<https://leetcode.com/problems/longest-string-chain/description/>

You are given an array of words where each word consists of lowercase English letters.

wordA is a **predecessor** of wordB if and only if we can insert **exactly one** letter anywhere in wordA **without changing the order of the other characters** to make it equal to wordB.

For example, "abc" is a **predecessor** of "abac", while "cba" is not a **predecessor** of "bcad".

A **word chain** is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a **predecessor** of word2, word2 is a **predecessor** of word3, and so on. A single word is trivially a **word chain** with k == 1.

Return the **length** of the **longest possible word chain** with words chosen from the given list of words.

**Example 1:**

**Input:** words = ["a","b","ba","bca","bda","bdca"]

**Output:** 4

**Explanation**: One of the longest word chains is ["a","ba","bda","bdca"].

**Example 2:**

**Input:** words = ["xbc","pcxbcf","xb","cxbc","pcxbc"]

**Output:** 5

**Explanation:** All the words can be put in a word chain ["xb", "xbc", "cxbc", "pcxbc", "pcxbcf"].

**Example 3:**

**Input:** words = ["abcd","dbqca"]

**Output:** 1

**Explanation:** The trivial word chain ["abcd"] is one of the longest word chains.["abcd","dbqca"] is not a valid word chain because the ordering of the letters is changed.

**Constraints:**

1 <= words.length <= 1000

1 <= words[i].length <= 16

words[i] only consists of lowercase English letters.

**Attempt 1: 2024-07-20**

**Solution 1: DFS + Memoization (180 min)**

**Style 1: Increasing Word Length**

class Solution {

    Map<Integer, List<String>> wordLenMap;

    Map<String, Integer> memo;

    public int longestStrChain(String[] words) {

        wordLenMap = new HashMap<>();

        memo = new HashMap<>();

        for(String word : words) {

            wordLenMap.putIfAbsent(word.length(), new ArrayList<>());

            wordLenMap.get(word.length()).add(word);

        }

        int maxLen = 1;

        for(String word : words) {

            maxLen = Math.max(maxLen, helper(word));

        }

        return maxLen;

    }

    private int helper(String word) {

        // If there are no words of the next length, we're done with this path.

        if(!wordLenMap.containsKey(word.length() + 1)) {

            return 1;

        }

        if(memo.containsKey(word)) {

            return memo.get(word);

        }

        int maxPath = 0;

        List<String> nextWords = wordLenMap.get(word.length() + 1);

        // For each word, find all words which are 1 letter longer and see if

        // they are valid successors.

        for(String nextWord : nextWords) {

            if(isOneCharLess(word, nextWord)) {

                maxPath = Math.max(maxPath, helper(nextWord));

            }

        }

        memo.put(word, maxPath + 1);

        return maxPath + 1;

    }

    private boolean isOneCharLess(String word, String nextWord) {

        int i = 0;

        int j = 0;

        while(i < word.length() && j < nextWord.length()) {

            if(word.charAt(i) == nextWord.charAt(j)) {

                i++;

            }

            j++;

        }

        return i == word.length();

    }

}

Time complexity: O(n\*m^2): In dfs(), we look at n word at most one time due to memoization.

At each dfs(), we compare all nextWords which takes at most O(m^2)

Space complexity: O(n) due to the call stack

**Style 2: Decreasing Word Length**

class Solution {

    Set<String> set;

    Map<String, Integer> memo;

    public int longestStrChain(String[] words) {

        set = new HashSet<>();

        memo = new HashMap<>();

        Collections.addAll(set, words);

        int maxPath = 1;

        for(String word : words) {

            maxPath = Math.max(maxPath, helper(word));

        }

        return maxPath;

    }

    private int helper(String word) {

        if(memo.containsKey(word)) {

            return memo.get(word);

        }

        StringBuilder sb = new StringBuilder(word);

        int maxPath = 0;

        // Delete each character, check if that's a valid word in the set,

        // add the character back and continue

        for(int i = 0; i < word.length(); i++) {

            sb.deleteCharAt(i);

            String prevWord = sb.toString();

            if(set.contains(prevWord)) {

                maxPath = Math.max(maxPath, helper(prevWord));

            }

            sb.insert(i, word.charAt(i));

        }

        memo.put(word, maxPath + 1);

        return maxPath + 1;

    }

}

Time complexity: O(n\*m^2): In dfs(), we look at 'n' words at most one time due to memoization.

