Trie

1. Implement Trie (Prefix Tree)

Pattern: Trie (Prefix Tree)

Problem Statement

> A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the `Trie` class: - `Trie()` Initializes the trie object. - `void insert(String word)` Inserts the string `word` into the trie. - `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise. - `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.

Sample Input & Output

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

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

```
Explanation:
- Insert "apple"
- search("apple") → true
- search("app") → false (not inserted as full word)
- startsWith("app") → true (prefix of "apple")
- Insert "app"
- search("app") → true
```

```
Input: ["Trie"]
Output: [null]
Explanation: Empty initialization.
```

```
Input: ["Trie", "insert", "search", "startsWith"]
[[], ["a"], ["b"], ["b"]]

Output: [null, null, false, false]
Explanation: "b" never inserted; no word starts with "b".
```

LeetCode Editorial Solution + Inline Tests

```
class TrieNode:
    def __init__(self):
        # Each node holds:
        # - children: dict mapping char → TrieNode
        # - is_end: bool marking end of a valid word
        self.children = {}
        self.is_end = False


class Trie:
    def __init__(self):
        # STEP 1: Initialize root as empty TrieNode
        #   - Root has no char; serves as entry point
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        # STEP 2: Traverse from root, create nodes as needed
        #   - For each char, go deeper or add new node
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # STEP 3: Mark last node as end of word
        node.is_end = True

    def search(self, word: str) -> bool:
        # STEP 4: Traverse trie following word chars
        node = self.root
        for char in word:
            if char not in node.children:
                return False  # Path broken → word missing
            node = node.children[char]
        # STEP 5: Must end at node marked as word end
        return node.is_end

    def startsWith(self, prefix: str) -> bool:
        # STEP 6: Same traversal as search, but don't
        #         require is_end = True - just need path
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True  # Prefix path exists


# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    #   Test 1: Normal case
    trie = Trie()
    trie.insert("apple")
    assert trie.search("apple") == True
    assert trie.search("app") == False
    assert trie.startsWith("app") == True
    trie.insert("app")
    assert trie.search("app") == True
    print("  Test 1 passed")
```

```
#   Test 2: Edge case - empty trie
trie2 = Trie()
assert trie2.search("anything") == False
assert trie2.startsWith("x") == False
print("  Test 2 passed")

#   Test 3: Tricky/negative - overlapping words
trie3 = Trie()
trie3.insert("a")
trie3.insert("aa")
trie3.insert("aaa")
assert trie3.search("a") == True
assert trie3.search("aa") == True
assert trie3.startsWith("aaaa") == False
assert trie3.startsWith("aa") == True
print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through **Test 1** step by step:

1. **`trie = Trie()`**
   - Creates a `Trie` object.
   - Inside: `self.root = TrieNode()` → root has `children = {}`, `is_end = False`.
2. **`trie.insert("apple")`**
   - Start at `node = root`.
   - Loop over `'a'`: not in root.children → add new node. Move to it.
   - `'p'`: not in current node's children → add node. Move.
   - Next `'p'`: same → add.

    - `'l'`: add.
    - `'e'`: add.
    - After loop: mark last node (`'e'`) with `is_end = True`.
3. **`trie.search("apple")`**
    - Traverse `'a'→'p'→'p'→'l'→'e'` — all exist.
    - At `'e'` node: `is_end = True` → return `True`.
4. **`trie.search("app")`**
    - Traverse `'a'→'p'→'p'` — exists.
    - At second `'p'` node: `is_end = False` (was never set) → return `False`.
5. **`trie.startsWith("app")`**
    - Same path `'a'→'p'→'p'` exists → return `True` (no need for `is_end`).
6. **`trie.insert("app")`**
    - Traverse existing `'a'→'p'→'p'`.
    - At second `'p'` node: set `is_end = True`.
7. **`trie.search("app")` again**
    - Now `is_end = True` → returns `True`.

Final state: both `"apple"` and `"app"` are stored as complete words; prefix `"app"` matches both.

Complexity Analysis

- **Time Complexity**:
    - `insert`, `search`, `startsWith`: `O(m)`
      > Where `m` = length of the word/prefix. Each operation visits one node per character.
- **Space Complexity**: `O(N * L)`
  > Where `N` = number of inserted words, `L` = average word length. In worst case (no shared prefixes), each word uses `L` new nodes. The trie stores one node per unique character in all words.

2. Word Break

Pattern: Dynamic Programming (DP) — Linear / 1D with Set Lookup

---

Problem Statement

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.
Note that the same word in the dictionary may be reused multiple times in the segmentation.

Sample Input & Output

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

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

