Trie

Chunk 1: Problems 1–4

1. Implement Trie (Prefix Tree)

2. Word Break

3. Design Add and Search Words Data Structure

4. Design In-Memory File System

We'll group them by pattern, provide a template for each core technique, then solve each problem in full detail.

Core Pattern Grouping

Pattern	Problems
Trie / Prefix Tree	Implement Trie, Design Add and Search Words, Design In-Memory File System
Dynamic Programming (DP)	Word Break

Let's go through each pattern in depth.

Pattern 1: Trie / Prefix Tree

How to Recognize

- You're dealing with **prefix-based queries**: e.g., "find all words starting with 'app' ", "check if a word is in a dictionary", "autocomplete".
- Frequent insert/search operations on strings.
- Wildcard matching (e.g., `.` in "add and search" problems).
- Need efficient storage of overlapping prefixes (like in dictionaries or file paths).

Step-by-Step Thinking Process (The Recipe)

1. **Define Trie Node Structure**
   - Each node has:
     - A dictionary (`children`) to map char → child node.
     - A boolean flag `is_end` to mark end of a valid word.
2. **Insertion**
   - Traverse from root; create nodes as needed.
   - At the last character, set `is_end = True`.
3. **Search**
   - Traverse characters one by one.
   - If any char missing → return `False`.
   - If reached end and `is_end == True` → return `True`.
4. **Wildcard Search (Backtracking)**
   - When encountering `.`, explore **all children** recursively.
   - Use DFS/backtracking to try every possibility.
5. **Path Compression via Shared Nodes**
   - Reuse nodes across words (e.g., "cat" and "car" share "c" → "a").

Common Pitfalls & Edge Cases

- Forgetting to reset `is_end` when inserting new words (if reusing nodes).
- Not handling empty string input properly.
- In wildcard search, not backtracking correctly — only exploring one path instead of all.
- Memory leaks if not cleaning up unused nodes (less critical in interviews).
- Using `dict.get()` without checking existence leads to bugs.

Problem 1: Implement Trie (Prefix Tree)

Summary

Design a data structure that supports: - Insert a word. - Check if a word exists. - Check if any word starts with a given prefix.

Official Example I/O

```
Input:
["Trie", "insert", "search", "startsWith", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"]]

Output:
[null, null, true, true, false]
```

Note: `"app"` is a prefix but not a complete word → `search("app") = false`.

Python Solution with Inline Comments

```python
class TrieNode:
    def __init__(self):
        # Dictionary to store child nodes: char -> TrieNode
        self.children = {}
        # Boolean flag to mark end of a word
        self.is_end = False


class Trie:
    def __init__(self):
        # Root node is an empty node (no char)
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        """Insert a word into the trie."""
        node = self.root
        for char in word:
            # If char not in children, create a new node
            if char not in node.children:
                node.children[char] = TrieNode()
            # Move to the next node
            node = node.children[char]
        # Mark the end of the word
        node.is_end = True

    def search(self, word: str) -> bool:
```

```python
        """Check if a word exists in the trie."""
        node = self.root
        for char in word:
            if char not in node.children:
                return False  # Word doesn't exist
            node = node.children[char]
        # Return True only if this is the end of a valid word
        return node.is_end

    def startsWith(self, prefix: str) -> bool:
        """Check if any word starts with the given prefix."""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False  # No such prefix
            node = node.children[char]
        # We don't need is_end here - just need to exist
        return True


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Initialize trie
    trie = Trie()

    # Insert "apple"
    trie.insert("apple")

    # Search for "apple" → should be True
    print(trie.search("apple"))  # Output: True

    # Check if "app" is a prefix → True
    print(trie.startsWith("app"))  # Output: True

    # Search for "app" → False (not a full word)
    print(trie.search("app"))  # Output: False
```

Example Walkthrough (Official Input)

1. `insert("apple")`:
   - Start at root.
   - 'a' → new node → move to it.
   - 'p' → new node → move.
   - 'p' → new node → move.
   - 'l' → new node → move.
   - 'e' → new node → set `is_end = True`.
   - Final state: `a → p → p → l → e` (end marked).
2. `search("apple")`:
   - Traverse `a-p-p-l-e`. All chars exist and `e` has `is_end = True` → return `True`.
3. `startsWith("app")`:
   - Traverse `a-p-p`. All exist → return `True` (even if not a full word).
4. `search("app")`:
   - Traverse `a-p-p`. But the last node does **not** have `is_end = True` → return `False`.