At each dfs(), we loop through m characters and use remove() and insert() which each cost O(m).

Space complexity: O(n) due to the call stack.

**Solution 2: DP (30 min)**

**Style 1: Increasing Word Length**

class Solution {

    public int longestStrChain(String[] words) {

        Map<Integer, List<String>> wordLenMap = new HashMap<>();

        for(String word : words) {

            wordLenMap.putIfAbsent(word.length(), new ArrayList<>());

            wordLenMap.get(word.length()).add(word);

        }

        Map<String, Integer> dp = new HashMap<>();

        // Different than Solution 2's (Decreasing word length) increasing length way,

        // here we sort word as decreasing length

        Arrays.sort(words, (String a, String b) -> b.length() - a.length());

        int maxPath = 1;

        // Same idea behind the previous approach but performed iteratively.

        for (String word: words) {

            int currLength = 1;

            List<String> nextWords = wordLenMap.get(word.length() + 1);

            // Before iterating over nextWords, a null check is performed to ensure nextWords is not null.

            // This prevents the NullPointerException, since we sort words as decreasing length, and

            // first query on length will be existing maximum length + 1, which surely not able to find

            // no word at this length, so nextWords will be null at first iteration, we have to avoid

            if(nextWords != null) {

                for (String nextWord : nextWords) {

                    if(isOneCharLess(word, nextWord)) {

                        currLength = Math.max(currLength, dp.getOrDefault(nextWord, 0) + 1);

                    }

                }

            }

            dp.put(word, currLength);

            maxPath = Math.max(maxPath, currLength);

        }

        return maxPath;

    }

    private boolean isOneCharLess(String word, String nextWord) {

        int i = 0;

        int j = 0;

        while(i < word.length() && j < nextWord.length()) {

            if(word.charAt(i) == nextWord.charAt(j)) {

                i++;

            }

            j++;

        }

        return i == word.length();

    }

}

**Style 2: Decreasing Word Length**

class Solution {

    // Start with multi-header DFS since for loop each word with

    // DFS method as helper(word)

    // On each helper(word) start with full length word till 1 length word

    // with recursively delete 1 char

    // -> top at full length of a word, bottom at 1 length of a word

    // DP -> bottom to top -> for each word, iterate as 1 length word (more precisely is

    // minimum length word) to full length word

    // This action is a bit different than standard convertion, its actually implement by

    // two statements:

    // (1) Arrays.sort(words, (String a, String b) -> a.length() - b.length()); // sort by length

    // (2) for (int i=0; i<word.length(); i++) { sb.deleteCharAt(i); ...dp.getOrDefault(prevWord, 0) + 1...}

    // The first statement will help to put smallest length word at beginning

    // of DP loop, the second statement together with first statement helps to

    // guarantee calculate from 1 length word(minimum length word) to full length

    // word, since after sort we start with smallest length word, and if still

    // delete char from this smallest length word to get its predecessor is surely

    // means no predecessor found, and 1 length word(minimum length word) will auto

    // set as count 1 by 'dp.getOrDefault(prevWord, 0) + 1', and to find valid

    // predecessor at least will move on to the next iteration which relate to

    // second minimum length word

    public int longestStrChain(String[] words) {

        Map<String, Integer> dp = new HashMap<>();

        Arrays.sort(words, (String a, String b) -> a.length() - b.length()); // sort by length

        int maxPath = 1;

