Problem 126: Word Ladder II

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

A

transformation sequence

from word

beginWord

to word

endWord

using a dictionary

wordList

is a sequence of words

beginWord -> s

1

-> s

2

-> ... -> s

k

such that:

Every adjacent pair of words differs by a single letter.

Every

s

i

for

1 <= i <= k

is in

wordList

. Note that

beginWord

does not need to be in

wordList

.

s

k

== endWord

Given two words,

beginWord

and

endWord

, and a dictionary

wordList

, return

all the

shortest transformation sequences

from

beginWord

to

endWord

, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words

[beginWord, s

1

, s

2

, ..., s

k

]

.

Example 1:

Input:

beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output:

[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

Explanation:

There are 2 shortest transformation sequences: "hit" -> "hot" -> "dot" -> "dog" -> "cog" "hit" -> "hot" -> "lot" -> "log" -> "cog"

Example 2:

Input:

beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

Output:

[]

Explanation:

The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Constraints:

1 <= beginWord.length <= 5

endWord.length == beginWord.length

1 <= wordList.length <= 500

wordList[i].length == beginWord.length

beginWord

,

endWord

, and

wordList[i]

consist of lowercase English letters.

beginWord != endWord

All the words in

wordList

are

unique

.

The

sum

of all shortest transformation sequences does not exceed

10

5

.

Code Snippets

C++:

```
class Solution {
public:
vector<vector<string>> findLadders(string beginWord, string endWord,
vector<string>& wordList) {

}
};
```

Java:

```
class Solution {
public List<List<String>> findLadders(String beginWord, String endWord,
List<String> wordList) {

}
}
```

Python3:

```
class Solution:
def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) ->
List[List[str]]:
```

Python:

```
class Solution(object):
def findLadders(self, beginWord, endWord, wordList):
"""
:type beginWord: str
:type endWord: str
:type wordList: List[str]
:rtype: List[List[str]]
"""
```

JavaScript:

```
/**
* @param {string} beginWord
* @param {string} endWord
* @param {string[]} wordList
* @return {string[][]}
*/
var findLadders = function(beginWord, endWord, wordList) {

};
```

TypeScript:

```
function findLadders(beginWord: string, endWord: string, wordList: string[]):
string[][] {

};
```

C#:

```
public class Solution {
public IList<IList<string>> FindLadders(string beginWord, string endWord,
IList<string> wordList) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
char*** findLadders(char* beginWord, char* endWord, char** wordList, int
wordListSize, int* returnSize, int** returnColumnSizes) {

}
```

Go:

```
func findLadders(beginWord string, endWord string, wordList []string)
[][]string {
```

```
}
```

Kotlin:

```
class Solution {
fun findLadders(beginWord: String, endWord: String, wordList: List<String>):
List<List<String>> {

}
}
```

Swift:

```
class Solution {
func findLadders(_ beginWord: String, _ endWord: String, _ wordList:
[String]) -> [[String]] {

}
}
```

Rust:

```
impl Solution {
pub fn find_ladders(begin_word: String, end_word: String, word_list:
Vec<String>) -> Vec<Vec<String>> {

}
}
```

Ruby:

```
# @param {String} begin_word
# @param {String} end_word
# @param {String[]} word_list
# @return {String[][]}
def find_ladders(begin_word, end_word, word_list)

end
```

PHP:

```
class Solution {

/**
* @param String $beginWord
* @param String $endWord
* @param String[] $wordList
* @return String[][]
*/
function findLadders($beginWord, $endWord, $wordList) {

}
}
```

Dart:

```
class Solution {
List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {

}
}
```

Scala:

```
object Solution {
def findLadders(beginWord: String, endWord: String, wordList: List[String]):
List[List[String]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec find_ladders(begin_word :: String.t, end_word :: String.t, word_list ::
[String.t]) :: [[String.t]]
def find_ladders(begin_word, end_word, word_list) do

end
end
```