Matches expected output: `[null, null, true, true, false]`

Complexity Analysis

- **Time**:
  - `insert`: O(m), where m = length of word.
  - `search`: O(m), same.
  - `startsWith`: O(m), same.
- **Space**:
  - O(ALPHABET_SIZE × N × M), worst case, where N = number of words, M = avg length.
  - In practice: shared prefixes reduce space usage significantly.

Problem 2: Word Break

Summary

Given a string `s` and a dictionary of words, determine if `s` can be segmented into a space-separated sequence of one or more dictionary words.

> Example: `s = "leetcode"`, dict = `["leet", "code"]` → return `True`.

Official Example I/O

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: "leetcode" can be segmented as "leet code".
```

Another example:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
```

Python Solution with Inline Comments

```python
def wordBreak(s: str, wordDict: list[str]) -> bool:
    """
    Determines if string s can be broken into words from wordDict.
    Uses Dynamic Programming: dp[i] = True if s[:i] can be segmented.
    """
    # Convert wordDict to set for O(1) lookup
    word_set = set(wordDict)

    # dp[i] represents whether s[0:i] can be segmented
    n = len(s)
    dp = [False] * (n + 1)

    # Base case: empty string can always be segmented
```

```python
    dp[0] = True

    # Fill dp array from left to right
    for i in range(1, n + 1):
        # Try every possible ending position j < i
        for j in range(i):
            # If s[0:j] is breakable AND s[j:i] is in dictionary
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break  # No need to check further j values

    return dp[n]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example 1
    s1 = "leetcode"
    wordDict1 = ["leet", "code"]
    print(wordBreak(s1, wordDict1))  # Output: True

    # Example 2
    s2 = "applepenapple"
    wordDict2 = ["apple", "pen"]
    print(wordBreak(s2, wordDict2))  # Output: True

    # Example 3 (should be False)
    s3 = "catsandog"
    wordDict3 = ["cats", "dog", "sand", "and", "cat"]
    print(wordBreak(s3, wordDict3))  # Output: False
```

Example Walkthrough (Example 1: `s = "leetcode"`)

- `word_set = {"leet", "code"}`
- `dp = [True, False, False, False, False, False, False, False, False, False]` (length 9+1)

Loop `i` from 1 to 9:

- `i=4`: `s[0:4] = "leet"` → `dp[0]=True` and `"leet"` in set → `dp[4] = True`

- `i=8`: `s[4:8] = "code"` → `dp[4]=True` and `"code"` in set → `dp[8] = True`
- Final `dp[9] = True` → return `True`

Success!

Complexity Analysis

- **Time**: $O(n^2 \times m)$, where n = length of `s`, m = average word length (due to substring slicing).
  - But since we use `set`, lookup is $O(1)$ per word.
  - Nested loops: $O(n^2)$, inner loop checks substrings.
- **Space**: $O(n + k)$, where k = size of `word_set`, plus `dp` array of size n+1.

  Optimization Tip: Instead of slicing `s[j:i]`, precompute or avoid copying strings. But acceptable in interviews.

Problem 3: Design Add and Search Words Data Structure

Summary

Design a data structure that supports: - `addWord(word)` – add a word. - `search(word)` – check if a word exists. Supports wildcards (`.` matches any single character).

> E.g., `search("b..")` would match `"bat"`, `"bed"`, etc.

Official Example I/O

```
Input:
["WordDictionary", "addWord", "addWord", "addWord", "search",
"search", "search", "search"]
[[], ["bad"], ["dad"], ["mad"], ["pad"], ["bad"], [".ad"], ["b.."]]

Output:
[null, null, null, null, false, true, true, true]
```

Python Solution with Inline Comments

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class WordDictionary:
    def __init__(self):
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        """Insert a word into the trie."""
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word: str) -> bool:
        """
        Search for a word with possible '.' wildcards.
        Use DFS to explore all possibilities when '.' is encountered.
        """
        def dfs(node, index):
            # Base case: reached end of word
            if index == len(word):
```

```python
                return node.is_end

            char = word[index]

            if char == '.':
                # Try all possible children (wildcard)
                for child in node.children.values():
                    if dfs(child, index + 1):
                        return True
                return False
            else:
                # Regular character: must exist in children
                if char not in node.children:
                    return False
                return dfs(node.children[char], index + 1)

        return dfs(self.root, 0)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    wd = WordDictionary()
    wd.addWord("bad")
    wd.addWord("dad")
    wd.addWord("mad")

