Topic Introduction

Today, we're diving into one of the most beloved patterns in coding interviews: **Depth-First Search (DFS) on a 2D Grid**, with a special twist using **Trie (Prefix Tree)** data structures. This powerful combo helps you efficiently search for words in a board of letters, a classic challenge featured in interviews at nearly every major tech company.

What is DFS in a 2D Grid?
DFS is a graph traversal technique where you visit a node (or cell), then recursively visit its neighbors before "backtracking" to explore new paths. On a 2D grid, each cell can be seen as a node, and you can move up, down, left, or right (and sometimes diagonally, depending on the problem). You'll typically use DFS to find paths, connected regions, or solve puzzles that require visiting and marking cells.

Why is it useful?
DFS lets you explore all possible paths or configurations in a grid, making it perfect for problems like maze solving, island counting, or, as in today's problems, searching for words. Adding a Trie makes searching for multiple words efficient, avoiding redundant work by pruning impossible paths early.

Simple Example:
Imagine you have a 3x3 grid, and you want to see if you can spell "CAT" by moving from letter to letter (no diagonals). You'd start at each 'C', try all 4 directions for the next letter, and so on, marking cells as visited to avoid revisiting in the same path.

The Problems: A Family of Word Searches

Today's trio:

- Word Search (LeetCode 79)
- Word Search II (LeetCode 212)
- Boggle Game (LeetCode 212, similar variant)

These are grouped for a reason: they all ask you to find words in a 2D grid of letters, moving to adjacent cells (up/down/left/right), using DFS. The first asks for a single word, the second for multiple (with a trie for efficiency), the third is a Boggle twist that also uses these patterns. Mastering this trio will make you confident in grid + search + pruning problems.

Problem 1: Word Search

LeetCode 79: Word Search

Problem Statement (in a nutshell):
Given a 2D grid of letters and a target word, can you trace out the word in the grid by moving from one cell to an adjacent cell (up, down, left, right)? Each letter must be used exactly once per path.

Example:

```
board = [
```

```
  ["A", "B", "C", "E"],
  ["S", "F", "C", "S"],
  ["A", "D", "E", "E"]
]
word = "ABCCED"
```

Output: `True`

Explanation: Start at (0,0) -> (0,1) -> (0,2) -> (1,2) -> (2,2) -> (2,1).

Another Test Case to Try:

```
board = [
  ["A", "B", "C", "E"],
  ["S", "F", "C", "S"],
  ["A", "D", "E", "E"]
]
word = "SEE"
```

Expected output: `True` (Try tracing it!)

Thinking Process:

- For each cell, check if it matches the first letter.
- If so, start searching recursively for the next letter in all 4 directions.
- Mark cells as visited to avoid cycles.
- If you finish all the letters, return True. If you run out of options, backtrack.

Brute-Force Approach:

Try all possible paths starting from every cell. This can be slow:

Time complexity: O(N * 3^L), where N = number of cells, L = length of the word. Why 3^L? At each step, you have up to 3 (not 4) options, because you can't revisit the previous cell.

Optimal Approach:

DFS with backtracking.

- For each cell:
    - If it matches the current word letter:
        - Mark it as visited (temporarily change the letter or use a visited set).
        - Recursively search its neighbors for the next letter.
        - Unmark when done (restore the letter).
    - If you reach the end of the word, return True.
    - If all paths fail, return False.

Let’s see it in action!

Python Solution

```
def exist(board, word):
    rows, cols = len(board), len(board[0])
```

```
    def dfs(r, c, i):
        # If all letters are found
        if i == len(word):
            return True
        # If out of bounds or not matching
        if r < 0 or c < 0 or r >= rows or c >= cols:
            return False
        if board[r][c] != word[i]:
            return False

        # Mark as visited by temporarily changing the value
        temp = board[r][c]
        board[r][c] = "#"

        # Explore all directions
        found = (dfs(r+1, c, i+1) or
                 dfs(r-1, c, i+1) or
                 dfs(r, c+1, i+1) or
                 dfs(r, c-1, i+1))

        # Restore the letter (backtrack)
        board[r][c] = temp
        return found

    for r in range(rows):
        for c in range(cols):
            if dfs(r, c, 0):
                return True
    return False
```

Time Complexity: O(N * 3^L)

N is the total number of cells, L is the word length.

Space Complexity: O(L) for recursion stack.

What's Happening Here?

- `dfs(r, c, i)`: Tries to find the i-th character of the word starting from cell (r, c).
- Marks the cell as visited (by changing its value) to prevent cycles.
- Searches all 4 directions for the next letter.
- If all letters are found (`i == len(word)`), returns True.
- If a path fails, restores the cell's letter (backtracking).

Trace Example:

For the word `"ABCCED"` in the earlier example, the search starts at (0,0), which is "A", and recursively follows the path matching each letter, marking and unmarking as it goes.

Try This Test Case (Dry-Run!):

```
board = [
  ["A", "B"],
  ["C", "D"]
]
word = "ACDB"
```

Can you find the word in this grid? What does your function return?

Take a moment to solve this on your own before reading further!

Problem 2: Word Search II

LeetCode 212: Word Search II

Problem Statement:
Given a 2D board and a list of words, find all words in the list that can be formed by tracing adjacent letters in the grid (same rules as before).

How is it different?
Now you have to search for **many words** at once. Brute-force would be to run the previous function for every word. But that's slow!