        // Same idea behind the previous approach but performed iteratively.

        for (String word: words) {

            int currLength = 1;

            StringBuilder sb = new StringBuilder(word);

            for (int i=0; i<word.length(); i++) {

                sb.deleteCharAt(i);

                String prevWord = sb.toString();

                currLength = Math.max(currLength, dp.getOrDefault(prevWord, 0) + 1);

                sb.insert(i, word.charAt(i));

            }

            dp.put(word, currLength);

            maxPath = Math.max(maxPath, currLength);

        }

        return maxPath;

    }

}

**How transfer below DFS style solution into Dynamic Programming style ? And explain how it convert to ?**

**Refer to chatGPT**

**For Style 1**

Let's break down how the DFS solution is converted into a dynamic programming (DP) solution for the Leetcode 1048 problem.

**DFS Solution:**

The DFS solution uses a recursive approach with memoization to explore all possible chains of words, where each word is one character longer than the previous one.

Data Structures:

wordLenMap: A map that groups words by their lengths.

memo: A memoization map to store the longest chain length for each word.

Recursive Function **helper(word)**:

Base Case: If there are no words of length word.length() + 1, return 1.

Memoization Check: If the result for the current word is already computed, return it from the memo.

Recursively check all possible next words (one character longer) and update the maximum path length.

Store the result in the memo and return it.

Main Function:

Initialize the wordLenMap and memo.

For each word, call the helper function and update the maximum chain length.

**Dynamic Programming Solution:**

The DP solution iteratively builds the longest chain lengths for words in decreasing order of length.

Data Structures:

wordLenMap: Same as in the DFS solution, grouping words by their lengths.

dp: A map to store the longest chain length for each word.

Sorting:

Sort the words in descending order of length. This ensures that when processing each word, all potential successor words (one character longer) have already been considered.

Iterative Process:

For each word, initialize currLength to 1.

Retrieve the list of potential next words (one character longer).

For each potential next word, check if it can be a valid successor using isOneCharLess.

Update currLength based on the maximum chain length of valid successor words.

Store the result in the dp map.

Update the overall maximum chain length.

**Conversion Explanation:**

Memoization to Iteration:

Instead of recursively calling the helper function, iterate over the words starting from the longest.

Use the dp map to store intermediate results, similar to how memoization was used.

Sorting:

Sorting the words in descending order allows us to process each word after its potential successors have been processed.

Dynamic Programming Table Update:

For each word, update the dp map iteratively instead of recursively.

**For Style 2**

To convert the given DFS solution for Leetcode 1048 into a dynamic programming solution, we'll avoid recursion and instead use a hashmap to store the longest chain length ending at each word. Here's the process for converting the DFS approach to DP:

**Identify the base cases: In the DFS approach, the base case is when no more valid predecessor words can be formed from the current word. This is naturally handled in the DP approach by initializing the chain length for each word as 1.**

**Use a HashMap to store intermediate results: Instead of recursively calling helper to get the maximum path, we store the result in a hashmap and use it to avoid redundant calculations.**

**Process words by increasing length: By sorting the words by length, we ensure that when processing each word, all potential predecessors have already been processed.**

Here is the dynamic programming implementation:

import java.util.\*;

class Solution {

public int longestStrChain(String[] words) {

// Sort words by length

Arrays.sort(words, Comparator.comparingInt(String::length));

// Map to store the longest chain length ending at each word

Map<String, Integer> dp = new HashMap<>();

int maxLength = 1;

for (String word : words) {

int currentLength = 1; // At least the word itself

// Try removing each character to form a predecessor word

for (int i = 0; i < word.length(); i++) {

StringBuilder sb = new StringBuilder(word);

sb.deleteCharAt(i);

String prevWord = sb.toString();

if (dp.containsKey(prevWord)) {

currentLength = Math.max(currentLength, dp.get(prevWord) + 1);

}

}

dp.put(word, currentLength);

maxLength = Math.max(maxLength, currentLength);

}

return maxLength;

}

}

**Explanation**

Sorting Words by Length:

Sorting the words by length ensures that we process the words in order of increasing length. This ensures that all potential predecessor words have already been processed when we encounter a word.

