Topic Introduction

Today's PrepLetter dives into a trio of classic interview problems that all revolve around one powerful data structure: the **Trie** (also known as a Prefix Tree). If you've ever needed to store and search for words efficiently, you've probably bumped into this structure — and if not, you're about to see why it's such a favorite for interviewers.

What is a Trie?

A Trie is a tree-like data structure where each node represents a single character of a word. Paths from the root to a node spell out prefixes or complete words. Unlike hash tables or sets, Tries allow you to efficiently store and retrieve strings, especially when dealing with prefixes.

How does it work?

Each node contains links (children) to possible next letters in the alphabet. Common prefixes are shared, which saves space and speeds up prefix-based search. A node is typically marked to indicate if it represents the end of a valid word.

Why is it useful in interviews?

Tries shine whenever you need to handle dictionary words, prefix-matching, autocomplete, or efficient wildcard searches. They're also a great test of your ability to design custom data structures and manipulate pointers/references.

Quick Example (not from our target problems):

Suppose you want to store the words: "cat", "car", and "dog". The Trie would look like this:

```
(root)
 |
 c
 |
 a
 / \
t   r
     \
      (end)
 |
d
 |
o
 |
g
```

Here, "cat" and "car" share the "ca" prefix, while "dog" branches off from the root.

Now let's look at three problems that explore different uses and flavors of Tries:

- **Implement Trie (Prefix Tree)**: Build the basic Trie data structure for add/search operations.
- **Design Add and Search Words Data Structure**: Enhance the Trie to handle wildcard searches with the '.' character.
- **Word Search II**: Use a Trie for multi-word searching on a 2D grid efficiently.

Why these three together?

They're all Trie-based, but each layers on new complexity:

- The first is foundational (basic structure and search).
- The second adds wildcards, requiring recursive or backtracking search.
- The third uses a Trie in a creative way for searching many words in a grid, showing Tries as an optimization tool.

Ready to level up your understanding of Tries? Let's jump in!

Problem 1: Implement Trie (Prefix Tree)

Problem Statement (Rephrased):

Leetcode 208: Implement Trie (Prefix Tree)

Design and implement a Trie (prefix tree) with three operations:

- `insert(word)`: Adds a word to the Trie.
- `search(word)`: Returns true if the word exists in the Trie.
- `startsWith(prefix)`: Returns true if there is any word in the Trie that starts with the given prefix.

Example:

```
trie = Trie()
trie.insert("apple")
trie.search("apple")    # returns True
trie.search("app")      # returns False
trie.startsWith("app")  # returns True
trie.insert("app")
trie.search("app")      # returns True
```

Walkthrough:

- First, we insert "apple".
- Then, we search for "apple" (should return True).
- Searching for just "app" returns False (it wasn't inserted as a word yet).
- But startsWith("app") is True, since "apple" exists.
- After inserting "app", searching for "app" returns True.

Another Test Case to Try:

After inserting "banana" and "band", what should `trie.startsWith("ba")` return? What about `trie.search("ban")`?

Brute-force Approach:

You could use a list to store all words, then scan the list linearly for each search or prefix check.

- **Time Complexity:** O(N * L), where N is the number of words, and L is the word length.
- **Drawback:** Too slow for large datasets.

Optimal Approach: Use a Trie!

- **Pattern:** Tree structure where each node represents a character.
- **How:**
  - Each node has a dictionary (or array) of children.

- Each node can be marked as `is_end` (does this node finish a word?).
- To insert, traverse letters and create nodes as needed.
- To search, traverse; if we finish and `is_end` is set, return True.
- To check a prefix, traverse; if all letters exist, return True.

Python Solution:

```
class TrieNode:
    def __init__(self):
        # Each node holds a mapping from character to TrieNode
        self.children = {}    # e.g., 'a' : TrieNode()
        self.is_end = False   # Is this the end of a word?

class Trie:
    def __init__(self):
        # The root node doesn't hold any character
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            # If the character is not in children, add it
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True   # Mark the end of a word

    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def startsWith(self, prefix: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

Time Complexity:

- `insert`: O(L)
- `search`: O(L)

- `startsWith`: O(L)

where L is the length of the word or prefix.

Space Complexity:

- O(N * L), N = number of words, L = average word length.

Explanation of Each Part:

- `TrieNode`: Represents each character node. Children can be any characters; `is_end` tracks if this node finishes a word.
- `Trie`: Holds the root node, and provides insert/search/prefix methods.
- Each method walks down the Trie, following each character, creating or checking nodes as it goes.

Trace Example:

Suppose we insert "cat", then search for "cat":

- Insert "c": not present, create node.
- "a": not present, create node.
- "t": not present, create node. Mark as end.

Now search for "cat":

- "c": exists, move down.
- "a": exists.
- "t": exists, and is marked as end. Return True.

Try This Test Case:

Insert "dog", "door", then check `search("do")` (should be False), and `startsWith("do")` (should be True).

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Design Add and Search Words Data Structure

Problem Statement (Rephrased):

Leetcode 211: Design Add and Search Words Data Structure

Build a data structure that supports:

- `addWord(word)`: Add a word.
- `search(word)`: Returns True if the word is in the structure.

The twist: the word may contain '.' which matches any single letter.

Example:

