【哈希集合-英文输入法

题目描述与示例

题目

主管期望你来实现英文输入法单词联想功能，需求如下：

1. 依据用户输入的单词前缀，从已输入的英文语句中联想出用户想输入的单词。
2. 按字典序输出联想到的单词序列，如果联想不到，请输出用户输入的单词前缀。

注意：

1. 英文单词联想时区分大小写
2. 缩略形式如 `"don't"` 判定为两个单词 `"don"` 和 `"t"`
3. 输出的单词序列不能有重复单词，且只能是英文单词，不能有标点符号

输入

输入两行。

首行输入一段由英文单词 `word` 和 `标点` 构成的语句 `str` ，接下来一行为一个英文单词前缀 `pre` 。

`0 < word.length() <= 20` ， `0 < str.length() <= 10000` ， `0 < pre.length() <= 20`

输出

输出符合要求的单词序列或单词前缀。存在多个时，单词之间以单个空格分割

示例一

输入

```
I love you
He
```

输出

```
He
```

说明

用户已输入单词语句 `"I love you"` ，可以提炼出 `"I"` , `"love"` , `"you"` 三个单词。接下来用户输入 `"He"` ，
从已经输入信息中无法联想到符合要求的单词，所以输出用户输入的单词前缀。

示例二

输入

```
The furthest distance in the world,Is not between life and death,But when I stand in front or you,Yet you don't know that I love you.
f
```

输出

```
front furthest
```

解题思路

首先我们需要处理输入，将输入的字符串 `s` 根据标点符号和空格隔开，得到一个由若干单词 `word` 组成的单词列表 `lst` 。这里稍微有点麻烦，不能再用我们熟悉的 `split()` 方法完成，而是改为较为麻烦的遍历写法。

首先我们初始化 `lst = [""]` ，即单词列表中存放了一个空字符串。然后我们遍历字符串 `s` 中的字符 `ch` ，当

- `ch` 是字母，则将其加入到 `lst` 最后一个元素的末尾，即延长当前单词。如果此时 `lst[-1]` 为一个空字符串 `""` ，则 `ch` 充当了某个单词首位字母的角色。
- `ch` 不是字母，说明遇到一个标点符号，当前单词的获取已经结束， `lst` 的末尾插入一个新的空字符串 `""` 。

上述思路整理为代码后即为：

```
lst = [""]
```

```
    2
    3 for ch in s:
    4     if ch.isalpha():
    5         lst[-1] += ch
    6     else:
    7         lst.append("")
```

当然这个过程也可用**正则表达式**以更加简短的代码来完成，但这部分知识已经超纲，大部分题目完全用不上，学有余力的同学可以自行研究一下。

得到 `lst` 之后，剩下的工作就相当简单了。由于 `lst` 中可能出现重复单词，我们**使用哈希集合进行去重操作**。又因为最后的输出要求按照字典序排序，因此去重之后再对哈希集合进行调用 `sorted()` 内置函数，再转化为列表。

```
1 lst_sorted = list(sorted(set(lst)))
```

对于 `lst_sorted` 中的每一个单词 `word` ，我们可以使用切片来获得其前 `pre_length` 个字符所构成的字符串，并与 `pre` 进行比较，就能够得知 `word` 是否包含前缀 `pre` 了。

```
1 pre_length = len(pre)
2 for word in lst_sorted:
3     if word[:pre_length] == pre:
4         ans.append(word)
```

