<https://leetcode.com/problems/search-suggestions-system/description/>

You are given an array of strings products and a string searchWord.

Design a system that suggests at most three product names from products after each character of searchWord is typed. Suggested products should have common prefix with searchWord. If there are more than three products with a common prefix return the three lexicographically minimums products.

Return a list of lists of the suggested products after each character of searchWord is typed.

**Example 1:**

**Input:** products = ["mobile","mouse","moneypot","monitor","mousepad"], searchWord = "mouse"

**Output:** [["mobile","moneypot","monitor"],["mobile","moneypot","monitor"],["mouse","mousepad"],["mouse","mousepad"],["mouse","mousepad"]]

**Explanation:** products sorted lexicographically = ["mobile","moneypot","monitor","mouse","mousepad"].After typing m and mo all products match and we show user ["mobile","moneypot","monitor"].After typing mou, mous and mouse the system suggests ["mouse","mousepad"].

**Example 2:**

**Input:** products = ["havana"], searchWord = "havana"

**Output:** [["havana"],["havana"],["havana"],["havana"],["havana"],["havana"]]

**Explanation:** The only word "havana" will be always suggested while typing the search word.

**Constraints:**

1 <= products.length <= 1000

1 <= products[i].length <= 3000

1 <= sum(products[i].length) <= 2 \* 10^4

All the strings of products are **unique**.

products[i] consists of lowercase English letters.

1 <= searchWord.length <= 1000

searchWord consists of lowercase English letters.

**Attempt 1: 2025-02-09**

**Solution 1: Sorting + Binary Search (30 min)**

class Solution {

    public List<List<String>> suggestedProducts(String[] products, String searchWord) {

        // Sort the products lexicographically

        Arrays.sort(products);

        List<List<String>> result = new ArrayList<>();

        String prefix = "";

        for(char c : searchWord.toCharArray()) {

            prefix += c;

            int start = binarySearch(products, prefix);

            List<String> suggestions = new ArrayList<>();

            // Collect up to 3 suggestions starting from the found index

            for(int i = start; i < Math.min(start + 3, products.length); i++) {

                // We still need to verify if the word start with given prefix

                // Test case:

                // products = ["bags","baggage","banner","box","cloths"], searchWord = "bags"

                // Output = [["baggage","bags","banner"],["baggage","bags","banner"],["baggage","bags","banner"],["bags","banner","box"]]

                // Expected = [["baggage","bags","banner"],["baggage","bags","banner"],["baggage","bags"],["bags"]]

                // Explain: "banner" not start with given prefix "bag", "banner" and "box" not start with given prefix "bags"

                if(products[i].startsWith(prefix)) {

                    suggestions.add(products[i]);

                } else {

                    break;

                }

            }

            result.add(suggestions);

        }

        return result;

    }

    // Find lower boundary

    private int binarySearch(String[] products, String prefix) {

        int lo = 0;

        int hi = products.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(products[mid].compareTo(prefix) >= 0) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Time Complexity: O(nlogn)

This solution efficiently handles the problem with a time complexity of O(n log n) for sorting and O(m log n)

for the binary search, where n is the number of products and m is the length of the search word.

Space Complexity: O(n)

**Solution 2: Sorting + Trie (30 min)**

class Solution {

    public List<List<String>> suggestedProducts(String[] products, String searchWord) {

        Arrays.sort(products);

        // Build Trie Tree

        TrieNode root = new TrieNode();

        for(String product : products) {

            TrieNode node = root;

            for(char c : product.toCharArray()) {

                if(node.children[c - 'a'] == null) {

                    node.children[c - 'a'] = new TrieNode();

                }

                node = node.children[c - 'a'];

                if(node.suggestions.size() < 3) {

                    node.suggestions.add(product);

                }

            }

        }

        // Query Trie Tree

        List<List<String>> result = new ArrayList<>();

        TrieNode node = root;

        for(char c : searchWord.toCharArray()) {

            if(node != null) {

                node = node.children[c - 'a'];

            }

            result.add(node == null ? new ArrayList<>() : node.suggestions);

        }

        return result;

    }

}

class TrieNode {

    TrieNode[] children;

    List<String> suggestions;

    public TrieNode() {

        this.children = new TrieNode[26];

        this.suggestions = new ArrayList<>();

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(n^2)

Sorting: O(n log n), where n is the number of products.

