Topic Introduction

Welcome to today's PrepLetter! We're diving into a family of interview challenges that test your ability to **design complex systems** — not just algorithms, but *miniatures* of real-world applications. Today's theme? **"Designing Interactive Data Systems."**

What connects these problems? All three require you to build a *stateful* data structure that supports multiple operations, often mimicking real software (like search, filesystems, or spreadsheets). This is much more than "find the max in an array"; you'll be combining classic structures (like tries, trees, and hash maps) and thinking about how to *organize*, *update*, and *query* information efficiently.

What is a Trie? (And Why Are We Talking About It?)

A **trie** (pronounced "try" or "tree") is a special kind of tree used to store associative data structures, usually strings. Each node represents a character, and paths from the root to a node spell out a word. Tries shine when you need to do prefix-based searches or autocomplete.

- **How it works:**
  Each node has a map from possible next characters to child nodes. You insert a word by adding nodes for each character. To search, you walk down the path for each character in the prefix.

- **Why is it useful?**
  Tries make prefix search O(L), where L is the length of the prefix. This is crucial for autocomplete, spellcheckers, or dictionary-like lookups.

Example (not from the problems):

Insert "cat", "car", and "dog" into a trie:

```
(root)
  ├─ c
  │   ├─ a
  │       ├─ t *
  │       ├─ r *
  ├─ d
      ├─ o
          ├─ g *
```

(Asterisks mark complete words.)

Searching for "ca" leads to "cat" and "car" quickly, in just 2 steps.

Now, let's see how these ideas power up three more advanced system design problems.

Why Are These 3 Problems Together?

All three problems ask you to design *interactive systems* with multiple operations and efficient querying:

- **Autocomplete**: Combines trie with frequency tracking and dynamic suggestions.
- **File System**: Requires simulating directory trees and file contents with efficient path navigation.
- **Excel Sum Formula**: Demands formula parsing, dependency tracking, and on-the-fly evaluation.

They're grouped because each is a microcosm of *real-world software engineering*: modeling state, supporting updates, and handling complex queries. You'll see how the right data structures and clean interfaces make these tasks possible!

Problem 1: Design Search Autocomplete System

LeetCode 642. Design Search Autocomplete System

Problem Rephrased:

You're building an autocomplete feature. As users type a search query one character at a time, your system should instantly suggest the top 3 matching sentences, ranked by how often they've been typed before (and lex order for ties). If a user completes a query (by typing `#`), the system should record it and update its frequencies.

How it Works:

- You start with a list of historical sentences and their frequencies.
- Each call to `input(c)`:
  - If `c` is a character, update the prefix and return the top 3 suggestions.
  - If `c` is `#`, treat the current prefix as a complete query, record it, and reset.

Example:

Suppose the historical sentences are:

- "i love you" (5)
- "island" (3)
- "i love leetcode" (2)
- "ironman" (2)

User types:
`i` → Suggestions: "i love you", "i love leetcode", "island"
(space) → Suggestions: "i love you", "i love leetcode"
`a` → Suggestions: [] (no match)

Try this test case yourself:
History: ["hello world", "hi there", "hello"] with frequencies [3,2,5]
User types: `h`, `e`, `l`, `l`, `o`, `#`

- What suggestions appear after each letter?

Brute-force approach:

- For every input, scan all sentences and pick those with the current prefix.
- Sort by frequency and lex order.
- **O(N * L)** per query (N = number of sentences, L = avg sentence length).

Optimal Approach:

- Use a **trie** to store sentences, so prefix lookups are fast.
- Each trie node keeps a mapping of sentences that pass through it and their frequencies.
- When typing, walk down the trie by prefix, then retrieve and rank suggestions.

How it works step by step:

- Build a trie from all historical sentences.
- At each node, keep a dictionary of sentence frequencies.
- For each input:
  - If character: add to prefix, descend trie, get top 3 matches.
  - If #: update the trie with the new sentence.

