720. Longest Word in Dictionary

Given an array of strings `words` representing an English Dictionary, return the longest word in `words` that can be built one character at a time by other words in `words`.

If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.

Note that the word should be built from left to right with each additional character being added to the end of a previous word.

Example 1:

Input: `words = ["w","wo","wor","worl","world"]`

Output: `"world"`

Explanation: The word `"world"` can be built one character at a time by `"w"`, `"wo"`, `"wor"`, and `"worl"`.

Example 2:

Input: `words = ["a","banana","app","appl","ap","apply","apple"]`

Output: `"apple"`

Explanation: Both `"apply"` and `"apple"` can be built from other words in the dictionary. However, `"apple"` is lexicographically smaller than `"apply"`.

Constraints:

- $1 \leq$ `words.length` ≤ 1000
- $1 \leq$ `words[i].length` ≤ 30
- `words[i]` consists of lowercase English letters.

Solution

Brute Force

For this problem, we're asked to find the longest word with smallest lexicographical order such that all its non-empty prefixes exist in `words`. Let's call a word **good** if all its prefixes exist in `words`. We can verify if this is the case by iterating through all prefixes to check if they all exist. Let's denote L as the length of the longest **good** word. Out of all **good** words with length L, we'll return the one with lexicographically least length.

Full Solution

Let's denote x_i as the length of `words[i]`.

We can observe that `words[i]` is **good** if $x_i = 1$ or the prefix of `words[i]` with length $x_i - 1$ is **good**.

Let's try to find a way to use this idea to process all words efficiently. Since processing a word with length x_i requires a word with length $x_i - 1$ to be processed, we should process words by **non-decreasing** length. This can be done by sorting and simply iterating through the sorted list. In addition, we'll use a hashmap to act as a lookup table for **good** words.

In our algorithm, we'll iterate through words by **non-decreasing** length. For each word, we'll check if it's **good** with the method mentioned above. If the word is **good**, we'll update it in our hashmap.

Time Complexity

Let's denote N as the length of `words` and S as the sum of lengths of all words in `words`.

Since sorting takes $\mathcal{O}(N \log N)$ and our main algorithm takes $\mathcal{O}(S)$ from comparing keys, our final time complexity is $\mathcal{O}(N \log N + S)$.

Time Complexity: $\mathcal{O}(N \log N + S)$

Space Complexity

Since our hashmap has $\mathcal{O}(N)$ memory, our space complexity is $\mathcal{O}(N)$.

Space Complexity: $\mathcal{O}(N)$

C++ Solution

```cpp
class Solution {
   public:
    static bool comp(string s, string t) {  // sorting comparator
        return s.size() < t.size();
    }
    string longestWord(vector<string>& words) {
        sort(words.begin(), words.end(), comp);  // sort words by non-decreasing length
        unordered_map<string, bool> goodWords;        // lookup for good words
        int maxLength = 0;
        string ans = "";
        for (string word : words) {
            if (word.size() == 1) {
                goodWords[word] = true;
            } else if (goodWords[word.substr(0, word.size() - 1)]) {  // word with length - 1 prefix is good
                goodWords[word] = true;
            }
            if (goodWords[word]) {
                if (maxLength < word.size()) {  // find longer word
                    maxLength = word.size();
                    ans = word;
                } else if (maxLength == word.size()) {  // find lexicographically smaller word
                    ans = min(ans, word);
                }
            }
        }
        return ans;
    }
};
```

Java Solution

```java
class Solution {
    public String longestWord(String[] words) {
        Arrays.sort(words, (a, b) -> a.length() - b.length()); // sort words by non-decreasing length
        HashMap<String, Boolean> goodWords = new HashMap(); // lookup for good words
        int maxLength = 0;
        String ans = "";
        for (String word : words) {
            if (word.length() == 1) {
                goodWords.put(word, true);
            } else if (goodWords.containsKey(word.substring(0, word.length() - 1))) {
                // word with length - 1 prefix is good
                goodWords.put(word, true);
            }
            if (goodWords.containsKey(word)) {
                if (maxLength < word.length()) { // find longer word
                    maxLength = word.length();
                    ans = word;
                } else if (maxLength == word.length()
                    && ans.compareTo(word) > 0) { // find lexicographically smaller word
                    ans = word;
                }
            }
        }
        return ans;
    }
}
```

Python Solution

Note: A set can be used in python which acts as a hashset and serves the same purpose as a hashmap in this solution.

```python
class Solution:
    def longestWord(self, words: List[str]) -> str:
        words.sort(key=len)  # sort words by non-decreasing length
        goodWords = set()  # lookup for good words
        maxLength = 0
        ans = ""
        for word in words:
            if len(word) == 1:
                goodWords.add(word)
            elif word[:-1] in goodWords:  # word with length - 1 prefix is good
                goodWords.add(word)
            if word in goodWords:
                if maxLength < len(word):  # find longer word
                    maxLength = len(word)
                    ans = word
                elif maxLength == len(word):  # find lexicographically smaller word
                    ans = min(ans, word)
        return ans
```