Trie insertion: O(m \* n), where m is the average length of the products.

Query: O(k), where k is the length of searchWord.

**Solution 3: Heap + Trie (30 min)**

class Solution {

    public List<List<String>> suggestedProducts(String[] products, String searchWord) {

        Arrays.sort(products);

        // Build Trie Tree

        TrieNode root = new TrieNode();

        for(String product : products) {

            TrieNode node = root;

            for(char c : product.toCharArray()) {

                if(node.children[c - 'a'] == null) {

                    node.children[c - 'a'] = new TrieNode();

                }

                node = node.children[c - 'a'];

                if(node.suggestions.size() < 3) {

                    node.addSuggestion(product);

                }

            }

        }

        // Query Trie Tree

        List<List<String>> result = new ArrayList<>();

        TrieNode node = root;

        for(char c : searchWord.toCharArray()) {

            if(node != null) {

                node = node.children[c - 'a'];

            }

            result.add(node == null ? new ArrayList<>() : node.getSuggestion());

        }

        return result;

    }

}

class TrieNode {

    TrieNode[] children;

    PriorityQueue<String> suggestions;

    public TrieNode() {

        this.children = new TrieNode[26];

        this.suggestions = new PriorityQueue<>((a, b) -> b.compareTo(a));

    }

    public void addSuggestion(String str) {

        suggestions.offer(str);

        if(suggestions.size() > 3) {

            suggestions.poll();

        }

    }

    public List<String> getSuggestion() {

        List<String> list = new ArrayList<>(suggestions);

        Collections.sort(list);

        return list;

    }

}

Time Complexity:

Trie insertion: O(m \* n \* log 3), where m is the average length of the products and log 3 is the cost of maintaining the heap.

Query: O(k), where k is the length of searchWord.

Space Complexity:

Trie: O(m \* n) to store all characters of all products.

Heap: O(3 \* m \* n) to store up to 3 products at each node.

**Refer to Deepseek**

To solve LeetCode problem 1268, "Search Suggestions System," we need to provide a list of suggested products based on the characters typed into a search box. The suggestions should be the top 3 lexicographically smallest products that share the same prefix as the typed characters.

Here's a step-by-step Java solution:

Sort the Products: First, sort the products lexicographically.

Use Binary Search: For each character typed, use binary search to find the range of products that match the current prefix.

Collect Suggestions: For each prefix, collect up to 3 suggestions from the sorted list.

Here's the Java code implementing this approach:

class SearchSuggestionsSystem {

public List<List<String>> suggestedProducts(String[] products, String searchWord) {

// Sort the products lexicographically

Arrays.sort(products);

List<List<String>> result = new ArrayList<>();

String prefix = "";

for (char c : searchWord.toCharArray()) {

prefix += c;

int start = binarySearch(products, prefix);

List<String> suggestions = new ArrayList<>();

// Collect up to 3 suggestions starting from the found index

for (int i = start; i < Math.min(start + 3, products.length); i++) {

if (products[i].startsWith(prefix)) {

suggestions.add(products[i]);

} else {

break;

}

}

result.add(suggestions);

}

return result;

}

private int binarySearch(String[] products, String prefix) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].compareTo(prefix) < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return left;

}

public static void main(String[] args) {

SearchSuggestionsSystem solution = new SearchSuggestionsSystem();

String[] products = {"mobile", "mouse", "moneypot", "monitor", "mousepad"};

String searchWord = "mouse";

List<List<String>> result = solution.suggestedProducts(products, searchWord);

for (List<String> list : result) {

System.out.println(list);

}

}

}

**Explanation:**

Sorting: The products array is sorted lexicographically to make it easier to find the range of products that match the prefix.

Binary Search: The binarySearch method finds the first index in the sorted array where the product is greater than or equal to the prefix. This helps in narrowing down the range of products that match the prefix.

Collecting Suggestions: For each prefix, we collect up to 3 products that start with the prefix by iterating from the found index.

