My Idea

1. Sort products
2. Using hashtable to store vector<string> with corresponding prefix
3. Traverse searched string and get vector<string> from hashtable

World Idea

**Solution**

**Approach 1: Binary Search**

**Intuition**

Since the question asks for the result in a sorted order, let's start with sorting products.  
An advantage that comes with sorting is Binary Search, we can binary search for the prefix. Once we locate the first match of prefix, all we need to do is to add the next 3 words into the result (if there are any), since we sorted the words beforehand.

**Algorithm**

1. Sort the input products.
2. Iterate each character of the searchWord adding it to the prefix to search for.
3. After adding the current character to the prefix binary search for the prefix in the input.
4. Add next 3 strings from the current binary search start index till the prefix remains same.
5. Another optimization that can be done is reducing the binary search space to current start index (This is due to the fact that adding more characters to the prefix will make the next search result's index be at least > current search's index).
6. class Solution {
7. public:
8. vector<vector<string>> suggestedProducts(vector<string> &products,
9. string searchWord) {
10. sort(products.begin(), products.end());
11. vector<vector<string>> result;
12. int start, bsStart = 0, n=products.size();
13. string prefix;
14. for (char &c : searchWord) {
15. prefix += c;
16. // Get the starting index of word starting with `prefix`.
17. start = lower\_bound(products.begin() + bsStart, products.end(), prefix) - products.begin();
18. // Add empty vector to result.
19. result.push\_back({});
20. // Add the words with the same prefix to the result.
21. // Loop runs until `i` reaches the end of input or 3 times or till the
22. // prefix is same for `products[i]` Whichever comes first.
23. for (int i = start; i < min(start + 3, n) && !products[i].compare(0, prefix.length(), prefix); i++)
24. result.back().push\_back(products[i]);
25. // Reduce the size of elements to binary search on since we know
26. bsStart = start;
27. }
28. return result;
29. }
30. };

**Complexity Analysis**

* Time complexity : O(nlog(n))+O(mlog(n))O(nlog(n)) + O(mlog(n))*O*(*nlog*(*n*))+*O*(*mlog*(*n*)). Where n is the length of products and m is the length of the search word. Here we treat string comparison in sorting as O(1)O(1)*O*(1). O(nlog(n))O(nlog(n))*O*(*nlog*(*n*)) comes from the sorting and O(mlog(n))O(mlog(n))*O*(*mlog*(*n*)) comes from running binary search on products m times.
  + In Java there is an additional complexity of O(m2)O(m^2)*O*(*m*2) due to Strings being immutable, here m is the length of searchWord.
* Space complexity : Varies between O(1)O(1)*O*(1) and O(n)O(n)*O*(*n*) where n is the length of products, as it depends on the implementation used for sorting. We ignore the space required for output as it does not affect the algorithm's space complexity. See [Internal details of std::sort](https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/).  
  Space required for output is O(m)O(m)*O*(*m*) where m is the length of the search word.

**Approach 2: Trie + DFS**

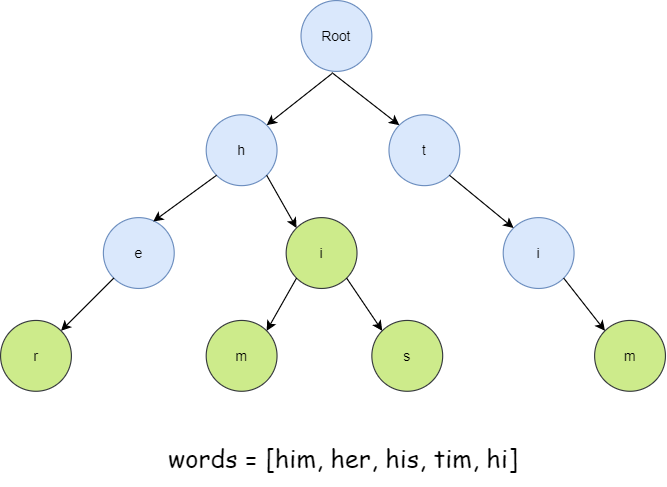
**Intuition**

Whenever we come across questions with multiple strings, it is best to think if [Trie](https://en.wikipedia.org/wiki/Trie" \t "_blank) can help us. What we need here is a way to search for all the words with given prefix, this is a well known problem that trie can solve. The question also asks for a sorted results, if you look closely a trie word is represented by it's preorder traversal. It is also worth noting that a preorder traversal of a trie will always result in a sorted traversal of results, thus all we need to do is limit the word traversal to 3.