```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false
Explanation: No valid segmentation covers the entire string.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        # STEP 1: Initialize structures
        #   - Convert wordDict to a set for O(1) lookups.
        #   - dp[i] = True means s[0:i] can be segmented.
        word_set = set(wordDict)
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True  # Empty string is always valid

        # STEP 2: Main loop / recursion
        #   - For each end index i (1 to n), check all possible
        #     start indices j (0 to i-1).
        #   - If dp[j] is True and s[j:i] is in word_set,
        #     then s[0:i] is segmentable → set dp[i] = True.
        for i in range(1, n + 1):
            for j in range(i):
                if dp[j] and s[j:i] in word_set:
                    dp[i] = True
                    break  # No need to check other j's

        # STEP 3: Update state / bookkeeping
        #   - Handled inline in loops above.

        # STEP 4: Return result
        #   - dp[n] tells if entire string is segmentable.
        return dp[n]

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.wordBreak("leetcode", ["leet", "code"]) == True

    #  Test 2: Edge case - empty string (not tested per LeetCode,
    #          but dp[0]=True handles it gracefully)
    # LeetCode guarantees non-empty s, so skip explicit test.
    assert sol.wordBreak("a", ["a"]) == True
```

```
#   Test 3: Tricky/negative - overlapping words but no full match
assert (sol.wordBreak("catsandog", ["cats","dog","sand","and","cat"])
        == False)

print("  All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `s = "leetcode"`, `wordDict = ["leet", "code"]`.

1. **Initialization**:
   - `word_set = {"leet", "code"}`
   - `n = 8`
   - `dp = [True, False, False, False, False, False, False, False, False]` (length 9: indices 0 to 8)
2. **i = 1** → check `j=0`: `s[0:1] = "l"` → not in set → `dp[1] = False`
3. **i = 2** → `j=0`: `"le"` ; `j=1`: skip (dp[1] is False) → `dp[2] = False`
4. **i = 3** → try `j=0` ("lee"), `j=1` ("ee"), `j=2` ("e") → all → `dp[3] = False`
5. **i = 4**:
   - `j=0`: `dp[0]=True` and `s[0:4]="leet"` set →
   - Set `dp[4] = True` and break inner loop.
6. **i = 5 to 7**:
   - No `j` where `dp[j]` is True **and** `s[j:i]` in set → remain `False`
7. **i = 8**:
   - Try `j=0`: `"leetcode"`
   - `j=1..3`: `dp[j]` is False → skip
   - `j=4`: `dp[4]=True`, check `s[4:8]="code"` → → set `dp[8] = True`
8. **Return** `dp[8] = True`

Final state:
`dp = [T, F, F, F, T, F, F, F, T]` → returns `True`.

Complexity Analysis

- **Time Complexity**: $O(n^2)$

  Outer loop runs `n` times. Inner loop runs up to `i` times (worst-case `n`).
  Each substring `s[j:i]` takes `O(i-j)` to create and hash, but in practice,
  average word length is bounded → often treated as $O(n^2)$ with small constant.
  Strictly: $O(n^3)$ in worst-case (e.g., all substrings checked), but acceptable for
  `n  300`.

- **Space Complexity**: `O(n + m)`

  `dp` array uses `O(n)`. `word_set` uses `O(m)` where `m = len(wordDict)`.
  Substrings are temporary but contribute to time, not persistent space.

3. Design Add and Search Words Data Structure

Pattern: Trie (Prefix Tree) + Backtracking (for wildcard support)

Problem Statement

Design a data structure that supports adding new words and finding if a string matches any previously added string.
Implement the `WordDictionary` class:
- `WordDictionary()` initializes the object.
- `void addWord(word)` Adds `word` to the data structure.
- `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise.

`word` may contain dots `'.'` where dots can match any letter.