**Example:**

For the input:

String[] products = {"mobile", "mouse", "moneypot", "monitor", "mousepad"};

String searchWord = "mouse";

The output will be:

[["mobile", "moneypot", "monitor"], ["mouse", "mousepad"], ["mouse", "mousepad"], ["mouse", "mousepad"], ["mouse", "mousepad"]]

This solution efficiently handles the problem with a time complexity of O(n log n) for sorting and O(m log n) for the binary search, where n is the number of products and m is the length of the search word.

**Binary Search: The binarySearch method finds the first index in the sorted array where the product is greater than or equal to the prefix. This helps in narrowing down the range of products that match the prefix.**

**the statement in code is "products[mid].compareTo(prefix)", how this compareTo method works ?**

The goal of the binary search in this problem is to find the first index in the sorted products array where a product starts with the given prefix. This is done by comparing each product with the prefix lexicographically.

**How Binary Search Works:**

Input: A sorted array of products and a prefix.

Output: The index of the first product that is greater than or equal to the prefix.

Logic:

If the current product is less than the prefix, we search in the right half of the array.

If the current product is greater than or equal to the prefix, we search in the left half of the array.

Result: The left pointer will eventually point to the first product that is greater than or equal to the prefix.

**How compareTo() method Works**

**The compareTo method is part of the String class in Java. It compares two strings lexicographically (i.e., based on the Unicode value of each character)**

int result = string1.compareTo(string2);

If string1 is \*\*less than\*\* string2, compareTo returns a \*\*negative integer\*\*.

If string1 is \*\*equal to\*\* string2, compareTo returns \*\*0\*\*.

If string1 is \*\*greater than\*\* string2, compareTo returns a \*\*positive integer\*\*.

**Example:**

Take s1 as products[i], take s2 as prefix

String s1 = "apple";

String s2 = "banana";

System.out.println(s1.compareTo(s2)); // Output: -1 (apple < banana)

System.out.println(s2.compareTo(s1)); // Output: 1 (banana > apple)

System.out.println(s1.compareTo("apple")); // Output: 0 (apple == apple)

String s1 = "abcd";

String s2 = "abc";

System.out.println(s1.compareTo(s2)); // Output: 1 (abcd > abc)

String s1 = "abc";

String s2 = "abc";

System.out.println(s1.compareTo(s2)); // Output: 0 (abc == abc)

String s1 = "abc";

String s2 = "abd";

System.out.println(s1.compareTo(s2)); // Output: -1 (abc < abd)

String s1 = "aba";

String s2 = "abc";

System.out.println(s1.compareTo(s2)); // Output: -2 (aba < abc)

**Binary Search (Find Lower Boundary)**

Here’s the binary search method from the solution:

private int binarySearch(String[] products, String prefix) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (products[mid].compareTo(prefix) >= 0) {

right = mid - 1; // Search in the left half

} else {

left = mid + 1; // Search in the right half

}

}

return left; // First index where product >= prefix

}

**Example**

String[] products = {"apple", "apricot", "banana", "bat", "battery"};

String prefix = "bat";

**Execution:**

Initial State:

left = 0, right = 4.

First Iteration:

mid = 2, products[2] = "banana".

"banana".compareTo("bat") returns a negative value ("banana" < "bat").

Move left = 3.

Second Iteration:

mid = 3, products[3] = "bat".

"bat".compareTo("bat") returns 0 ("bat" == "bat").

Move right = 2.

Termination:

left = 3, right = 2 (loop ends).

Return left = 3.

The first index where products[index] >= "bat" is 3.

**Why This Works:**

The binary search ensures that we efficiently narrow down the range of products that match the prefix.

**The compareTo method helps us determine whether a product is less than, equal to, or greater than the prefix.**

By returning left, we get the starting index of the range of products that match the prefix.

**Key Points:**

**Binary Search is used to find the lower bound of the prefix in the sorted products array.**

The compareTo method compares strings lexicographically, which is essential for determining the order of products.

This approach is efficient with a time complexity of O(log n) for each binary search.

Let me know if you need further clarification!

**Trie + Sort and Trie + Heap**

**1. Trie + Sort Approach**

**Trie Structure**: A Trie (prefix tree) is used to store the products. Each node in the Trie contains:

A **children** map to store the next characters.

A **suggestion** list to store up to 3 lexicographically smallest products that share the prefix up to that node.