Erlang:

```
-spec find_ladders(BeginWord :: unicode:unicode_binary(), EndWord ::
unicode:unicode_binary(), WordList :: [unicode:unicode_binary()]) ->
[[unicode:unicode_binary()]].
find_ladders(BeginWord, EndWord, WordList) ->
.
```

Racket:

```
(define/contract (find-ladders beginWord endWord wordList)
(-> string? string? (listof string?) (listof (listof string?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Word Ladder II
* Difficulty: Hard
* Tags: string, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<vector<string>> findLadders(string beginWord, string endWord,
vector<string>& wordList) {

}
};
```

Java Solution:

```
/**
* Problem: Word Ladder II
* Difficulty: Hard
* Tags: string, hash, search
*
```

```
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public List<List<String>> findLadders(String beginWord, String endWord,
List<String> wordList) {

}
}
```

Python3 Solution:

```
"""
Problem: Word Ladder II
Difficulty: Hard
Tags: string, hash, search

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) ->
List[List[str]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def findLadders(self, beginWord, endWord, wordList):
"""
:type beginWord: str
:type endWord: str
:type wordList: List[str]
:rtype: List[List[str]]
"""
```

JavaScript Solution:

```
/**
* Problem: Word Ladder II
* Difficulty: Hard
* Tags: string, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {string} beginWord
* @param {string} endWord
* @param {string[]} wordList
* @return {string[][]}
*/
var findLadders = function(beginWord, endWord, wordList) {

};
```

TypeScript Solution:

```
/**
* Problem: Word Ladder II
* Difficulty: Hard
* Tags: string, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function findLadders(beginWord: string, endWord: string, wordList: string[]):
string[][] {

};
```

C# Solution:

```
/*
* Problem: Word Ladder II
```

```
* Difficulty: Hard
* Tags: string, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public IList<IList<string>> FindLadders(string beginWord, string endWord,
IList<string> wordList) {

}
}
```

C Solution:

```
/*
* Problem: Word Ladder II
* Difficulty: Hard
* Tags: string, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
char*** findLadders(char* beginWord, char* endWord, char** wordList, int
wordListSize, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Word Ladder II
// Difficulty: Hard
// Tags: string, hash, search
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func findLadders(beginWord string, endWord string, wordList []string)
[][]string {

}
```

Kotlin Solution:

```
class Solution {
fun findLadders(beginWord: String, endWord: String, wordList: List<String>):
List<List<String>> {

}
}
```

Swift Solution:

```
class Solution {
func findLadders(_ beginWord: String, _ endWord: String, _ wordList:
[String]) -> [[String]] {

}
}
```

Rust Solution:

```
// Problem: Word Ladder II
// Difficulty: Hard
// Tags: string, hash, search
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
```

```
pub fn find_ladders(begin_word: String, end_word: String, word_list:
Vec<String>) -> Vec<Vec<String>> {

}
}
```

Ruby Solution:

```
# @param {String} begin_word
# @param {String} end_word
# @param {String[]} word_list
# @return {String[][]}
def find_ladders(begin_word, end_word, word_list)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $beginWord
* @param String $endWord
* @param String[] $wordList
* @return String[][]
*/
function findLadders($beginWord, $endWord, $wordList) {

}
}
```

Dart Solution:

```
class Solution {
List<List<String>> findLadders(String beginWord, String endWord, List<String>
wordList) {

}
}
```

Scala Solution:

```
object Solution {
def findLadders(beginWord: String, endWord: String, wordList: List[String]):
List[List[String]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec find_ladders(begin_word :: String.t, end_word :: String.t, word_list ::
[String.t]) :: [[String.t]]
def find_ladders(begin_word, end_word, word_list) do

end
end
```

Erlang Solution:

```
-spec find_ladders(BeginWord :: unicode:unicode_binary(), EndWord ::
unicode:unicode_binary(), WordList :: [unicode:unicode_binary()]) ->
[[unicode:unicode_binary()]].
find_ladders(BeginWord, EndWord, WordList) ->
.
```

Racket Solution:

```
(define/contract (find-ladders beginWord endWord wordList)
(-> string? string? (listof string?) (listof (listof string?)))
)
```