Sample Input & Output

```
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]
Explanation:
- "pad" is not in the dictionary → false
- "bad" was added → true
- ".ad" matches "bad", "dad", or "mad" → true
- "b.." matches "bad" → true
```

```
Input: ["WordDictionary", "addWord", "search"]
       [[], ["a"], ["."]]
Output: [null, null, true]
Explanation: "." matches the single letter "a".
```

```
Input: ["WordDictionary", "addWord", "search", "search"]
       [[], ["abc"], ["ab"], ["abcd"]]
Output: [null, null, false, false]
Explanation: Partial and overlong queries don’t match.
```

LeetCode Editorial Solution + Inline Tests

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

class WordDictionary:
    def __init__(self):
        # STEP 1: Initialize root of the Trie
        #   - Root is an empty node that branches to first letters
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        # STEP 2: Insert word into Trie
        #   - Traverse character by character
        #   - Create new nodes as needed
        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  # Mark end of valid word

    def search(self, word: str) -> bool:
        # STEP 3: Search with support for '.'
        #   - Use DFS/backtracking to handle wildcards
        #   - Try all children when encountering '.'
        def dfs(node, i):
            # Base case: reached end of word
            if i == len(word):
                return node.is_end

            char = word[i]
            if char == '.':
                # Try every child path
                for child in node.children.values():
                    if dfs(child, i + 1):
                        return True
                return False
            else:
                # Exact match required
                if char not in node.children:
                    return False
                return dfs(node.children[char], i + 1)

        return dfs(self.root, 0)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    #  Test 1: Normal case
    wd = WordDictionary()
    wd.addWord("bad")
    wd.addWord("dad")
    wd.addWord("mad")
    assert wd.search("pad") == False
    assert wd.search("bad") == True
    assert wd.search(".ad") == True
    assert wd.search("b..") == True
    print("  Test 1 passed")

    #   Test 2: Edge case - single letter with wildcard
    wd2 = WordDictionary()
    wd2.addWord("a")
    assert wd2.search(".") == True
    assert wd2.search("a") == True
    assert wd2.search("aa") == False
    print("  Test 2 passed")

    #   Test 3: Tricky/negative - partial match & overlong
    wd3 = WordDictionary()
    wd3.addWord("abc")
    assert wd3.search("ab") == False      # not a complete word
    assert wd3.search("abcd") == False    # longer than any word
    assert wd3.search("a.c") == True      # exact wildcard match
    assert wd3.search(".b.") == True
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `search(".ad")` after adding `"bad"`, `"dad"`, `"mad"`:

1. **Start at root** (`i = 0`, char = `'.'`)
   - Wildcard → loop through all children: `'b'`, `'d'`, `'m'`
   - Try `'b'` branch first
2. **At `'b'` node** (`i = 1`, char = `'a'`)
   - `'a'` is in `'b'`'s children → move to `'a'` node
3. **At `'a'` node** (`i = 2`, char = `'d'`)
   - `'d'` is in `'a'`'s children → move to `'d'` node
4. **At `'d'` node** (`i = 3`, end of word)

- Check `is_end`: `True` (since `"bad"` was added)
- Return `True` → propagate up

Match found on first wildcard branch (`'b'`). No need to try `'d'` or `'m'`.

Now try `search("pad")`:

1. **Root** → look for `'p'` in children → not present → return `False` immediately.

This shows how the Trie enables early termination on mismatches, while backtracking handles wildcards by exploring all possibilities.

Complexity Analysis

- **Time Complexity**:
  - `addWord`: `O(L)` where `L` = word length (one pass through Trie)
  - `search`: `O(26^L)` in worst case (e.g., word = `"...."` with depth `L`)
    > In practice, much faster due to early pruning. For non-wildcard searches, it's `O(L)`.
- **Space Complexity**: `O(N * L)`
  > Where `N` = number of words, `L` = average word length. Each character may create a new `TrieNode`. Worst-case no shared prefixes.

4. Design In-Memory File System

Pattern: Trie + Hash Map (Hierarchical Data Structure)

Problem Statement

Design a in-memory file system to simulate the following functions:

- `ls`: Given a path in string format. If it is a file path, return a list that only contains this file's name. If it is a directory path, return the list of file and directory names **in this directory**. The answer should be in **lexicographic order**.

- `mkdir`: Given a directory path that does not exist, you should make a new directory according to the path. If the middle directories in the path do not exist, you should create them as well.
- `addContentToFile`: Given a file path and file content in string format. If the file doesn't exist, you need to create that file containing the given content. If the file already exists, you need to **append** the given content to the original content.
- `readContentFromFile`: Return the content in the file at the given path.

Assumptions: - All paths are absolute (start with `/`). - Path components are separated by `/`. - No file or directory name contains `/`. - No duplicate names in the same directory.