**Sorting**: The products array is sorted lexicographically before inserting into the Trie. This ensures that the suggestion list in each Trie node contains the smallest 3 products in order.

**Insertion**: For each product, we traverse the Trie character by character and add the product to the suggestion list of each node if the list has fewer than 3 products.

**Query**: For each character in searchWord, we traverse the Trie and retrieve the suggestion list from the current node.

**Time Complexity:**

Sorting: O(n log n), where n is the number of products.

Trie insertion: O(m \* n), where m is the average length of the products.

Query: O(k), where k is the length of searchWord.

**Space Complexity:**

Trie: O(m \* n) to store all characters of all products.

**2. Trie + Heap Approach**

**Trie Structure**: Similar to the Trie + Sort approach, but instead of a suggestion list, each node contains a max-heap (h) to store up to 3 lexicographically smallest products.

**Heap**: A max-heap is used to maintain the smallest 3 products. If the heap size exceeds 3, the largest product is removed.

**Insertion**: For each product, we traverse the Trie character by character and add the product to the heap of each node.

**Query**: For each character in searchWord, we traverse the Trie and retrieve the sorted list of products from the heap.

**Time Complexity:**

Trie insertion: O(m \* n \* log 3), where m is the average length of the products and log 3 is the cost of maintaining the heap.

Query: O(k), where k is the length of searchWord.

**Space Complexity:**

Trie: O(m \* n) to store all characters of all products.

Heap: O(3 \* m \* n) to store up to 3 products at each node.

**Implementation**

**1. Trie + Sort**

class Solution {

public List<List<String>> suggestedProducts(String[] products, String searchWord) {

// Sort the products lexicographically

Arrays.sort(products);

// Build the Trie

TrieNode root = new TrieNode();

for (String product : products) {

TrieNode node = root;

for (char c : product.toCharArray()) {

node = node.children.computeIfAbsent(c, k -> new TrieNode());

if (node.suggestions.size() < 3) {

node.suggestions.add(product);

}

}

}

// Query the Trie

List<List<String>> result = new ArrayList<>();

TrieNode node = root;

for (char c : searchWord.toCharArray()) {

if (node != null) {

node = node.children.get(c);

}

result.add(node == null ? Collections.emptyList() : node.suggestions);

}

return result;

}

private static class TrieNode {

Map<Character, TrieNode> children = new HashMap<>();

List<String> suggestions = new ArrayList<>();

}

}

**2. Trie + Heap**

class Solution {

public List<List<String>> suggestedProducts(String[] products, String searchWord) {

// Build the Trie

TrieNode root = new TrieNode();

for (String product : products) {

TrieNode node = root;

for (char c : product.toCharArray()) {

node = node.children.computeIfAbsent(c, k -> new TrieNode());

node.addSuggestion(product);

}

}

// Query the Trie

List<List<String>> result = new ArrayList<>();

TrieNode node = root;

for (char c : searchWord.toCharArray()) {

if (node != null) {

node = node.children.get(c);

}

result.add(node == null ? Collections.emptyList() : node.getSuggestions());

}

return result;

}

private static class TrieNode {

Map<Character, TrieNode> children = new HashMap<>();

PriorityQueue<String> heap = new PriorityQueue<>((a, b) -> b.compareTo(a));

public void addSuggestion(String product) {

heap.offer(product);

if (heap.size() > 3) {

heap.poll();

}

}

public List<String> getSuggestions() {

List<String> suggestions = new ArrayList<>(heap);

Collections.sort(suggestions);

return suggestions;

}

}

}

**Comparison**

|  |  |  |
| --- | --- | --- |
| **Approach** | **Advantages** | **Disadvantages** |
| **Trie + Sort** | Simple and easy to implement. | Requires sorting the entire  products  array upfront. |
| **Trie + Heap** | Dynamically maintains the top 3 suggestions without sorting the entire array. | Slightly more complex due to heap operations. |

**Which One to Use?**

Use Trie + Sort if the number of products is small or if sorting is not a bottleneck.

**Use Trie + Heap if you want to avoid sorting the entire array and prefer a more dynamic approach.**

Both approaches are efficient and work well for the problem. Choose based on your specific requirements and constraints.