Documentation for Longest Word in Dictionary through Deleting

1. □ **Problem Statement**

   Given:

   - A string s.
   - A word dictionary.

   Task:
   Return the longest word in the dictionary that can be formed by deleting some characters of s.

   - If there are multiple answers, return the lexicographically smallest one.
   - If no valid word exists, return an empty string.

2. 💡 **Intuition**

   To find a word from the dictionary that exists as a subsequence of the main string s, we can:

   - Check each word using a two-pointer method to see if it's a valid subsequence.
   - Prioritize longer and lexicographically smaller words by sorting the dictionary.

3. 🔍 **Key Observations**

   - The task is essentially about subsequence matching.
   - We must process every word in the dictionary.
   - Sorting helps us quickly find the correct match without having to compare every match manually.

4. 🛠 **Approach**

   - Define a helper function is_subsequence(word):
     - Use two pointers to check if the word can be formed by deleting characters in s.

- Sort the dictionary:
    - Primary key: -len(word) → longest first.
    - Secondary key: word → lexicographically smallest first.
- Iterate over the sorted dictionary:
    - Return the first word that is a subsequence of s.

5. **⚠ Edge Cases**

- Dictionary is empty → return "".
- No word in the dictionary is a subsequence → return "".
- All words are of the same length. → Pick the one with the smallest lexicographical order.

6. **□ Complexity Analysis**

Time Complexity

- Sorting: O(n * log n) where n = len(dictionary)
- Subsequence check: For each word (length m), check against s (length k) → O(m + k) in worst case.
- Total:
  O(n * log n + n * k) where k = len(s)

Space Complexity

- Sorting and storage: O(n)
- No extra space used other than minor variables → O(1) auxiliary space.

7. **🔁 Alternative Approaches**

- Trie-based optimization: Build a trie from the dictionary and perform DFS over s to match words.
- Indexed mapping of characters in s: Store indices of characters in s and binary search to speed up subsequence checking.

8. □ **Test Cases**

```
# Test Case 1
    s = "abpcplea"
    dictionary = ["ale", "apple", "monkey", "plea"]
    # Output: "apple"

# Test Case 2
    s = "abpcplea"
    dictionary = ["a", "b", "c"]
    # Output: "a"

# Test Case 3
    s = "abcd"
    dictionary = ["abcde"]
    # Output: ""

# Test Case 4
    s = "bab"
    dictionary = ["ba","ab","a","b"]
    # Output: "ab" (both 'ab' and 'ba' are valid, but 'ab' is smaller)
```

9. □ **Final Thoughts**

- This problem teaches subsequence checking, sorting with custom keys, and greedy selection.
- Sorting the dictionary upfront saves extra logic for handling tiebreakers.
- Optimizations may be required for large-scale input, especially if s and dictionary words are near the upper bounds.