Python Solution:

```
from collections import defaultdict, deque

class TrieNode:
    def __init__(self):
        self.children = {}
        self.counts = defaultdict(int)  # sentence: frequency

class AutocompleteSystem:
    def __init__(self, sentences, times):
        self.root = TrieNode()
        self.prefix = ""
        for sentence, freq in zip(sentences, times):
            self._insert(sentence, freq)
        self.curr = self.root  # Pointer for fast traversal

    def _insert(self, sentence, freq):
        node = self.root
        for c in sentence:
            if c not in node.children:
                node.children[c] = TrieNode()
            node = node.children[c]
            node.counts[sentence] += freq

    def input(self, c):
        if c == "#":
            self._insert(self.prefix, 1)
            self.prefix = ""
            self.curr = self.root
            return []
        self.prefix += c
        if self.curr and c in self.curr.children:
            self.curr = self.curr.children[c]
```

```
            # Get top 3 by frequency, then lex
            items = list(self.curr.counts.items())
            items.sort(key=lambda x: (-x[1], x[0]))
            return [item[0] for item in items[:3]]
        else:
            self.curr = None
            return []
```

Time Complexity:

- Insert: O(L) per sentence (L = sentence length).
- Input: O(P + S log S), where P = prefix length, S = number of sentences under the node (but limited to top 3 for output).

Space Complexity:

- O(N * L) for trie (N = number of sentences).

What does each part do?

- `TrieNode` holds children and counts of sentences passing through.
- `_insert` adds sentences to the trie and updates frequency counts at each node.
- `input` updates the current prefix, traverses down the trie, and sorts candidate sentences by frequency.

Trace Example:

Suppose trie has "i love you" (5), "i love leetcode" (2), "island" (3).

User types `i`:

- prefix = "i"
- Traverse: at node "i", counts has all 3 sentences.
- Suggestions: sort by frequency, return ["i love you", "island", "i love leetcode"]

User types " ":

- prefix = "i "
- At node "i ", counts: "i love you", "i love leetcode"
- Suggestions: ["i love you", "i love leetcode"]

Try this test case yourself:

History: ["java", "javascript", "jaws"] with frequencies [2,3,1]

Type: `j`, `a`, `v`

- What are the suggestions after each letter?

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

Reflective prompt:

Could you adapt this trie to support fuzzy search (e.g., matching with typos)? Give it a shot after finishing this!

Problem 2: Design In-Memory File System

LeetCode 588. Design In-Memory File System

Problem Rephrased:

Implement a file system that lives in memory. It should support:

- `ls(path)`: List directory contents or filename.
- `mkdir(path)`: Make new directories.
- `addContentToFile(path, content)`: Create or append to a file.
- `readContentFromFile(path)`: Return file content.

Why is this similar?

Like autocomplete, you need a **tree structure** (but this time for directories and files). You're managing a hierarchy, updating nodes, and supporting multiple operations.

Example:

- `mkdir("/a/b/c")`
- `addContentToFile("/a/b/c/d", "hello")`
- `ls("/a/b/c")` → ["d"]
- `readContentFromFile("/a/b/c/d")` → "hello"

Try this test case:

- `mkdir("/x/y")`
- `addContentToFile("/x/y/z", "foo")`
- `ls("/x/y")`
- `ls("/")`

Brute-force approach:

- Maintain a flat map of paths to content.
- For every operation, scan and parse the path.
- This is inefficient for large hierarchies and hard to update.

Optimal Approach:

Model the file system as a **tree of nodes**:

- Each node is either a directory (holds children) or file (holds content).
- To traverse a path, split it by "/", walk the tree, create nodes as needed.

Step-by-step logic:

- Each node has:
  - Name, isFile flag, children (dict), and content (for files).
- For `mkdir` and `addContentToFile`, walk down the tree, creating nodes as needed.
- For `ls`, walk to the node and return sorted child names (if dir) or filename (if file).
- For `readContentFromFile`, walk to the file node and return content.

Pseudocode:

```
class Node:
    isFile: bool
    content: str
```

```
    children: dict

class FileSystem:
    root: Node

    mkdir(path):
        walk down path, creating directories as needed

    addContentToFile(path, content):
        walk down path, create file node if needed, append content

    ls(path):
        walk down path
        if dir: return sorted list of children
        if file: return [filename]

    readContentFromFile(path):
        walk down path, return file content
```

Example Trace:

Operations:

- mkdir("/a/b")
- addContentToFile("/a/b/file.txt", "abc")
- ls("/a/b") → ["file.txt"]
- readContentFromFile("/a/b/file.txt") → "abc"

Try this test case:

- mkdir("/docs")
- addContentToFile("/docs/readme.md", "start")
- addContentToFile("/docs/readme.md", " here")
- ls("/docs")
- readContentFromFile("/docs/readme.md")

Time Complexity:

- Each operation: O(P), where P is the number of directories in the path.