Sample Input & Output

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

Output:
[null, [], null, null, ["a"], "hello"]

Explanation:
- Initially, root is empty → `ls("/")` returns [].
- `mkdir("/a/b/c")` creates nested dirs.
- `addContentToFile("/a/b/c/d", "hello")` creates file "d" with content.
- `ls("/")` now returns ["a"] (only top-level dir).
- `readContentFromFile("/a/b/c/d")` returns "hello".
```

```
Input: ["FileSystem", "mkdir", "ls"]
[[], ["/zijzll"], ["/"]]
Output: [null, null, ["zijzll"]]
```

```
Input: ["FileSystem", "addContentToFile", "ls", "readContentFromFile"]
[[], ["/file", "content"], ["/file"], ["/file"]]
Output: [null, null, ["file"], "content"]
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, Dict

class FileSystem:
    def __init__(self):
        # Each node is a dict with:
        #   - 'is_file': bool
        #   - 'content': str (if file)
        #   - 'children': dict (if dir)
        self.root = {'is_file': False, 'content': '', 'children': {}}

    def _get_node(self, path: str):
        # Traverse to the node at given path; create intermediates if needed.
        if path == "/":
            return self.root
        parts = path.split("/")[1:]  # Skip leading empty string
        node = self.root
        for part in parts:
            if part not in node['children']:
                node['children'][part] = {
                    'is_file': False,
                    'content': '',
                    'children': {}
                }
            node = node['children'][part]
        return node

    def ls(self, path: str) -> List[str]:
        node = self._get_node(path)
        if node['is_file']:
            # Return just the filename (last part of path)
            return [path.split("/")[-1]]
        # Return sorted list of children names
        return sorted(node['children'].keys())

    def mkdir(self, path: str) -> None:
        self._get_node(path)  # Ensures path exists

    def addContentToFile(self, filePath: str, content: str) -> None:
        node = self._get_node(filePath)
        node['is_file'] = True
        node['content'] += content

    def readContentFromFile(self, filePath: str) -> str:
        node = self._get_node(filePath)
        return node['content']

# ------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    #  Test 1: Normal case
    fs = FileSystem()
    fs.mkdir("/a/b/c")
    fs.addContentToFile("/a/b/c/d", "hello")
    assert fs.ls("/") == ["a"]
    assert fs.readContentFromFile("/a/b/c/d") == "hello"
    print("  Test 1 passed")

    #  Test 2: Edge case - root listing after file creation
    fs2 = FileSystem()
    fs2.addContentToFile("/file", "content")
    assert fs2.ls("/") == ["file"]
    assert fs2.ls("/file") == ["file"]
    print("  Test 2 passed")

    #  Test 3: Tricky/negative - deeply nested, lexicographic order
    fs3 = FileSystem()
    fs3.mkdir("/x/y")
    fs3.mkdir("/a/b")
    fs3.addContentToFile("/x/y/f1", "1")
    fs3.addContentToFile("/a/b/f2", "2")
    assert fs3.ls("/") == ["a", "x"]  # lex order
    assert fs3.ls("/x/y") == ["f1"]
    print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1** step by step:

1. **`fs = FileSystem()`**
    - Creates `self.root = {'is_file': False, 'content': '', 'children': {}}`
    - State: empty root directory.
2. **`fs.mkdir("/a/b/c")`**
    - Calls `_get_node("/a/b/c")`
    - Splits path → `["a", "b", "c"]`
    - Starts at root.
        - `"a"` not in root → creates node for `"a"`
        - `"b"` not in `"a"`'s children → creates node for `"b"`
        - `"c"` not in `"b"`'s children → creates node for `"c"`
    - Final structure:

      ```
      root
        a
           b
              c (dir, empty children)
      ```
3. **`fs.addContentToFile("/a/b/c/d", "hello")`**
    - `_get_node` traverses to `/a/b/c/d`
        - `"d"` not in `"c"`'s children → creates node for `"d"`
    - Sets `node['is_file'] = True`
    - Sets `node['content'] = "" + "hello" = "hello"`
    - Now `"d"` is a file with content.
4. **`fs.ls("/")`**
    - Gets root node → not a file → returns sorted keys of `root['children']` → `["a"]`
5. **`fs.readContentFromFile("/a/b/c/d")`**
    - Traverses to `"d"` node → returns `"hello"`

All assertions pass.

Complexity Analysis

- **Time Complexity**:
  - `ls`: `O(m + k log k)`
    > `m` = path depth (to traverse), `k` = number of children (to sort).

  - `mkdir`, `addContentToFile`, `readContentFromFile`: `O(m)`
    > Only path traversal; no sorting.
- **Space Complexity**: `O(N)`
  > `N` = total number of directories and files. Each path component stored once in trie-like structure.