1048. Longest String Chain

Leetcode Link

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

$\texttt{word}_A$ {" "} is a **predecessor** of{" "} $\texttt{word}_B$ {" "} if and only if we can insert **exactly one** letter anywhere in{" "} $\texttt{word}_A$ {" "} **without changing the order of the other characters** to make it equal to{" "} $\texttt{word}_B$.

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

A **word chain** is a sequence of words{" "} $\texttt{[word}_1\texttt{, word}_2\texttt{, ..., word}_k\texttt{]}$ {" "} with `k >= 1`, where{" "} $\texttt{word}_1$ {" "} is a **predecessor** of{" "} $\texttt{word}_2$,{" "} $\texttt{word}_2$ {" "} is a **predecessor** of{" "} $\texttt{word}_3$, 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"]{"\n"}
Output: 4{"\n"}
Explanation: One of the longest word chains is ["a","
ba","bda","bdca"].{"\n"}
```

Example 2:

```
Input: words = ["xbc","pcxbcf","xb","cxbc","pcxbc"]{"\n"}
Output: 5{"\n"}
Explanation: All the words can be put in a word chain
["xb", "xbc", "cxbc", "pcxbc", "pcxbcf"].{"\n"}
```

Example 3:

```
Input: words = ["abcd","dbqca"]{"\n"}
Output: 1{"\n"}
Explanation: The trivial word chain ["abcd"] is one of the
longest word chains.{"\n"}["abcd","dbqca"] is not a valid word chain because
the ordering of the letters is changed.{"\n"}
```

Constraints:

- `1 <= words.length <= 1000`
- `1 <= words[i].length <= 16`
- `words[i]` only consists of lowercase English letters.

Solution

Naive Solutions

We could go through all possible chains and check if each works, or we could use a recursive function that adds one character every time, but these solutions are too slow.

$\mathcal{O}(n^2L^2)$ Dynamic Programming Solution

Let's build a chain from the shortest string to the longest string. Say we currently have the chain $(\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_{k-1}, \mathrm{w}_k)$. We see that it doesn't matter what any of the words before w_k are, only that there are k of them. This suggests a dynamic programming solution, where $\mathrm{dp}[i] =$ the length of the longest chain that ends with the string $\mathrm{words}[i]$.

First sort words by increasing length. An outer loop will loop $\mathrm{current\_word\_index}$. For each $\mathrm{current\_word\_index}$, we have an inner loop to check every word before it to see if it can be a predecessor. We'll call this index $\mathrm{previous\_word\_index}$. If so, we update $\mathrm{dp}[\mathrm{current\_word\_index}]$ to $\max(\mathrm{dp}[\mathrm{current\_word\_index}], \mathrm{dp}[\mathrm{previous\_word\_index}] + 1)$ because we can extend the chain ending in $\mathrm{previous\_word}$ by appending $\mathrm{current\_word}$.

For example, when $\mathrm{words} = [\texttt{"a"}, \texttt{"ab"}, \texttt{"cb"}, \texttt{"cab"}]$, $\mathrm{current\_word\_index} = 3$, and $\mathrm{previous\_word\_index} = 1$, we find that removing $\texttt{"c"}$ from $\mathrm{words}[\mathrm{current\_word\_index}] = \texttt{"cab"}$ yields $\texttt{"ab"} = \mathrm{words}[\mathrm{previous\_word\_index}]$.

Time complexity

Let $n = \mathrm{words.length}$ and $L = \max\{\mathrm{words[i].length}\}$.

Sorting takes $\mathcal{O}(n \log n)$.

Our nested loops take $\mathcal{O}(n^2)$ and call `isPredecessor()` that runs in $\mathcal{O}(L^2)$, so this part takes $\mathcal{O}(n^2L^2)$.

The total time complexity of this algorithm is therefore $\mathcal{O}(n \log n + n^2L^2) = \mathcal{O}(n^2L^2)$.

Space complexity

The dp array consumes $\mathcal{O}(n)$ memory.

$\mathcal{O}(n^2L^2)$ Solutions below

C++ Solution