总体来说本题难度不大，甚至很难归类为哪一种具体的算法。

难点其实主要在于对输入的字符串处理，初始化 `lst = [""]` 实际上是一个颇有技巧的做法。

当然本题还存在着前缀树的最优解法，但也严重超纲，不要求掌握。

代码

解法一

Python

```python
# 题目：2024E-英文输入法
# 分值：100
# 作者：许老师-闭着眼睛学数理化
# 算法：哈希集合
# 代码看不懂的地方，请直接在群上提问

s = input()
pre = input()

# 初始化列表lst用于存放所有单词
lst = [""]

# 遍历s中的所有字符ch，如果
# 1. ch是字母，则加入到lst最后一个元素的末尾，即延长当前单词
# 2. ch不是字母，说明遇到一个标点符号，结束当前单词的获取，lst的末尾插入一个新的空字符串""
# 这个过程也可以使用正则表达式来完成，不要求掌握，学有余力的同学可以自学一下
for ch in s:
    if ch.isalpha():
        lst[-1] += ch
    else:
        lst.append("")

# 用哈希集合去重lst中可能出现的重复单词
# 去重后进行排序，排序后在转化为列表lst_sorted
lst_sorted = list(sorted(set(lst)))

# 初始化答案数组
ans = list()

# 获得pre的长度，用于切片
pre_length = len(pre)
# 遍历lst_sorted中的每一个单词
for word in lst_sorted:
    # 如果word前pre_length个字符的切片等于pre
    # 说明word的前缀是pre，将其加入答案数组ans中
    if word[:pre_length] == pre:
        ans.append(word)

# 如果ans长度大于0，说明至少存在一个单词的前缀是pre，输出由所有单词组成的字符串
# 如果ans长度等于0，说明不存在任何一个单词的前缀是pre，返回pre
print(" ".join(ans) if len(ans) > 0 else pre)
```

Java

```java
import java.util.HashSet;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 从输入获取字符串s和前缀pre
        String s = scanner.nextLine();
        String pre = scanner.nextLine();

        // 初始化列表lst用于存放所有单词
        ArrayList<String> lst = new ArrayList<>();
        lst.add("");

        // 遍历s中的所有字符ch
        for (char ch : s.toCharArray()) {
            // 如果ch是字母，则加入到lst最后一个元素的末尾，即延长当前单词
            if (Character.isLetter(ch)) {
                int lastIndex = lst.size() - 1;
                lst.set(lastIndex, lst.get(lastIndex) + ch);
            } else {
                // 如果ch不是字母，说明遇到一个标点符号，结束当前单词的获取，lst的末尾插入一个新的空字符串""
                lst.add("");
            }
        }

        // 用哈希集合去重lst中可能出现的重复单词
        HashSet<String> set = new HashSet<>(lst);
        // 去重后进行排序，排序后在转化为列表lstSorted
        ArrayList<String> lstSorted = new ArrayList<>(set);
        Collections.sort(lstSorted);

        // 初始化答案数组
        ArrayList<String> ans = new ArrayList<>();
        // 获得pre的长度，用于切片
        int preLength = pre.length();

        // 遍历lstSorted中的每一个单词
        for (String word : lstSorted) {
            // 如果word前preLength个字符的切片等于pre
            // 说明word的前缀是pre，将其加入答案数组ans中
            if (word.length() >= preLength && word.substring(0, preLength).equals(pre)) {
```

```
                ans.add(word);
            }
        }

        // 如果ans长度大于0，说明至少存在一个单词的前缀是pre，输出由所有单词组成的字符串
        // 如果ans长度等于0，说明不存在任何一个单词的前缀是pre，返回pre
        if (ans.size() > 0) {
            System.out.println(String.join(" ", ans));
        } else {
            System.out.println(pre);
        }
    }
}