Using a HashMap for DP:

dp is a hashmap that stores the longest chain length ending at each word. The key is the word, and the value is the longest chain length ending at that word.

Processing Each Word:

Initialize currentLength to 1 (the word itself).

For each character in the word, remove it to form a predecessor word.

Check if the predecessor word exists in dp. If it does, update currentLength to be the maximum of currentLength and dp.get(prevWord) + 1.

This ensures that we are considering all possible predecessor chains for the current word.

Updating Results:

Store the currentLength for the current word in dp.

Update maxLength to the maximum value of currentLength.

**Complexity Analysis**

Time Complexity: O(n \* m^2), where n is the number of words and m is the maximum length of a word. Sorting takes O(n log n), and for each word, removing each character and checking the predecessor in the map takes O(m^2).

Space Complexity: O(n \* m), where n is the number of words and m is the maximum length of a word. The hashmap stores each word with its chain length, and the maximum number of characters to store is n \* m.

This dynamic programming solution ensures that we compute the longest string chain efficiently by leveraging a hashmap to store intermediate results and avoid redundant calculations.

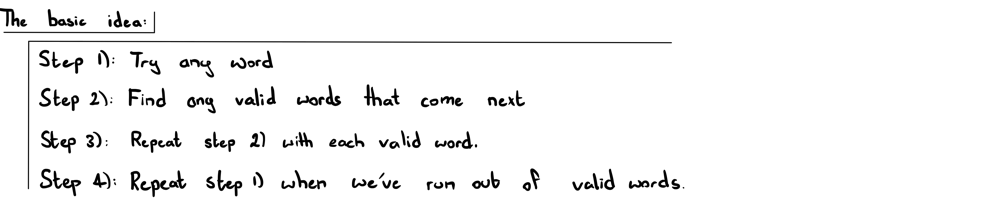
**Refer to**

<https://leetcode.com/problems/longest-string-chain/solutions/2153004/explaining-three-approaches-java/>

**Intuition:**

There are quite a few ways to solve this problem. Below are some ways I've compiled together and I'll try my best to explain how they work and how you might transition from an intuitive thought process to code. Let's get started!

The fundamental logic behind all three solutions remains the same:



Below are the three main approaches I'll go through that follow the above template:

DFS + Memoization: Increasing the length of the word in each call

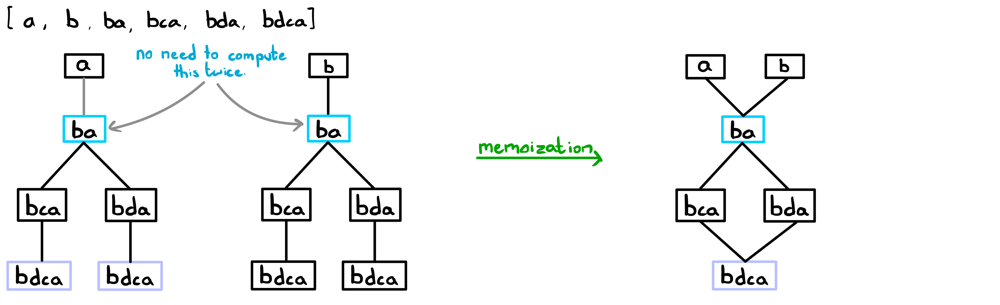
DFS + Memoization: Decreasing the length of the word in each call

DP: Decreasing the length of the word in each iteration

**DFS + Memoization:**

When you're first thinking of a memoization solution to this question, you've probably thought of starting from any word and then finding all the valid "next words" that follow after. This makes the most sense intuitively since this is quite literally what the question is asking.