```cpp
class Solution {
  public:
    bool isPredecessor(string &previous_word, string &current_word) {
            if (previous_word.size() + 1 == current_word.size()) {
                    for (int k = 0; k < current_word.size(); k++) {
                            if (previous_word == current_word.substr(0, k) + current_word.substr(k + 1))
                                    return true;
                    }
            }
            return false;
    }
    int longestStrChain(vector<string> &words) {
            // sort words by length
            sort(words.begin(), words.end(), [](auto x, auto y) { return x.size() < y.size(); });
            int n = words.size();
            vector<int> dp(n, 1);
            for (int current_word_index = 0; current_word_index < n; current_word_index++) {
                    for (int previous_word_index = 0; previous_word_index < current_word_index; previous_word_index++) {
                            if (isPredecessor(words[j], words[current_word_index])) {
                                    dp[current_word_index] = max(dp[current_word_index], dp[previous_word_index] + 1);
                            }
                    }
            }
            return *max_element(dp.begin(), dp.end());
    }
};
```

Java Solution

```java
class Solution {
    public static boolean isPredecessor(String previous_word, String current_word) {
        if (previous_word.length() + 1 == current_word.length()) {
            for (int k = 0; k < current_word.length(); k++) {
                if (previous_word.equals(current_word.substring(0, k) + current_word.substring(k+1)))
                    return true;
            }
        }
        return false;
    }
    public int longestStrChain(String[] words) {
        // sort words by length
        Arrays.sort(words, (a, b) -> Integer.compare(a.length(), b.length()));
        int n = words.length;

        // add padding to words to avoid index out of bounds errors
        for (int i = 0; i < n; i++) {
            words[i] = "#" + words[i];
        }

        int ans = 1;
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        for (int current_word_index = 0; current_word_index < n; current_word_index++) {
            for (int previous_word_index = 0; previous_word_index < current_word_index; previous_word_index++) {
                if (isPredecessor(words[previous_word_index], words[current_word_index])) {
                    dp[current_word_index] = Math.max(dp[current_word_index], dp[previous_word_index] + 1);
                    ans = Math.max(ans, dp[current_word_index]);
                }

            }
        }
        return ans;
    }
}
```

Python Solution

```python
class Solution:
    def isPredecessor(self, previous_word, current_word):
        if len(previous_word) + 1 == len(current_word):
            for k in range(len(current_word)):
                if previous_word == current_word[0:k] + current_word[k+1:]:
                    return True
        return False

    def longestStrChain(self, words: List[str]) -> int:
        words.sort(key=len)
        n = len(words)
        dp = [1 for _ in range(n)]
        for current_word_index in range(n):
            for previous_word_index in range(current_word_index):
                if self.isPredecessor(words[previous_word_index], words[current_word_index]):
                    dp[current_word_index] = max(dp[current_word_index], dp[previous_word_index] + 1)
        return max(dp)
```

JavaScript Solution

```javascript
var isPredecessor = function (previous_word, current_word) {
  if (previous_word.length + 1 == current_word.length) {
    for (let k = 0; k < current_word.length; k++) {
      if (
        previous_word ===
        current_word.slice(0, k) + current_word.slice(k + 1)
      )
        return true;
    }
  }
  return false;
};

/**
 * @param {string[]} words
 * @return {number}
 */
var longestStrChain = function (words) {
  // sort words by length
  words.sort((a, b) => a.length - b.length);
  n = words.length;
  dp = new Array(n).fill(1);

