Solution: Search Suggestions System

Let's solve the Search Suggestions System problem using the Trie pattern.

We'll cover the following

Statement

Given an array of strings called `products` and a word to search, design a system that, when each character of the searched word is typed, suggests at most three product names from `products`. Suggested products should share a common prefix with the searched word. If more than three products exist with a common prefix, return the three product names that appear first in lexicographical order.

Return the suggested products, which will be a list of lists after each character of searched word is typed.

Constraints:

- $1 \leq$ `products.length` ≤ 1000
- $1 \leq$ `products[i].length` ≤ 3000
- $1 \leq$ `sum(products[i].length)` $\leq 2 \times 10^3$
- All the strings of products are unique.
- `products[i]` consists of lowercase English letters.
- $1 \leq$ `searched word.length` ≤ 1000
- The searched word consists of lowercase English letters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

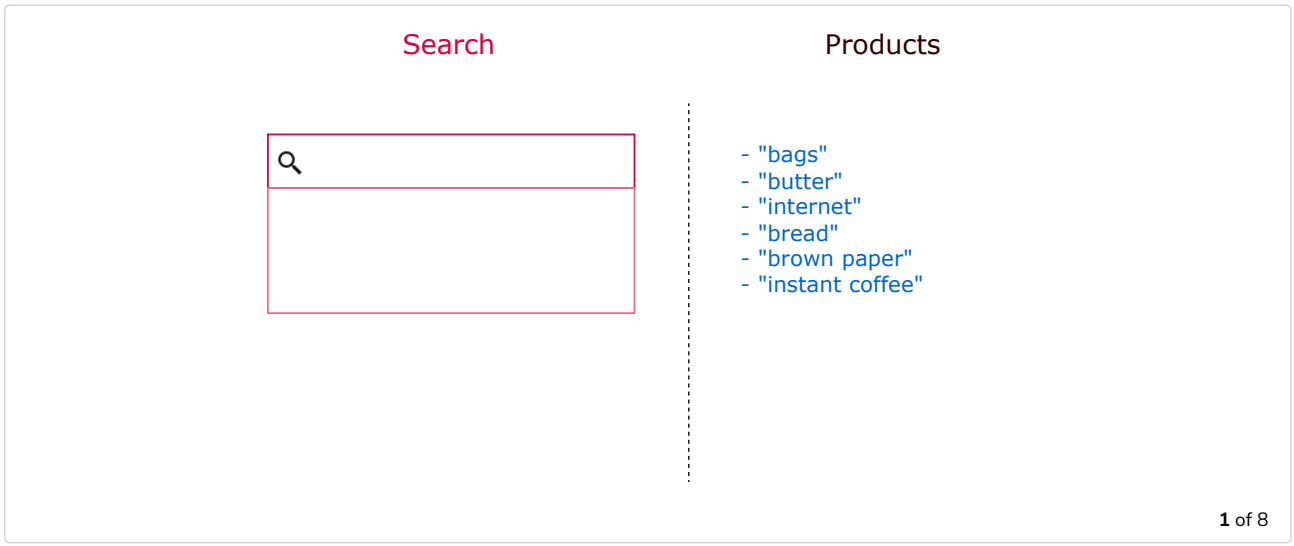
Naive approach

A naive approach would be to first sort the given product data in lexicographic order to help with retrieving three matching products. Next we'll make a substring for every character of the word to be searched. We'll search in the list of product names to check whether the current substring exists or not. If it exists, we'll store the results (containing matching product names) for the current substring. We'll repeat this process until we have traversed the whole list of product names.

It takes $O(n \log n)$ time to sort the given data, where n is the number of product names. Given that m is the number of characters in the search word, there will be m substrings derived from the search word. So, it will take $O(m \times n)$ time to find the matching product names for all the substrings derived from the search word. So, the total time complexity of the search is $O(m \times n + n \log n)$. However, the space complexity of the suggested algorithm is $O(1)$.

Optimized approach using trie

The idea is to reduce the time complexity using the trie pattern. Although it will increase the space complexity a bit, it will help us reduce the time complexity to a great extent. We can feed the products' data into the system and create a trie out of it. Next, we can search for the required product in the recorded data.