Space Complexity:

- O(N), where N is the number of files and directories.

How is this similar to the previous problem?

Both use a tree structure for fast traversal and updates, but this time nodes represent directories or files, not characters.

Problem 3: Design Excel Sum Formula

LeetCode 631. Design Excel Sum Formula

Problem Rephrased:

You are building an Excel-like spreadsheet. Support these operations:

- `set(r, c, val)`: Set cell (r, c) to val.
- `get(r, c)`: Return cell value.
- `sum(r, c, refs)`: Set cell (r, c) to the sum of other cells (refs can be ranges or single cells).

Whenever a referenced cell changes, cells depending on it should update automatically.

Why is this a step up?

Now you're not just storing data, but also *tracking dependencies* and *evaluating formulas dynamically*. This is a tree/graph problem, with potential cycles and propagation.

What's different?

- You must parse references (e.g., "A1:B2") into actual cells.
- Track which cells depend on which others.
- Update cell values when dependencies change.

Optimal Approach:

- Store each cell's value and formula (if any).
- For `sum`, parse refs into cell positions, store dependencies.
- When a cell changes, propagate updates to dependents.
- Use DFS or BFS to update all downstream cells.

Pseudocode:

```
class Excel:
    cells: dict[(row, col)] -> value
    formulas: dict[(row, col)] -> list of (row, col) referenced

    set(r, c, val):
        remove formula for (r, c)
        set value
        propagate changes

    sum(r, c, refs):
        parse refs into list of (row, col)
        store formula for (r, c)
        compute sum
        propagate changes

    get(r, c):
        return current value

    propagate(r, c):
        for every cell that depends on (r, c):
```

```
            recalculate value
            propagate
```

Example Trace:

- set(1, "A", 2)
- sum(2, "B", ["A1", "A1:B1"])
  - B2 = A1 + A1 + B1
- set(1, "B", 3)
  - Updates B2 automatically

Try this test case:

- set(1, "A", 5)
- set(1, "B", 7)
- sum(2, "C", ["A1:B1"])
- set(1, "A", 10)
- get(2, "C") // Should reflect updated sum

Time Complexity:

- set: O(D), D = number of dependents (could be large for deep chains)
- sum: O(R), R = number of referenced cells
- Space: O(N + M), N = number of cells, M = number of dependencies

Nudge:
How would you avoid infinite loops if someone made a circular formula dependency? Can you detect cycles?

Summary and Next Steps

Today, you tackled three *design-style* problems that all require tree-based modeling and careful management of state and dependencies:

- **Autocomplete:** Trie with frequency tracking for fast prefix searches.
- **File System:** Directory tree for hierarchical data storage and path traversal.
- **Excel:** Table/grid with dependency tracking for dynamic formula updates.

Key patterns and insights:

- Use tree structures (trie or n-ary trees) to model hierarchical or prefix-based data.
- Manage state at each node to support efficient queries or updates.
- When dealing with formulas or dependencies, track relationships and propagate changes.

Common traps:

- Forgetting to update all references or downstream dependencies.
- Not handling edge cases (e.g., root directory, empty prefixes).
- Overcomplicating traversal — keep your tree navigation simple and robust.

Action List

- **Solve all 3 problems on your own** — even the one with code provided.
- **Try alternative approaches:** For the file system, can you do it with a flat map? For autocomplete, try a heap to maintain top suggestions.
- **Tinker with edge cases:** Empty paths, deep hierarchies, cyclic dependencies.
- **Check other "design" system problems:** Like LRU cache, Twitter clone, or Messenger.
- **Compare code styles:** Read other people's submissions to see how they structure classes and handle tricky cases.

It's totally fine to get stuck — the important part is to keep practicing and building your confidence. Happy coding!