```

C++

```
#include <iostream>
#include <unordered_set>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    string s;
    getline(cin, s);

    string pre;
    getline(cin, pre);

    // 初始化列表lst用于存放所有单词
    vector<string> lst;
    lst.push_back("");

    // 遍历s中的所有字符ch
    for (char ch : s) {
        // 如果ch是字母，则加入到lst最后一个元素的末尾，即延长当前单词
        if (isalpha(ch)) {
            int lastIndex = lst.size() - 1;
            lst[lastIndex] += ch;
        } else {
            // 如果ch不是字母，说明遇到一个标点符号，结束当前单词的获取，lst的末尾插入一个新的空字符串""
            lst.push_back("");
        }
```

```
    }

    // 用哈希集合去重lst中可能出现的重复单词
    unordered_set<string> set(lst.begin(), lst.end());
    // 去重后进行排序，排序后在转化为列表lstSorted
    vector<string> lstSorted(set.begin(), set.end());
    sort(lstSorted.begin(), lstSorted.end());

    // 初始化答案数组
    vector<string> ans;
    // 获得pre的长度，用于切片
    int preLength = pre.length();

    // 遍历lstSorted中的每一个单词
    for (string word : lstSorted) {
        // 如果word前preLength个字符的切片等于pre
        // 说明word的前缀是pre，将其加入答案数组ans中
        if (word.length() >= preLength && word.substr(0, preLength) == pre) {
            ans.push_back(word);
        }
    }

    // 如果ans长度大于0，说明至少存在一个单词的前缀是pre，输出由所有单词组成的字符串
    // 如果ans长度等于0，说明不存在任何一个单词的前缀是pre，返回pre
    if (!ans.empty()) {
        for (int i = 0; i < ans.size(); i++) {
            cout << ans[i];
            if (i != ans.size() - 1) {
                cout << " ";
            }
        }
        cout << endl;
    } else {
        cout << pre << endl;
    }

    return 0;
}

```

时空复杂度

时间复杂度：`O(NlogN + NK)`。排序需要的时间复杂度为O(NlogN)。遍历`lst_sorted`需要`O(N)`的复杂度，每次对`word`进行切片操作需要`O(K)`的复杂度，故遍历过程共需要`O(NK)`的时