**Approach 1: Increase Word Length:**



As you can see from the above illustration, we're computing the results for some strings multiple times. If we already know how deep a path can go from any given string, why calculate it again? We don't have to; we can just store the result of that string in a memo cache and never have to recalculate for that string ever again. This concept is known asmemoization.

How do we implement this in code?

Turns out that all the valid words that follow the current word are guaranteed to be exactly one letter longer. Therefore, we can just store all the words with their associated lengths in a hashmap. E.g. Words with length 1: "a" and "b" would be stored with key 1. Therefore, we can just get the list of words with a certain length, loop through them and see if any of them are valid words to continue the chain.

To compare two words, we'll use a custom function as you'll see in the code.

**DFS + Memoization | Increasing Word Length:**

Time complexity: O(n\*m^2): In dfs(), we look at n word at most one time due to memoization. At each dfs(), we compare all nextWords which takes at most O(m^2)

Space complexity: O(n) due to the call stack

class Solution {

    private Map<Integer, List<String>> wordLengthMap;

    private Map<String, Integer> memo;

    public int longestStrChain(String[] words) {

        // store each word with its corresponding length

        wordLengthMap = new HashMap<>();

        for (String word: words) {

            wordLengthMap.putIfAbsent(word.length(), new ArrayList<>());

            wordLengthMap.get(word.length()).add(word);

        }

        int maxPath = 1;

        memo = new HashMap<>();

        for (String word: words)

            maxPath = Math.max(maxPath, dfs(word));

        return maxPath;

    }

    private int dfs(String word) {

        if (!wordLengthMap.containsKey(word.length() + 1)) return 1; // if there are no words of the next length, we're done with this path.

        if (memo.containsKey(word)) return memo.get(word); // if we're computed this word before, return the result.

        int maxPath = 0;

        // for each word, find all words which are 1 letter longer and see if they are valid successors.

        List<String> nextWords = wordLengthMap.get(word.length() + 1);

        for (String nextWord: nextWords)

            if (isOneOff(word, nextWord))

                maxPath = Math.max(maxPath, dfs(nextWord));

        memo.put(word, maxPath + 1); // store our result

        return memo.get(word);

    }

    // returns true if two strings differ by no more than 1 letter

    private boolean isOneOff(String a, String b) {

        int count = 0;

            if (a.charAt(j) != b.charAt(i)) count++;

            else j++;

        }

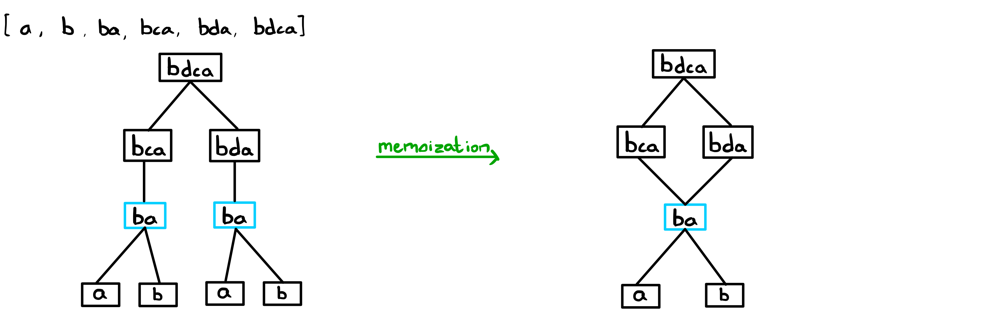
        return count <= 1;

    }

}

**Approach 2: Decrease Word Length:**

The slightly less intuitive idea here is to go in reverse; removing a letter from each word as the "next" step. This is a good example of the "reframing the question" strategy I've discussed in previous posts! This approach has the same time complexity as the previous approach but it's a good example of how you can approach a question from multiple angles. Let's see how this would work:



Again, memoization is applied to avoid recomputation.

How do we implement this in code?