  for (
    let current_word_index = 0;
    current_word_index < n;
    current_word_index++
  ) {
    for (
      let previous_word_index = 0;
      previous_word_index < current_word_index;
      previous_word_index++
    ) {
      if (
        isPredecessor(words[previous_word_index], words[current_word_index])
      ) {
        dp[current_word_index] = Math.max(
          dp[current_word_index],
          dp[previous_word_index] + 1
        );
      }
    }
  }
  return Math.max(...dp);
};
```

$\mathcal{O}(nL^2)$ Dynamic Programming Solution (requires HashMap)

In our last solution, checking all previous words for predecessors was slow.

Observe that each word can only have $\mathcal{O}(L)$ predecessors, so we can generate them all (in $\mathcal{O}(L^2)$) and check the best dp value we can get from them.

Let $\mathrm{dp}[\mathrm{str}] =$ the length of the longest chain that ends with the string str. dp will be a hashmap to allow for $\mathcal{O}(1)$ retrieval by key.

As we did before, sort words by increasing length. Loop through the current string $\mathrm{current\_word}$. Then for each index j, let $\mathrm{previous\_word}_j$ be $\mathrm{current\_word}$ with its jth character removed. Set $\mathrm{dp}[\mathrm{current\_word}]$ to $\max\{\mathrm{dp}[\mathrm{previous\_word}_j] + 1\}$. Our answer is the max of all entries in dp.

Time complexity

Let $n = \mathrm{words.length}$ and $L = \max\{\mathrm{words[i].length}\}$.

Sorting takes $\mathcal{O}(n \log n)$.

For every string (there are $\mathcal{O}(n)$ of them), create each of its predecessors in $\mathcal{O}(L^2)$ and check for their value in dp in $\mathcal{O}(1)$. This comes together to $\mathcal{O}(nL^2)$.

The total time complexity is therefore $\mathcal{O}(n \log n + nL^2)$.

Space complexity

The dp hashmap consumes $\mathcal{O}(n)$ memory.

Bonus:

We can perform counting sort in $\mathcal{O}(n + L)$, giving a total time complexity of $\mathcal{O}((n + L) + nL^2) = \mathcal{O}(nL^2)$.

$\mathcal{O}(n \log n + nL^2)$ Solutions below

C++ Solution

```cpp
class Solution {
public:
    int longestStrChain(vector<string> &words) {
            // sort words by length
            sort(words.begin(), words.end(), [](auto x, auto y) { return x.size() < y.size(); });
            int n = words.size(), ans = 1;
        unordered_map<string, int> dp;
        for (auto &current_word: words) {
            for (int j = 0; j < current_word.size(); j++) {
                string previous_word = current_word.substr(0, j) + current_word.substr(j + 1);
                dp[current_word] = max(dp[current_word], dp[previous_word] + 1);
            }
            ans = max(ans, dp[current_word]);
        }
            return ans;
    }
};
```

Java Solution

```java
class Solution {
    public int longestStrChain(String[] words) {
        // sort words by length
        Arrays.sort(words, (a, b) -> Integer.compare(a.length(), b.length()));
        int n = words.length;
        // add padding to words so that we don't have to do s.substring(0, -1).
        // which would throw an index out of bounds error
        for (int i = 0; i < n; i++) {
            words[i] = "#" + words[i];
        }
        TreeMap<String, Integer> dp = new TreeMap<>();
        int ans = 1;
        for (String current_word: words) {
            dp.put(current_word, 1);
            for (int j = 0; j < current_word.length(); j++) {
                String previous_word = current_word.substring(0, j) + current_word.substring(j + 1);
                dp.put(current_word, Math.max(dp.get(current_word), dp.getOrDefault(previous_word, 0) + 1));
            }
            ans = Math.max(ans, dp.get(current_word));
        }
        return ans;
    }
}
```

Python Solution

```python
class Solution:
    def longestStrChain(self, words: List[str]) -> int:
        # sort words by length
        words.sort(key=len)
        n = len(words)
        ans = 0
        dp = defaultdict(lambda: 0)
        for current_word in words:
            for j in range(len(current_word)):
                previous_word = current_word[0:j] + current_word[j+1:]
                dp[current_word] = max(dp[current_word], dp[previous_word] + 1)
            ans = max(ans, dp[current_word])
        return ans
```

JavaScript Solution

```javascript
/**
 * @param {string[]} words
 * @return {number}
 */
var longestStrChain = function (words) {
  words.sort((a, b) => a.length - b.length);
  n = words.length;
  dp = new Map();
  ans = 0;
  for (current_word of words) {
    dp[current_word] = 1;
    for (let j = 0; j < s.length; j++) {
      previous_word = current_word.slice(0, j) + current_word.slice(j + 1);
      // We need (dp[previous_word] || 0) to get 0 if dp does not contain previous_word, otherwise we'd get Nan,
      // which is larger than any integer
      dp[current_word] = Math.max(
        dp[current_word],
        (dp[previous_word] || 0) + 1
      );
    }
    ans = Math.max(ans, dp[current_word]);
  }
  return ans;
};
```

Got a question? Ask the Teaching Assistant anything you don't understand.