Topic Introduction

Today's PrepLetter dives into **Word Segmentation using Dynamic Programming**—a classic pattern for problems involving breaking up strings into meaningful words. These problems regularly appear in coding interviews because they test your understanding of dynamic programming (DP), recursion, memoization, and sometimes trie data structures.

What is Word Segmentation?

Word segmentation is the act of splitting a string into a sequence of dictionary words. For example, given `"catsanddog"` and a dictionary `["cat", "cats", "and", "sand", "dog"]`, a valid segmentation is `["cats", "and", "dog"]`.

How does it work?

At its heart, word segmentation is about checking if prefixes of a string exist in a dictionary, then recursively (or iteratively) checking if the remaining suffix can also be split. Dynamic programming or memoization helps avoid redundant recalculations.

Why is it useful in interviews?

- It tests your ability to recognize overlapping subproblems.
- It requires careful handling of indices and substrings.
- It can be extended to more complex variations (like returning all possible splits, not just one).

Simple Example (Not one of today's problems):

Suppose you have the string `"applepie"` and the dictionary `["apple", "pie"]`. Is it possible to split `"applepie"` into words from the dictionary?

Yes: `"apple"` + `"pie"`.

This can be solved by checking prefixes, and using DP to cache solutions for suffixes.

Introducing Today's Problems

We'll explore three problems that all revolve around segmenting words using dictionaries:

- **Word Break** (Leetcode 139): Can the string be split into valid dictionary words?
- **Word Break II** (Leetcode 140): Return *all* possible ways to segment the string into dictionary words.
- **Concatenated Words** (Leetcode 472): Given a word list, find all words that can be formed by concatenating at least two other words from the list.

Why are these grouped together?

All three require breaking up strings using a dictionary, often using dynamic programming or tries for efficiency. The first checks if a split exists, the second finds all splits, and the third works in reverse—finding words that are themselves built from other words.

Problem 1: Word Break

Problem Statement (in simple terms):

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

Problem link

Example:

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

Let's walk through another example:

Input:
`s = "applepenapple"`
`wordDict = ["apple", "pen"]`
Output: `True`
Explanation: `"apple pen apple"` is valid.

Pen-and-paper encouragement:
Try breaking up `"catsandog"` with dictionary `["cats","dog","sand","and","cat"]`.

Brute-force approach:

Try all possible ways to split the string at every position and check if both parts are valid recursively.
Time Complexity: Exponential, O(2^n), where n is the length of `s`.

Optimal approach: Dynamic Programming

- **Core Pattern:**

Use a DP array where `dp[i]` is True if `s[0:i]` can be segmented.

- **Step-by-step Logic:**
  - Create a boolean array `dp` of size `len(s)+1`, initialize `dp[0] = True` (empty string).
  - For each index `i` from 1 to `len(s)`:
    - For each `j` from 0 to `i`:
      - If `dp[j]` is True and `s[j:i]` is in the dictionary, set `dp[i] = True`.
      - Break inner loop if you find a valid split.
  - Return `dp[len(s)]`.

Here's the code:

```
def wordBreak(s, wordDict):
    # Convert wordDict to a set for O(1) lookups
    word_set = set(wordDict)
    n = len(s)
    # dp[i]: True if s[0:i] can be segmented
    dp = [False] * (n + 1)
```

```
    dp[0] = True  # Empty string is always "breakable"

    for i in range(1, n + 1):
        for j in range(i):
            # If s[0:j] can be broken and s[j:i] is in wordDict
            if dp[j] and s[j:i] in word_set:
                dp[i] = True
                break  # No need to check further splits

    return dp[n]
```

Time Complexity: O(n^2) (two nested loops for all substrings)
Space Complexity: O(n) (for the dp array)

Explanation of the code:

- We use `dp[i]` to mean: can the substring `s[0:i]` be segmented?
- For every possible end index `i`, we look at all possible split points `j`.
- If we've found that `s[0:j]` can be segmented and `s[j:i]` is a word, we mark `dp[i]` as True.
- At the end, `dp[n]` tells us if the entire string is segmentable.

Trace on a test case:

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

- `dp[0]=True` (empty)
- i=4: `s[0:4]="leet"` in dict, `dp[4]=True`
- i=8: `s[4:8]="code"` in dict, and `dp[4]=True`, so `dp[8]=True`

So, return `dp[8]=True`.

Try this test case yourself:

Input:
`s = "catsandog"`
`wordDict = ["cats","dog","sand","and","cat"]`

What does the DP table look like? What is the answer?

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

Problem 2: Word Break II

Problem Statement (rephrased):

Given a string `s` and a dictionary, return all possible sentences where `s` can be segmented into a sequence of dictionary words.

Problem link

Example:

Input:

```
s = "catsanddog"
wordDict = ["cat","cats","and","sand","dog"]
```

Output:

```
["cats and dog", "cat sand dog"]
```

Explanation:

- `"cats and dog"` is possible (`"cats"`, `"and"`, `"dog"`).
- `"cat sand dog"` is also possible (`"cat"`, `"sand"`, `"dog"`).

How is this different from Problem 1?

Instead of a simple True/False, we need to find *all ways* to break up the string and return the actual sentences.

Brute-force approach:

Try every possible way to split the string, recursively generate all splits, and collect valid ones.

Time Complexity: Exponential (very slow for long strings).

Optimal approach: DP with Memoization (Top-Down)

- **Core Pattern:**

Use recursion to try every split, and memoize results to avoid recomputation.