    print(wd.search("pad"))
    # Output: False
    print(wd.search("bad"))
    # Output: True
    print(wd.search(".ad"))
    # Output: True (matches "bad", "cad"? but only "bad", "dad", "mad" exist)
    print(wd.search("b.."))
    # Output: True (
        # matches "bad", "bed"? but only "bad",
        # "dad", "mad" → "bad" and "mad" are valid)
```

Note: `.ad` → matches `bad`, `dad`, `mad` → all have `'a'` at index 1, `'d'` at index 2 → so yes.

Example Walkthrough (`.ad`)

- Start at root.
- Index 0: `'.'` → try all children: `b`, `d`, `m`.
- Try `b` → go to `b` node.
  - Index 1: `'a'` → exists under `b`? Yes → go to `a`.
  - Index 2: `'d'` → exists under `a`? Yes → go to `d`.
  - Index 3: end → check `is_end` → `True` → return `True`.

Returns `True`.

Complexity Analysis

- **Time**:
  - `addWord`: O(m), m = word length.
  - `search`: O(26^m) worst-case (if all dots), but usually much better due to pruning.
    - In practice: bounded by trie depth and branching factor.
- **Space**: O(N × M), where N = number of words, M = avg length.

> Key Insight: Backtracking + Trie makes wildcard search feasible.

Problem 4: Design In-Memory File System

Summary

Design a file system with: - `mkdir(path)` – create directory at `path`. - `addContentToFile(filePath, content)` – write content to file. - `readContentFromFile(filePath)` – read file content. - Paths are like `/a/b/c`, and can be relative or absolute.

> Important: Files and directories coexist; `/a/file.txt` is a file, `/a/b` is a dir.

Official Example I/O

```
Input:
["FileSystem", "mkdir", "addContentToFile", "readContentFromFile", "mkdir", "addContentToFile
[[], ["/a"], ["/a/b.txt", "hello"], ["/a/b.txt"], ["/c"], ["/c/d.txt", "world"], ["/c/d.txt"]

Output:
[null, null, null, "hello", null, null, "world"]
```

Python Solution with Inline Comments

```python
class FileSystem:
    def __init__(self):
        # Root directory (empty name, stores children)
        self.root = {}

    def mkdir(self, path: str) -> None:
        """Create a directory at the given path."""
        parts = path.split('/')
        node = self.root

        # Traverse each part of the path
        for part in parts[1:]:  # Skip first empty part from split('/')
            if part not in node:
                node[part] = {}  # Create new directory
            node = node[part]

    def addContentToFile(self, filePath: str, content: str) -> None:
        """Add content to a file. Creates file if not exists."""
        parts = filePath.split('/')
        node = self.root

        # Navigate to parent directory
        for part in parts[1:-1]:
            node = node[part]

        # Last part is filename
        filename = parts[-1]
```

```python
        # If file doesn't exist, create it (as a string)
        if filename not in node:
            node[filename] = ""  # Initialize empty file

        # Append content
        node[filename] += content

    def readContentFromFile(self, filePath: str) -> str:
        """Read the content of a file."""
        parts = filePath.split('/')
        node = self.root

        # Navigate to parent directory
        for part in parts[1:-1]:
            node = node[part]

        # Get the file content
        filename = parts[-1]
        return node[filename]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    fs = FileSystem()

    fs.mkdir("/a")
    fs.addContentToFile("/a/b.txt", "hello")
    print(fs.readContentFromFile("/a/b.txt"))  # Output: hello

    fs.mkdir("/c")
    fs.addContentToFile("/c/d.txt", "world")
    print(fs.readContentFromFile("/c/d.txt"))  # Output: world
```

Example Walkthrough

1. `mkdir("/a")`:
    - Split → `['', 'a']`
    - Go to `root['a']` → create `{}`.

2. `addContentToFile("/a/b.txt", "hello")`:

   - Split → `['', 'a', 'b.txt']`
   - Go to `root['a']` → then `node = root['a']`
   - `filename = 'b.txt'`, not in `node` → create `node['b.txt'] = ""`
   - Append `"hello"` → now `node['b.txt'] = "hello"`

3. `readContentFromFile("/a/b.txt")`:

   - Same path → returns `"hello"`.

Matches expected output.

Complexity Analysis

- **Time**:
  - `mkdir`: O(k), k = number of path components.
  - `addContentToFile`: O(k), same.
  - `readContentFromFile`: O(k), same.
- **Space**: O(total characters stored), including file names and content.

  Key Insight: Use nested dictionaries to simulate hierarchical file system.

Summary of Chunk 1

Problem	Pattern	Key Idea
Implement Trie	Trie	Efficient prefix storage
Word Break	DP	`dp[i]` = can s[0:i] be segmented?
Add/Search Words	Trie + DFS	Wildcard matching via backtracking
Design File System	Trie-like Hierarchical DS	Nested dicts for paths

---