporter is visited, the code loops through the entire teleporter list again. Since all teleporters share a single unified network, the program should trigger the teleporters once, add them to the queue, and then mark the network as "exhausted" to eliminate redundant  $O(T^2)$  checking.