- **Step-by-step Logic:**
  - Define a helper function that, given a substring, returns all possible sentences.
  - For each prefix of the substring, if it's in the dictionary, recursively solve for the suffix.
  - Combine prefix and each result from the suffix.
  - Memoize the results for each substring to speed up repeated calls.

Pseudocode:

```
function wordBreakII(s, wordDict):
    memo = dict()
    function dfs(start):
        if start == len(s): return [""]
        if start in memo: return memo[start]
        results = []
        for end in range(start+1, len(s)+1):
            word = s[start:end]
            if word in wordDict:
                sub_sentences = dfs(end)
                for sub in sub_sentences:
                    if sub == "":
                        results.append(word)
                    else:
                        results.append(word + " " + sub)
        memo[start] = results
```

```
        return results
    return dfs(0)
```

Example Trace:

With `s = "catsanddog"`, `wordDict = ["cat","cats","and","sand","dog"]`:

- From position 0:
  - "cat" -> recursively solve "sanddog"
  - "cats" -> recursively solve "anddog"
- Recursion builds all combinations.

Try this test case:

Input:

```
s = "pineapplepenapple"
wordDict = ["apple","pen","applepen","pine","pineapple"]
```

Expected output:

```
["pine apple pen apple", "pineapple pen apple", "pine applepen apple"]
```

Step-by-step:

- At each index, try every prefix.
- If the prefix is a word, recursively solve the remaining string.
- Combine results with spaces.

Time Complexity:

Worst-case exponential, but memoization prunes many repeated subproblems.

Space Complexity:

O(N) for memoization, plus space for output.

Go through this trace:

For `"catsanddog"`:

- Start at 0: "cat" and "cats" are valid.
- "cat" leaves "sanddog": "sand" + "dog" is possible.
- "cats" leaves "anddog": "and" + "dog" is possible.

So, result: `["cat sand dog", "cats and dog"]`.

Try this yourself:

Input:

```
s = "catsandog"
wordDict = ["cats","dog","sand","and","cat"]
```

Can you find any valid sentences?

Problem 3: Concatenated Words

Problem Statement:

Given a list of words, return all words that are formed by concatenating at least two smaller words from the same list.

Problem link

How is this more challenging?

Now, instead of checking if a *given* string can be segmented, you check *every word* in the list to see if it can be built from other words (not itself). This is a "reverse" of the Word Break problem.

Brute-force approach:

For each word, try all possible splits and check if both parts are in the list or can be formed by concatenation.
Time Complexity: O(n * 2^L), where n = number of words, L = max word length.

Optimal approach: DP with DFS and Set

- **Core Pattern:**

For each word, use DFS with memoization to check if it can be segmented into two or more shorter words.
Be careful: don't use the word itself as a building block.

- **Step-by-step Guidance:**
    - Put all words in a set for O(1) lookup.
    - For each word in the list:
        - Temporarily remove the word from the set (so it doesn't use itself).
        - Use DFS (or DP) to check if the word can be segmented into at least two words from the set.
        - Add the word back to the set after checking.
        - If so, add the word to the output list.

Pseudocode:

```
function findAllConcatenatedWords(words):
    wordSet = set(words)
    result = []

    function canForm(word):
        if word in memo: return memo[word]
        for i from 1 to len(word)-1:
            prefix = word[0:i]
            suffix = word[i:]
            if prefix in wordSet:
                if suffix in wordSet or canForm(suffix):
                    memo[word] = True
                    return True
        memo[word] = False
        return False
```

```
    for word in words:
        if word == "": continue
        remove word from wordSet
        if canForm(word): result.append(word)
        add word back to wordSet

    return result
```

Try this example:

Input:
`words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]`

Expected output:
`["catsdogcats","dogcatsdog","ratcatdogcat"]`

Step-by-step:

- For `"catsdogcats"`: can be split as `"cats" + "dog" + "cats"`.
- For `"dogcatsdog"`: `"dog" + "cats" + "dog"`.
- For `"ratcatdogcat"`: `"rat" + "cat" + "dog" + "cat"`.

Time Complexity:
O(n * L^2), where n = number of words, L = word length (checking all splits per word).

Space Complexity:
O(n * L) for memoization and the word set.

Try this yourself:

Input:
`words = ["cat", "dog", "catdog"]`

What should the output be?

Reflect:
Can you think of a way to use a Trie to speed this up for very large input?

Summary and Next Steps

Today, you tackled three classic string segmentation problems, all relying on dynamic programming and memoization:

- **Word Break** checks if a string is breakable into dictionary words (True/False).
- **Word Break II** finds all possible ways to break up the string.
- **Concatenated Words** searches a list for words that are themselves made up of other words in the list.

Key patterns and insights:

- DP arrays (bottom-up or top-down memoization) are lifesavers for overlapping subproblems in string segmentation.
- For "all possible sentences" (Word Break II), recursion with memoization is critical for efficiency.
- For "concatenated words," remember to avoid using the word itself as a building block—remove it from the set temporarily!
- Be careful with off-by-one errors in indices and substring extraction.
- Trie structures can sometimes optimize dictionary lookups, especially for large word lists.

Common mistakes:

- Forgetting to memoize recursive calls, leading to exponential runtimes.
- Not handling empty strings correctly.
- Accidentally allowing a word to use itself in concatenation.

Action List

- Solve all three problems on your own—even the one with code provided.
- Try implementing Problem 2 and Problem 3 using a Trie instead of a set.
- Explore related problems like "Palindrome Partitioning" for more segmentation practice.
- Compare your solution with others, paying attention to edge cases.
- If you get stuck, revisit the DP pattern and work out small examples by hand.

Keep practicing, and soon you'll spot these patterns instantly in interviews. Happy coding!