Concatenated Words

LeetCode Admin Nov 17, 2022

Solution

The main logic of the solutions is similar to the question Word Break.

Approach 1: Dynamic Programming

Intuition

Consider the word list as a dictionary, then the problem is: which words can be created by concatenating two or more words in the dictionary?

This is a famous "reachability" problem and it can be solved by DP(dynamic programming), BFS and/or DFS. Though they seem to be different, the core idea behind them is the same. Namely, if a given `word` can be created by concatenating the given words, we can split it into 2 parts, the prefix and the suffix, the prefix is a shorter word which can be got by concatenating the given words and the suffix is a given word.

Namely, `word` = (another shorter word that can be created by concatenation) + (a given word in the dictionary). We can enumerate the suffix and look it up in the dictionary and the prefix part is just a sub-problem to solve.

Algorithm

State definition

Formally, for each word, let's define the sub-problem as whether a (possibly empty) prefix can be created by concatenation. So the state of the dynamic programming algorithm can be defined as a boolean array: let $dp[i]$ denote whether word's prefix of length i (index range [0, i - 1]) can be created by concatenation.

Induction

We need to calculate $dp[i]$ for each i in range [0, word.length]. Let's do it by induction.

The base case is simple: $dp[0]$ = true, since it's the empty string that can always be created without using any words in the dictionary.

Now, let's consider the value of $dp[i]$ for i > 0.

If $dp[i]$ is true, as mentioned before, we can split this prefix into 2 parts, a prefix of length j < i which can be created by the words in the dictionary, and the remaining suffix which is exactly a single word in the dictionary.

$dp[i]$ is true if and only if there is an integer j, such that 0 <= j < i and the word's substring (index range [j, i - 1]) is in the dictionary.

Note: There is an corner case, when i == length, since we don't want to use the word in the dictionary directly, we should check 1 <= j < i instead.

The answer

$dp[word.length]$ tells if the word can be created by concatenation.

Here is how the algorithm works with "catsdogcats" if we have "cats" and "dog" in the dictionary.

Index	0	1	2	3	4	5	6	7	8	9	10	11
word[index - 1]		c	a	t	s	d	o	g	c	a	t	s
dp[index]	true	false	false	false	true	false	false	true	false	false	true	true
Note	Base case				dp[0]=true && the suffix “cats” is a word in the list			dp[4] = true && the suffix “dog” is a word in the list				dp[7] = true && the suffix “cats” is a word in the list

For instance, $dp[7]$ tells if we can create "catsdogs" (the first 7 letters). It's true because we can split it into a prefix "cats" which we know we can create because $dp[4]$ is true, and a suffix "dogs", which is in dictionary.

Steps

1. Put all the words into a HashSet as a `dictionary`.
2. Create an empty list `answer`.
3. For each `word` in the `words` create a boolean array `dp` of length = `word.length + 1`, and set dp[0] = true.
4. For each index `i` from 1 to `word.length`, set `dp[i]` to true if we can find a value `j` from 0 (1 if i == `word.length`) such that dp[j] = true and `word.substring(j, i)` is in the `dictionary`.
5. Put `word` into `answer` if dp[word.length] = true.
6. After processing all the `words`, return `answer`.

Implementation

C++ Java Copy

```java
class Solution {
    public List<String> findAllConcatenatedWordsInADict(String[] words) {
        final Set<String> dictionary = new HashSet<>(Arrays.asList(words));
        final List<String> answer = new ArrayList<>();
        for (final String word : words) {
            final int length = word.length();
            final boolean[] dp = new boolean[length + 1];
            dp[0] = true;
            for (int i = 1; i <= length; ++i) {
                for (int j = (i == length ? 1 : 0); !dp[i] && j < i; ++j) {
                    dp[i] = dp[j] && dictionary.contains(word.substring(j, i));
                }
            }
            if (dp[length]) {
                answer.add(word);
            }
        }
        return answer;
    }
}
```

Complexity Analysis

Here, N is the total number of strings in the array `words`, namely `words.length`, and M is the length of the longest string in the array `words`.

- Time complexity: $O(M^3 \cdot N)$.

Although we use HashSet, we need to consider the cost to calculate the hash value of a string internally which would be $O(M)$. So putting all words into the HashSet takes $O(N * M)$. For each word, the i and j loops take $O(M^2)$. The internal logic to take the substring and search in the HashSet needs to calculate the hash value for the substring too, and it should take another $O(M)$, so for each word, the time complexity is $O(M^3)$ and the total time complexity for N words is $O(M^3 \cdot N)$

- Space complexity: $O(N \cdot M)$.

This is just the space to save all words in the `dictionary`, if we don't take M as a constant.

Approach 2: DFS

Intuition

As mentioned before, this problem can be transformed into a reachability problem and thus can be solved by a DFS (or BFS) algorithm. For each word, we construct a directed graph with all prefixes as nodes. For simplicity, we can represent each prefix by its length.

So the graph contains (word.length + 1) nodes. For edges, consider 2 prefixes i and j with 0 <= i < j <= word.length, if prefix j can be created by concatenating prefix i and a word in the dictionary, we add a directed edge from node i to node j.

> When i = 0, we require `j < word.length` as there should be an edge from node `0` to node `word.length`. Determining whether a word can be created by concatenating 2 or more words in the dictionary is the same as determining whether there is a path from node `0` to node `word.length` in the graph.

Algorithm

For each word, construct the implicit graph mentioned above, then add it to the answer if the node `word.length` can be reached from node `0` in the graph which can be checked using DFS.

Implementation

C++ | Java

Copy

```java
class Solution {
    private boolean dfs(final String word, int length, final boolean[] visited, final Set<String> dictionary) {
        if (length == word.length()) {
            return true;
        }
        if (visited[length]) {
            return false;
        }
        visited[length] = true;
        for (int i = word.length() - (length == 0 ? 1 : 0); i > length; --i) {
            if (dictionary.contains(word.substring(length, i))
                && dfs(word, i, visited, dictionary)) {
                return true;
            }

        }
        return false;
    }

    public List<String> findAllConcatenatedWordsInADict(String[] words) {
        final Set<String> dictionary = new HashSet<>(Arrays.asList(words));
        final List<String> answer = new ArrayList<>();
        for (final String word : words) {
            final int length = word.length();
            final boolean[] visited = new boolean[length];
            if (dfs(word, 0, visited, dictionary)) {
                answer.add(word);
```

Complexity Analysis

Here, N is the total number of strings in the array `words`, namely `words.length`, and M is the length of the longest string in the array `words`.

- Time complexity: $O(M^3 \cdot N)$.

For each word, the constructed graph has M nodes and $O(M^2)$ edges, and the DFS algorithm for reachability is $O(M^2)$ without considering the time complexities of substring and HashSet. If we consider everything, the time complexity to check one word is $O(M^3)$ and the total time complexity to check all words is $O(M^3 \cdot N)$.

- Space complexity: $O(N \cdot M)$.

This is the space to save all words in the `dictionary` , if we don't take M as a constant, there is also $O(M)$ for the call stack to execute DFS, which wouldn't affect the space complexity anyways.

Share

Comments (33)

Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

💡 Article Commenting Rules

1. This comment section is for questions and comments regarding this **LeetCode article**. All posts must respect our LeetCode Community Rules.

2. Concerns about errors or bugs in the article, problem description, or test cases should be posted on LeetCode Feedback, so that our team can address them.

Thunder…

Jan 27, 2023