间复杂度。总的时间复杂度为两者相加，即 `O(NlogN + NK)` ，如果 `N` 远大于 `K` ，也会退化成 `O(NlogN)` 。

空间复杂度： `O(NM)` 。主要为 `lst_sorted` 的所占空间。

`N` 为单词数目， `M` 为单词平均长度， `K` 为前缀单词 `pre` 的长度。

解法二*

（前缀树解法，不要求掌握，感兴趣的同学可以研究一下）

Python

```python
# 题目：2024E-英文输入法
# 分值：100
# 作者：许老师-闭着眼睛学数理化
# 算法：前缀树
# 代码看不懂的地方，请直接在群上提问

# 构建前缀树节点类
class Trie():
    def __init__(self) -> None:
        self.children = [None] * 52     # 大小写均存在，需要构建长度为52的children列表
        self.isEnd = False              # 结束标识符，True表示当前节点是一个单词的结尾

    # 将单词word加入前缀树的函数
    def addword(self, word):
        node = self
        # 遍历该单词中的所有字符
        for ch in word:
            # 获得ch在children列表中对应的索引
            ch_idx = self.getIdx(ch)
            # 如果对应位置为None
            if node.children[ch_idx] is None:
                # 则为这个ch字符创建一个新的前缀树节点
                node.children[ch_idx] = Trie()
            # 令前缀树节点前进到ch所在的节点
            node = node.children[ch_idx]
        # 完成该单词的添加，设置最后一个字符的节点的结束标识符为True
        node.isEnd = True

    # 根据字符ch获得在children列表中的对应索引的函数
```

```
30      def getIdx(self, ch):
31          # 如果ch是小写，得到26-51的索引
32          if ch.islower():
33              ch_idx = ord(ch) - ord("a") + 26
34          # 如果ch是大写，得到0-25的索引
35          else:
36              ch_idx = ord(ch) - ord("A")
37          return ch_idx
38
39
40      # 根据在children列表中的索引idx获得对应字符ch的函数
41      def getCh(self, idx):
42          # 如果idx大于等于26，是一个小写字母
43          if idx >= 26:
44              ch = chr(idx + ord("a") - 26)
45          # 如果idx小于26，是一个大写字母
46          else:
47              ch = chr(idx + ord("A"))
48          return ch
49
50
51      # 获得前缀prefix最后一个字符所在的节点
52      def getLastNode(self, prefix):
53          node = self
54          for ch in prefix:
55              ch_idx = self.getIdx(ch)
56              if node.children[ch_idx] is None:
57                  return None
58              node = node.children[ch_idx]
59          return node
60
61
62      # 对前缀树进行dfs前序遍历，搜索得到所有后缀
63      def dfs(self, pre, ans, path):
64          node = self
65          # 遇到一个单词结束标识符，将当前path合并为字符串后加入ans
66          if node.isEnd:
67              # 要注意path此时仅仅是后缀，要得到完整的单词字符串还要在前面加上pre
68              ans.append(pre + "".join(path))
69          # 如果node.children存在任意一个非None节点，需要对非空节点继续进行DFS搜索
70          if any(node.children):
71              # 遍历node.children中的所有下一个节点nxt_node
72              for nxt_idx, nxt_node in enumerate(node.children):
73                  # 如果nxt_node不为空，则继续递归地进行DFS搜索
74                  if nxt_node is not None:
75                      # 根据nxt_idx获得对应的字符nxt_ch
76                      nxt_ch = self.getCh(nxt_idx)
```

```
                    # 将字符nxt_ch加在path末尾的结果，作为参数传入nxt_node的dfs递归
                    nxt_node.dfs(pre, ans, path + [nxt_ch])

s = input()
pre = input()

# 初始化列表lst用于存放所有单词
lst = [""]

# 遍历s中的所有字符ch，如果
# 1. ch是字母，则加入到lst最后一个元素的末尾，即延长当前单词
# 2. ch不是字母，说明遇到一个标点符号，结束当前单词的获取，lst的末尾插入一个新的空字符串""
# 这个过程也可以使用正则表达式来完成，不要求掌握，学有余力的同学可以自学一下
for ch in s:
    if ch.isalpha():
        lst[-1] += ch
    else:
        lst.append("")

# 对lst进行去重，因为使用前缀树，所以无需排序
lst = list(set(lst))

# 初始化前缀树根节点
root = Trie()

# 遍历lst中的每一个单词word，构建前缀树
for word in lst:
    root.addword(word)

# 调用前缀树中的getLastNode()方法，得到前缀pre在树中的最后一个节点
lastNode = root.getLastNode(pre)

# 如果lastNode为空，说明在root前缀树中，不存在任何前缀为pre的单词，输出pre
if lastNode is None:
    print(pre)
# 如果lastNode非空，说明在root前缀树中，存在前缀为pre的单词，要找到所有单词
else:
    # 初始化答案数组
    ans = list()
    # 从lastNode开始，调用dfs，找到所有单词，按顺序储存在ans中
    lastNode.dfs(pre, ans, [])
    # 最后将ans用空格隔开合并为字符串后输出
    print(" ".join(ans))
```