Questions using Trie:

[79. Word Search](https://leetcode.com/problems/word-search)

[211. Design Add and Search Words Data Structure](https://leetcode.com/problems/design-add-and-search-words-data-structure)

  
*Figure 1. A trie made from words*

**Algorithm**

1. Create a Trie from the given products input.
2. Iterate each character of the searchWord adding it to the prefix to search for.
3. After adding the current character to the prefix traverse the trie pointer to the node representing prefix.
4. Now traverse the tree from curr pointer in a preorder fashion and record whenever we encounter a complete word.
5. Limit the result to 3 and return dfs once reached this limit.
6. Add the words to the final result.

// Custom class Trie with function to get 3 words starting with given prefix

class Trie

{

    // Node definition of a trie

    struct Node {

        bool isWord = false;

        vector<Node \*> children{vector<Node \*>(26, NULL)};

    } \* Root, \*curr;

    // Runs a DFS on trie starting with given prefix and adds all the words in the result, limiting result size to 3

    void dfsWithPrefix(Node \* curr, string & word, vector<string> & result) {

        if (result.size() == 3)

            return;

        if (curr->isWord)

            result.push\_back(word);

        // Run DFS on all possible paths.

        for (char c = 'a'; c <= 'z'; c++)

            if (curr->children[c - 'a']) {

                word += c;

                dfsWithPrefix(curr->children[c - 'a'], word, result);

                word.pop\_back();

            }

    }

public:

    Trie() {

        Root = new Node();

    }

    // Inserts the string in trie.

    void insert(string & s) {

        // Points curr to the root of trie.

        curr = Root;

        for (char &c : s) {

            if (!curr->children[c - 'a'])

                curr->children[c - 'a'] = new Node();

            curr = curr->children[c - 'a'];

        }

        // Mark this node as a completed word.

        curr->isWord = true;

    }

    vector<string> getWordsStartingWith(string & prefix) {

        curr = Root;

        vector<string> result;

        // Move curr to the end of prefix in its trie representation.

        for (char &c : prefix) {

            if (!curr->children[c - 'a'])

                return result;

            curr = curr->children[c - 'a'];

        }

        dfsWithPrefix(curr, prefix, result);

        return result;

    }

};

class Solution {

public:

    vector<vector<string>> suggestedProducts(vector<string> &products,

                                             string searchWord) {

        Trie trie=Trie();

        vector<vector<string>> result;

        // Add all words to trie.

        for(string &w:products)

            trie.insert(w);

        string prefix;

        for (char &c : searchWord) {

            prefix += c;

            result.push\_back(trie.getWordsStartingWith(prefix));

        }

        return result;

    }

};

**Complexity Analysis**

* Time complexity : O(M)O(M)*O*(*M*) to build the trie where M is total number of characters in products For each prefix we find its representative node in O(len(prefix))O(\text{len(prefix)})*O*(len(prefix)) and dfs to find at most 3 words which is an O(1) operation. Thus the overall complexity is dominated by the time required to build the trie.
  + In Java there is an additional complexity of O(m2)O(m^2)*O*(*m*2) due to Strings being immutable, here m is the length of searchWord.
* Space complexity : O(26n)=O(n)O(26n)=O(n)*O*(26*n*)=*O*(*n*). Here n is the number of nodes in the trie. 26 is the alphabet size.  
  Space required for output is O(m)O(m)*O*(*m*) where m is the length of the search word.