Optimal Approach:
Use a **Trie** (prefix tree) to store the words. This lets you share work between words that have common prefixes. While DFSing from a cell, you traverse the Trie in parallel, so you immediately stop exploring paths that can't form any valid word.

Brute-Force:
Run DFS for each word. Time: O(W N 3^L), W = number of words.

With Trie + DFS:

- Build a Trie of all words.
- For each cell, start DFS:
    - At each step, check if the current path is a prefix in the Trie.
    - If it forms a word, add it to the result.
    - Mark cells as visited as before.
    - Backtrack after.

Example:

```
board = [
  ["o", "a", "a", "n"],
  ["e", "t", "a", "e"],
  ["i", "h", "k", "r"],
  ["i", "f", "l", "v"]
]
```

```
words = ["oath", "pea", "eat", "rain"]
Output: ["eat", "oath"]
```

Another Test Case to Try:

```
board = [
  ["a", "b"],
  ["c", "d"]
]
words = ["ab", "cb", "ad", "bd", "ac", "ca", "da", "bc", "db", "adcb"]
Expected Output: ["ab", "bd", "ac", "ca", "db"]
```

Pseudocode:

```
1. Build a Trie from the list of words.
2. For each cell in the board:
    - Start DFS with the current node in the Trie.
    - If the Trie node marks a complete word, add it to results.
    - For each neighbor:
        - If the letter exists as a child in Trie:
            - Mark as visited.
            - DFS into neighbor with next Trie node.
            - Unmark after.
3. Return all found words (avoid duplicates).
```

Step-by-Step Example (on the main example):

- Build Trie:
    - "oath", "pea", "eat", "rain"
- Start DFS from every cell.
    - If you start at (0,0) "o", follow Trie for "o" -> "a" -> ... until you complete "oath".
    - Similarly for others.
- Collect found words.

Time Complexity:

Roughly O(N * 4^L), but much faster in practice due to Trie pruning.

Space Complexity:

O(Total letters in all words) for the Trie, plus recursion stack.

Trace Example:

Suppose you start at (0,0) "o":

- "o" in Trie? Yes. Go to (0,1) "a"...
- Continue following Trie and grid until you find "oath".

Try This Test Case (Dry-Run!):

```
board = [
  ["a", "p", "p", "l", "e"]
```

```
]
words = ["apple", "app", "ape", "plea"]
Expected Output: ["apple", "app"]
```

Which words can you find?

Problem 3: Boggle Game (LeetCode 212 variant)

Boggle Game

(Note: Boggle variants may differ, but typically you can move in **8 directions**: up, down, left, right, and diagonals. Sometimes, you may be asked to find all valid words in the grid.)

What's Different Here?

- You may be allowed to move in **8 directions** (vs 4).
- Sometimes, you need to find all words that can be formed (like Boggle).
- The word list may be a standard dictionary.
- Each cell can be used only once per word.

Approach:

Very similar to Problem 2, but extend DFS to 8 directions.

Pseudocode:

```
1. Build a Trie of all valid words.
2. For each cell in the board:
    - Start DFS with the current node in the Trie.
    - If Trie node marks a word, add to result.
    - For each of 8 neighbors:
        - If letter exists as a child in Trie:
            - Mark as visited.
            - DFS into neighbor.
            - Unmark after.
3. Return all found words (no duplicates).
```

Example:

```
board = [
  ["g", "i", "z"],
  ["u", "e", "k"],
  ["q", "s", "e"]
]
words = ["geeks", "for", "quiz", "go"]
Output: ["geeks", "quiz"]
```

Another Test Case to Try:

```
board = [
```

```
  ["a", "b"],
  ["c", "d"]
]
words = ["abc", "abd", "abdc", "bdc"]
Expected Output: ["abc", "abd"]
```

Try tracing these out, moving in 8 directions.

Time and Space Complexity:

Time: O(N * 8^L), N = number of cells, L = max word length.

Space: O(Total Trie size + recursion stack).

Give it a try!

Can you modify your earlier DFS to work with diagonals? Try writing the code and dry-run the test case.

Summary and Next Steps

We grouped these problems together because they all use **DFS on a 2D grid**, often enhanced with a **Trie** to efficiently search for words. The core patterns you've practiced today are:

- **DFS with backtracking:** Explore all paths, marking/unmarking cells as you go.
- **Using a Trie:** Greatly speeds up word searches when dealing with multiple queries, by pruning impossible paths early.
- **Grid traversal:** Mastering up/down/left/right (and diagonals!) is a must for many real-world and interview problems.

Common mistakes to avoid:

- Not marking cells as visited, leading to cycles.
- Forgetting to unmark (backtrack) cells after DFS.
- Not pruning paths early with the Trie, leading to TLE (Time Limit Exceeded).
- For Boggle-style problems: not searching diagonally or missing valid directions.

What to Do Next

- Solve all 3 problems on your own, even if you already saw the code for the first.
- Try modifying your solution for Problem 2 to work for the Boggle variant (8 directions).
- Explore other classic grid search problems, like "Number of Islands" or "Surrounded Regions".
- Compare your code to others on LeetCode — especially see how they handle edge cases and efficiency.
- Challenge yourself: Can you implement the Trie from scratch for these problems?
- Remember: Getting stuck is part of learning. Keep practicing and you'll internalize these patterns!

Happy coding!