Java

```java
import java.util.*;

class TrieNode {
    TrieNode[] children;
    boolean isEnd;

    // 构建前缀树节点类
    public TrieNode() {
        this.children = new TrieNode[52]; // 大小写均存在，需要构建长度为52的children数组
        this.isEnd = false;               // 结束标识符，True表示当前节点是一个单词的结尾
    }

    // 将单词word加入前缀树的函数
    public void addWord(String word) {
        TrieNode node = this;
        // 遍历该单词中的所有字符
        for (char ch : word.toCharArray()) {
            // 获得ch在children数组中对应的索引
            int chIdx = getIdx(ch);
            // 如果对应位置为null
            if (node.children[chIdx] == null) {
                // 则为这个ch字符创建一个新的前缀树节点
                node.children[chIdx] = new TrieNode();
            }
            // 令前缀树节点前进到ch所在的节点
            node = node.children[chIdx];
        }
        // 完成该单词的添加，设置最后一个字符的节点的结束标识符为true
        node.isEnd = true;
    }

    // 根据字符ch获得在children数组中的对应索引的函数
    public int getIdx(char ch) {
        // 如果ch是小写，得到26-51的索引
        if (Character.isLowerCase(ch)) {
            return ch - 'a' + 26;
        } else {
            // 如果ch是大写，得到0-25的索引
            return ch - 'A';
        }
    }

    // 根据在children数组中的索引idx获得对应字符ch的函数
    public char getCh(int idx) {
```

```java
        // 如果idx大于等于26，是一个小写字母
        if (idx >= 26) {
            return (char) (idx + 'a' - 26);
        } else {
            // 如果idx小于26，是一个大写字母
            return (char) (idx + 'A');
        }
    }

    // 获得前缀prefix最后一个字符所在的节点
    public TrieNode getLastNode(String prefix) {
        TrieNode node = this;
        // 遍历prefix中的每个字符
        for (char ch : prefix.toCharArray()) {
            int chIdx = getIdx(ch);
            if (node.children[chIdx] == null) {
                return null; // 如果某个字符不存在，返回null
            }
            node = node.children[chIdx];
        }
        return node; // 返回前缀最后一个字符所在的节点
    }

    // 对前缀树进行dfs前序遍历，搜索得到所有后缀
    public void dfs(String pre, List<String> ans, List<Character> path) {
        // 遇到一个单词结束标识符，将当前path合并为字符串后加入ans
        if (isEnd) {
            StringBuilder sb = new StringBuilder(pre);
            for (char ch : path) {
                sb.append(ch);
            }
            ans.add(sb.toString());
        }
        // 如果children存在任意一个非null节点，需要对非空节点继续进行DFS搜索
        for (int i = 0; i < children.length; i++) {
            if (children[i] != null) {
                // 根据索引获得对应的字符
                char nxtCh = getCh(i);
                // 将字符加在path末尾的结果，作为参数传入递归
                List<Character> newPath = new ArrayList<>(path);
                newPath.add(nxtCh);
                children[i].dfs(pre, ans, newPath);
            }
        }
    }
}

```

```java
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String s = scanner.nextLine();
        String pre = scanner.nextLine();

        // 初始化列表lst用于存放所有单词
        String[] words = s.split("[^a-zA-Z]+");
        Set<String> set = new HashSet<>(Arrays.asList(words));

        List<String> lst = new ArrayList<>(set);
        // 初始化前缀树根节点
        TrieNode root = new TrieNode();

        // 遍历lst中的每一个单词word，构建前缀树
        for (String word : lst) {
            root.addWord(word);
        }

        // 调用前缀树中的getLastNode()方法，得到前缀pre在树中的最后一个节点
        TrieNode lastNode = root.getLastNode(pre);

        // 如果lastNode为空，说明在root前缀树中，不存在任何前缀为pre的单词，输出pre
        if (lastNode == null) {
            System.out.println(pre);
        } else {
            // 如果lastNode非空，说明在root前缀树中，存在前缀为pre的单词，要找到所有单词
            List<String> ans = new ArrayList<>();
            // 从lastNode开始，调用dfs，找到所有单词，按顺序储存在ans中
            lastNode.dfs(pre, ans, new ArrayList<>());
            // 最后将ans用空格隔开合并为字符串后输出
            System.out.println(String.join(" ", ans));
        }
    }
}

```

C++

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>

// 构建前缀树节点类
```

```cpp
class TrieNode {
public:
    TrieNode* children[52];
    bool isEnd;

    TrieNode() {
        for (int i = 0; i < 52; ++i) {
            children[i] = nullptr;
        }
        isEnd = false; // 结束标识符，true表示当前节点是一个单词的结尾
    }

    // 将单词word加入前缀树的函数
    void addWord(const std::string& word) {
        TrieNode* node = this;
        for (char ch : word) {
            int chIdx = getIdx(ch);
            // 如果对应位置为nullptr
            if (node->children[chIdx] == nullptr) {
                // 则为这个ch字符创建一个新的前缀树节点
                node->children[chIdx] = new TrieNode();
            }
            // 令前缀树节点前进到ch所在的节点
            node = node->children[chIdx];
        }
        // 完成该单词的添加，设置最后一个字符的节点的结束标识符为true
        node->isEnd = true;
    }