The way we find the next (or in this case, the previous) words is a bit different to how we did it in the previous approach despite following the same general structure. Since we're decreasing a letter each time, we can simply chop off a letter from the current word, see if it exists in words[], and add the letter back afterwards before continuing to the next letter.

For constant time look-up, we'll convert our words[] array to a hash set.

**DFS + Memoization | Decreasing Word Length:**

Time complexity: O(n\*m^2): In dfs(), we look at 'n' words at most one time due to memoization. At each dfs(), we loop through m characters and use remove() and insert() which each cost O(m).

Space complexity: O(n) due to the call stack.

class Solution {

    private Set<String> wordDict;

    private Map<String, Integer> memo;

    public int longestStrChain(String[] words) {

        wordDict = new HashSet<>();

        Collections.addAll(wordDict, words); // adding all words to a set for constant look-up

        int maxPath = 1;

        memo = new HashMap<>();

        for (String word: words)

            maxPath = Math.max(maxPath, dfs(word));

        return maxPath;

    }

    private int dfs(String word) {

        if (memo.containsKey(word)) return memo.get(word); // if we're computed this word before, return the result.

        StringBuilder sb = new StringBuilder(word);

        int maxPath = 0;

        // delete each character, check if that's a valid word in the set, add the character back and continue

        for (int i=0; i<word.length(); i++) {

            sb.deleteCharAt(i);

            String prevWord = sb.toString();

            if (wordDict.contains(prevWord))

                maxPath = Math.max(maxPath, dfs(prevWord));

            sb.insert(i, word.charAt(i));

        }

        memo.put(word, maxPath + 1); // store the result

        return memo.get(word);

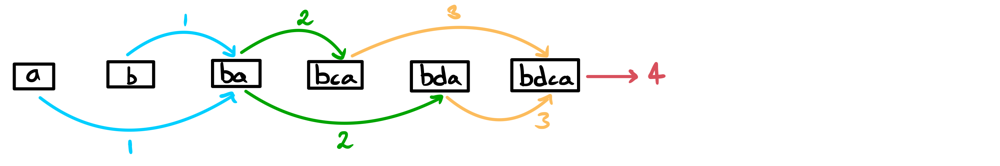
    }

}

**Dynamic Programming:**

**This is just your usual memoization to DP conversion. Nothing special! It's basically an iterative version of the corresponding memoization algorithm. We can choose either of the above methods to convert to but I've chosen the second approach.** As an exercise for yourself, try converting approach 1 from memoization to DP :)

**Approach 3: Decrease Word Length using DP:**



How do we implement this in code?

The most important adjustment here is that we need to sort our input array by length since at each word, we add to it the max length from each previous string in the chain. So, we would like all shorter strings to be computed before longer ones. **If you chose to convert approach 1 to DP, you'd have to sort the array in reverse order. Remember, the fundamental approach hasn't changed, but our execution has altered slightly since we're replacing our memo table with a DP table,**

That's it! Now we're ready to code something up.

**DP | Decreasing Word Length:**

Time complexity: O(nlogn + n\*m^2): sorting + same reasons as above.

Space complexity: O(n) for the dp map.

class Solution {

    public int longestStrChain(String[] words) {

        Arrays.sort(words, (String a, String b) -> a.length() - b.length()); // sort by length

        Map<String, Integer> dp = new HashMap<>();

        int maxPath = 1;

        // same idea behind the previous approach but performed iteratively.

        for (String word: words) {

            int currLength = 1;

            StringBuilder sb = new StringBuilder(word);

            for (int i=0; i<word.length(); i++) {

                sb.deleteCharAt(i);

                String prevWord = sb.toString();

                currLength = Math.max(currLength, dp.getOrDefault(prevWord, 0) + 1);

                sb.insert(i, word.charAt(i));

            }

            dp.put(word, currLength);

            maxPath = Math.max(maxPath, currLength);

        }

        return maxPath;

    }

}