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

We'll start by sorting out the products' list since we want our data in lexicographic order. We'll insert the products in the trie by creating new nodes for each new character encountered. Each node of the trie will have the following:

- A `children` dictionary
- A list of words to search

We'll create a new node if the current character key doesn't exist in the `children` dictionary. We'll then append the current product to the search list of the node. If the list's length becomes greater than three, we'll just consider the top three elements and disregard the rest.

The highlighted lines below implement this logic:

Java

main.java

Node.java

```
class SearchSuggestion {

    private Node root = new Node();
    public String printStringWithMarker(String data ,  int pValue)
    {
        String out = "";
        for(int i = 0; i < data.length(); i++)
        {
            if(i == pValue)
            {
                out += "«" + String.valueOf(data.charAt(i)) + "»";
            }
            else
                out += String.valueOf(data.charAt(i));
        }
        return out;
    }
    public void insert(String word) {
        Node node = root;
        int idx = 0;

        for (char ch : word.toCharArray()){
            System.out.println("\t\t" + printStringWithMarker(word, idx));
            idx += 1;
            int index = ch - 'a';
            // Create a new node if char does not exist in children dictionar
            if (node.child[index] == null) {
                System.out.println("\t\t" + ch + " does not exist in the chil
```

Inserting Products

Next, we'll search for the given word in our trie. We'll iterate through every character of the word and check whether the `children` dictionary contains the key of the current character. If there's any data available corresponding to the current character, we'll append it to the result array. Otherwise, we'll append empty lists for the remaining characters of the searched word and terminate the search. In simpler terms, we'll perform prefix matching for all possible substrings of the searched word that start with the first character of the word. For example, if we're searching for "apple", we'll perform prefix matching for "a", "ap", "app", "appl", and "apple".

Java

main.java

Node.java

```
class SearchSuggestion {

    private Node root = new Node();
    public String printStringWithMarker(String data ,  int pValue)
    {
        String out = "";
        for(int i = 0; i < data.length(); i++)
        {
            if(i == pValue)
            {
                out += "«" + String.valueOf(data.charAt(i)) + "»";
            }
            else
                out += String.valueOf(data.charAt(i));
        }
        return out;
    }
    public void insert(String word) {
        Node node = root;
        int idx = 0;
```

```
21          for (char ch : word.toCharArray()){
22              System.out.println("\t\t" + printStringWithMarker(word, idx));
23              idx += 1;
24              int index = ch - 'a';
25              // Create a new node if char does not exist in children dictionar
26              if (node.child[index] == null) {
27                  node.child[index] = new Node();
```

Search Suggestions System

Just the code

Here's the complete solution to this problem:

Java

main.java

Node.java

```
1   class SearchSuggestion {
2
3       private Node root = new Node();
4
5       public void insert(String word) {
6           Node node = root;
7           int idx = 0;
8           for (char ch : word.toCharArray()){
9               idx += 1;
10              int index = ch - 'a';
11              // Create a new node if char does not exist in children dictionar
12              if (node.child[index] == null) {
13                  node.child[index] = new Node();
14              }
15              node = node.child[index];
16                  node.searchWords.pollLast();
19              }
20          }
21      }
22
23      public List<List<String>> search(String searchWord) {
24          List<List<String>> result = new ArrayList<>();
25          Node node = root;
26          for (char ch : searchWord.toCharArray()) {
27              int index = ch - 'a';
28              if (node != null) {
```

Search Suggestions System

Solution summary

To search for a word, we'll first sort our products' list and populate the data in the trie data structure. Then to search for a specific word, we'll iterate through each letter of the searched word and check whether the `children` dictionary contains the key of the current letter or not. If there's such a key, we'll store the result. Else, we'll append empty lists to the result for the remaining characters of the searched word and return.

Time complexity

- **Insert**: In the worst case, each call to the insert function will have a time complexity of $O(w)$, where w is the length of the product name.
- **Search**: The time required to search for a word is $O(m)$, where m is the length of the searched word.

Space complexity

- **Insert**: In the worst case, space complexity will be $O(1)$.
- **Search**: The space complexity is $O(p)$, where p is the number of characters in the list of product names.

Back
Search Suggestions S...

Next
Replace Words

Mark as Completed