```
wordDictionary = WordDictionary()
wordDictionary.addWord("bad")
wordDictionary.addWord("dad")
wordDictionary.addWord("mad")
wordDictionary.search("pad") # False
wordDictionary.search("bad") # True
wordDictionary.search(".ad") # True
wordDictionary.search("b..") # True
```

How is this different from Problem 1?

- The wildcard '.' means you can't just follow a single path in the Trie; you might need to follow *all* possible paths at each '.'.

Brute-force Approach:

Store all words in a list; for each search, check every word, comparing characters (with '.' matching any character).

- **Time Complexity:** O(N * L), N = words, L = word length.

Optimal Approach: Trie with Backtracking for Wildcards

- **Pattern:** Trie + recursion for wildcards.
- **How:**
  - When adding words: same as Problem 1.
  - For search, if the character is '.' recursively try all possible child nodes at that position.
  - If a character is a letter: follow its specific child as normal.
  - If you reach the end and `is_end` is True, return True.

Pseudocode:

```
class Node:
    children: dict
    is_end: bool

class WordDictionary:
    root = Node()

    addWord(word):
        node = root
        for char in word:
            if char not in node.children:
                node.children[char] = Node()
            node = node.children[char]
        node.is_end = True

    search(word):
        return dfs(root, word, 0)

    function dfs(node, word, i):
        if i == len(word):
            return node.is_end
        char = word[i]
        if char == '.':
            for child in node.children.values():
                if dfs(child, word, i+1):
                    return True
            return False
        else:
            if char not in node.children:
```

```
                return False
            return dfs(node.children[char], word, i+1)
```

Example Trace:

Suppose we have "cat" and "car", and search for "ca.":

- 'c': found, move down.
- 'a': found, move down.
- '.': try all children ('t', 'r').
- For 't': end of word, is_end True (so "cat" matches).
- For 'r': also is_end, so "car" matches.
- Return True.

Try This Test Case:

Add "bat", "ball", search for "ba.." (should be True), search for "b.t" (should be True), search for "b..l" (should be True).

Time and Space Complexity:

- `addWord`: O(L)
- `search`: Worst case O(26^L) if all wildcards, but usually O(L * B), B = branching factor (average non-wildcard nodes).

Problem 3: Word Search II

Problem Statement (Rephrased):

Leetcode 212: Word Search II

Given a 2D board of letters and a list of words, find all words in the list that can be formed in the board by tracing adjacent (horizontally or vertically) letters. Each letter cell may be used only once per word.

What's different or more challenging?

- Searching for *many* words at once.
- Need to efficiently check if a path matches any word or prefix in the list.

Naive Approach:

For each word, do a separate DFS on the board to check if it exists.

- **Time Complexity:** O(W *M* N * L), W = words, M/N = board size, L = word length.

Optimal Approach: Trie + Backtracking Search on the Board

- **Pattern:** Build a Trie of all words (so you can check prefixes quickly as you search).
- **How:**
  - Build a Trie from the word list.
  - For each cell in the board, start a DFS:
    - At each step, check if the prefix formed so far exists in the Trie.
    - If it does, continue; if not, prune the search.
    - If you match a full word (`is_end`), add it to results.
    - Use a visited set or temporarily mark cells to avoid revisiting.
    - Remove found words from the Trie to prevent duplicates.
- This approach only explores possible prefixes, saving tons of work.

Pseudocode:

```
Build Trie from words
result = []

for each cell (i, j) in board:
    dfs(i, j, root of Trie, path="")

function dfs(i, j, node, path):
    if board[i][j] not in node.children:
        return
    node = node.children[board[i][j]]
    path += board[i][j]
    if node.is_end:
        add path to result
        node.is_end = False      # Avoid duplicates

    mark board[i][j] as visited
    for each neighbor (ni, nj):
        if valid and not visited:
            dfs(ni, nj, node, path)
    unmark board[i][j] as visited
```

Example:

Suppose `board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]]` and `words = ["oath","pea","eat","rain"]`

The output is ["oath","eat"].

Try This Test Case:

Board: [["a","b"],["c","d"]], Words: ["ab","ac","bd","cd","abcd"].

What words can be found?

Time and Space Complexity:

- Building Trie: O(W * L)
- DFS: O(M N 4^L), but pruned heavily thanks to Trie.
- Space for Trie: O(W * L), W = words, L = length.

Reflect:

Notice how using a Trie here prunes the search space massively. Without it, performance would suffer badly!

Summary and Next Steps

Today you dove deep into three classic Trie-based interview problems.

- **Key takeaways:**
  - Tries are perfect for storing and searching words (especially prefix-based).

- They support not just exact searches, but wildcards and prefix queries.
- For multi-word searches (like Word Search II), Tries can be used to combine all searches into one efficient traversal.

Common pitfalls:

- Forgetting to mark word ends (`is_end`) properly.
- Not handling wildcards ('.') recursively.
- Not pruning the search space in grid-based problems, leading to TLE.
- Accidentally revisiting cells in grid DFS.

Patterns to remember:

- Trie for exact/prefix matching.
- Recursion/backtracking for wildcards.
- Combining Trie with DFS for grid searches.

Action List

- Practice all three problems — try to code them *without* peeking at the solutions.
- For Problem 2 and 3, consider alternate solutions (e.g., backtracking without a Trie, or using a set).
- Explore similar problems: autocomplete systems, prefix sum queries, and spell-checkers.
- Compare your code with others to spot edge-case handling, memory usage, and style.
- If you get stuck, break the problem down: try pen-and-paper, or explain it to a friend or rubber duck.
- Most importantly: keep practicing! Every Trie you write is a win for your interview toolkit.

Happy coding!