    // 根据字符ch获得在children数组中的对应索引的函数
    int getIdx(char ch) {
        if (islower(ch)) {
            return ch - 'a' + 26; // 如果ch是小写，得到26-51的索引
        } else {
            return ch - 'A'; // 如果ch是大写，得到0-25的索引
        }
    }

    // 根据在children数组中的索引idx获得对应字符ch的函数
    char getCh(int idx) {
        if (idx >= 26) {
            return idx + 'a' - 26; // 如果idx大于等于26，是一个小写字母
        } else {
            return idx + 'A'; // 如果idx小于26，是一个大写字母
        }
    }

```

```cpp
    // 获得前缀prefix最后一个字符所在的节点
    TrieNode* getLastNode(const std::string& prefix) {
        TrieNode* node = this;
        for (char ch : prefix) {
            int chIdx = getIdx(ch);
            if (node->children[chIdx] == nullptr) {
                return nullptr;
            }
            node = node->children[chIdx];
        }
        return node;
    }

    // 对前缀树进行dfs前序遍历，搜索得到所有后缀
    void dfs(const std::string& pre, std::vector<std::string>& ans, const
std::vector<char>& path) {
        if (isEnd) {
            std::string word = pre;
            for (char ch : path) {
                word += ch;
            }
            ans.push_back(word);
        }
        for (int i = 0; i < 52; ++i) {
            if (children[i] != nullptr) {
                char nxtCh = getCh(i);
                std::vector<char> newPath(path);
                newPath.push_back(nxtCh);
                children[i]->dfs(pre, ans, newPath);
            }
        }
    }
};

int main() {
    std::string s, pre;
    std::getline(std::cin, s);
    std::getline(std::cin, pre);

    // 初始化列表lst用于存放所有单词
    std::unordered_set<std::string> wordSet;
    size_t start = 0;
    while (start < s.size()) {
        while (start < s.size() && !isalpha(s[start])) {
            ++start;
        }
        size_t end = start;
```

```
 99          while (end < s.size() && isalpha(s[end])) {
100              ++end;
101          }
102          if (start < s.size()) {
103              wordSet.insert(s.substr(start, end - start));
104          }
105          start = end + 1;
106      }
107
108      // 对lst进行去重，因为使用前缀树，所以无需排序
109      std::vector<std::string> words(wordSet.begin(), wordSet.end());
110      // 初始化前缀树根节点
111      TrieNode* root = new TrieNode();
112
113      // 遍历lst中的每一个单词word，构建前缀树
114      for (const std::string& word : words) {
115          root->addWord(word);
116      }
117
118      // 调用前缀树中的getLastNode()方法，得到前缀pre在树中的最后一个节点
119      TrieNode* lastNode = root->getLastNode(pre);
120
121      // 如果lastNode为空，说明在root前缀树中，不存在任何前缀为pre的单词，输出pre
122      if (lastNode == nullptr) {
123          std::cout << pre << std::endl;
124      } else { // 如果lastNode非空，说明在root前缀树中，存在前缀为pre的单词，要找到所有
    单词
125          // 初始化答案数组
126          std::vector<std::string> ans;
127          // 从lastNode开始，调用dfs，找到所有单词，按顺序储存在ans中
128          lastNode->dfs(pre, ans, std::vector<char>());
129          for (const std::string& word : ans) {
130              std::cout << word << " ";
131          }
132          std::cout << std::endl;
133      }
134
135      delete root; // Don't forget to release memory
136      return 0;
137  }
138
```

时空复杂度

时间复杂度：`O(NM)`。建树、检查前缀的时间复杂度。

空间复杂度：`O(D)`。

`N` 为单词数目，`M` 为单词平均长度，`D` 为前缀树的节点数，远小于 `NM`。