Solution

Approach 1: Horizontal scanning

**Intuition**

For a start we will describe a simple way of finding the longest prefix shared by a set of strings *LCP*(*S*1​…*Sn*​). We will use the observation that :

*LCP*(*S*1​…*Sn*​) = *LCP*(*LCP*(*LCP*(*S*1​, *S*2​), *S*3​), …*Sn*​)

**Algorithm**

To employ this idea, the algorithm iterates through the strings [*S*1​…*Sn*​], finding at each iteration i*i* the longest common prefix of strings *LCP*(*S*1​…*Si*​) When  *LCP*(*S*1​…*Si*​) is an empty string, the algorithm ends. Otherwise after *n* iterations, the algorithm returns  *LCP*(*S*1​…*Sn*​).

public String longestCommonPrefix(String[] strs) {

if (strs.length == 0) return "";

String prefix = strs[0];

for (int i = 1; i < strs.length; i++)

while (strs[i].indexOf(prefix) != 0) {

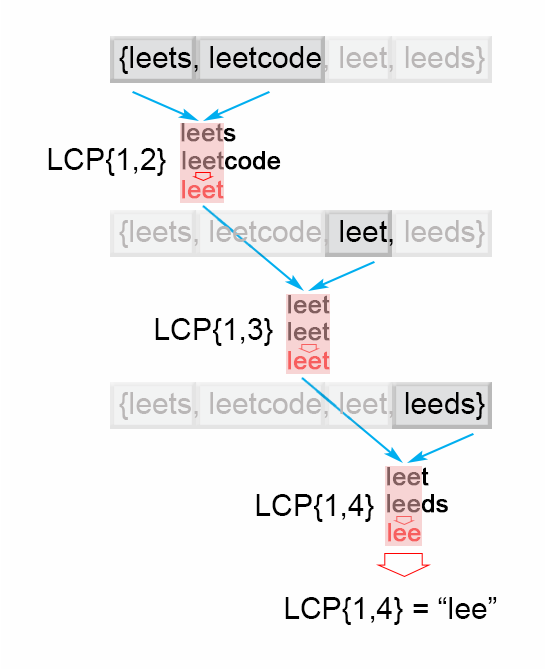
prefix = prefix.substring(0, prefix.length() - 1);

if (prefix.isEmpty()) return "";

}

return prefix;

}



*Figure 1. Finding the longest common prefix (Horizontal scanning)*

**Complexity Analysis**

* Time complexity :  *O*(*S*) , where S is the sum of all characters in all strings.

In the worst case all n*n* strings are the same. The algorithm compares the string *S*1 with the other strings  [*S*2​…*Sn*​] There are *S* character comparisons, where *S* is the sum of all characters in the input array.

* Space complexity : *O*(1). We only used constant extra space.

Approach 2: Vertical scanning

**Algorithm**

Imagine a very short string is the common prefix at the end of the array. The above approach will still do *S* comparisons. One way to optimize this case is to do vertical scanning. We compare characters from top to bottom on the same column (same character index of the strings) before moving on to the next column.

**Complexity Analysis**

* Time complexity : *O*(*S*) , where S is the sum of all characters in all strings. In the worst case there will be *n* equal strings with length m*m* and the algorithm performs  *S*=*m*⋅*n* character comparisons. Even though the worst case is still the same as [Approach 1](https://leetcode.com/problems/longest-common-prefix/solution/#approach-1-horizontal-scanning), in the best case there are at most  *n*⋅*minLen* comparisons where *minLen* is the length of the shortest string in the array.
* Space complexity :  *O*(1). We only used constant extra space.

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

for (int i = 0; i < strs[0].length() ; i++){

char c = strs[0].charAt(i);

for (int j = 1; j < strs.length; j ++) {

if (i == strs[j].length() || strs[j].charAt(i) != c)

return strs[0].substring(0, i);

}

}

return strs[0];

}

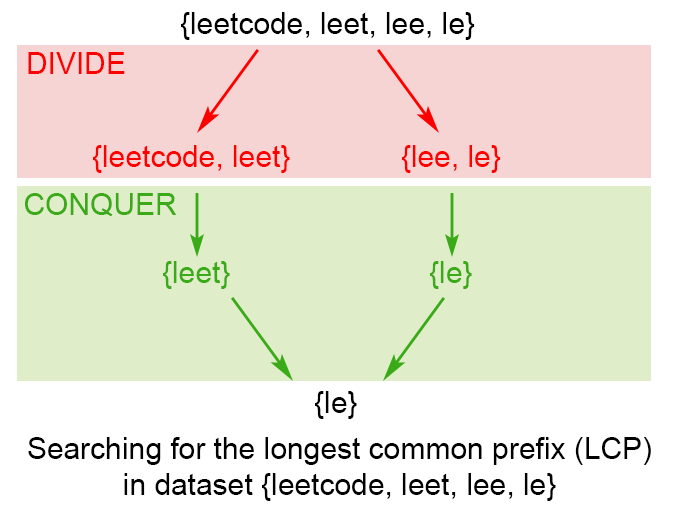
Approach 3: Divide and conquer

**Intuition**

The idea of the algorithm comes from the associative property of LCP operation. We notice that :  *LCP*(*S*1​…*Sn*​)=*LCP*(*LCP*(*S*1​…*Sk*​),*LCP*(*Sk*+1​…*Sn*​)) , where  *LCP*(*S*1​…*Sn*​) is the longest common prefix in set of strings  [*S*1​…*Sn*​] ,  1<*k*<*n*

**Algorithm**

To apply the observation above, we use divide and conquer technique, where we split the  *LCP*(*Si*​…*Sj*​) problem into two subproblems  *LCP*(*Si*​…*Smid*​) and  *LCP*(*Smid*+1​…*Sj*​), where mid is 2*i*+*j*​. We use their solutions lcpLeft and lcpRight to construct the solution of the main problem *LCP*(*Si*​…*Sj*​). To accomplish this we compare one by one the characters of lcpLeft and lcpRight till there is no character match. The found common prefix of lcpLeft and lcpRight is the solution of the *LCP*(*Si*​…*Sj*​).



*Figure 2. Finding the longest common prefix of strings using divide and conquer technique*

**Complexity Analysis**

In the worst case we have n*n* equal strings with length m*m*

* Time complexity :  *O*(*S*), where *S* is the number of all characters in the array, *S*=*m*⋅*n* Time complexity is 2⋅*T*(2*n*​)+*O*(*m*). Therefore time complexity is *O*(*S*). In the best case this algorithm performs *O*(*minLen*⋅*n*) comparisons, where *minLen* is the shortest string of the array
* Space complexity :  *O*(*m*⋅log*n*)

There is a memory overhead since we store recursive calls in the execution stack. There are log*n* recursive calls, each store need m*m* space to store the result, so space complexity is *O*(*m*⋅log*n*)

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

return longestCommonPrefix(strs, 0 , strs.length - 1);

}

private String longestCommonPrefix(String[] strs, int l, int r) {

if (l == r) {

return strs[l];

}

else {

int mid = (l + r)/2;

String lcpLeft = longestCommonPrefix(strs, l , mid);

String lcpRight = longestCommonPrefix(strs, mid + 1,r);

return commonPrefix(lcpLeft, lcpRight);

}

}

String commonPrefix(String left,String right) {

int min = Math.min(left.length(), right.length());

for (int i = 0; i < min; i++) {

if ( left.charAt(i) != right.charAt(i) )

return left.substring(0, i);

}

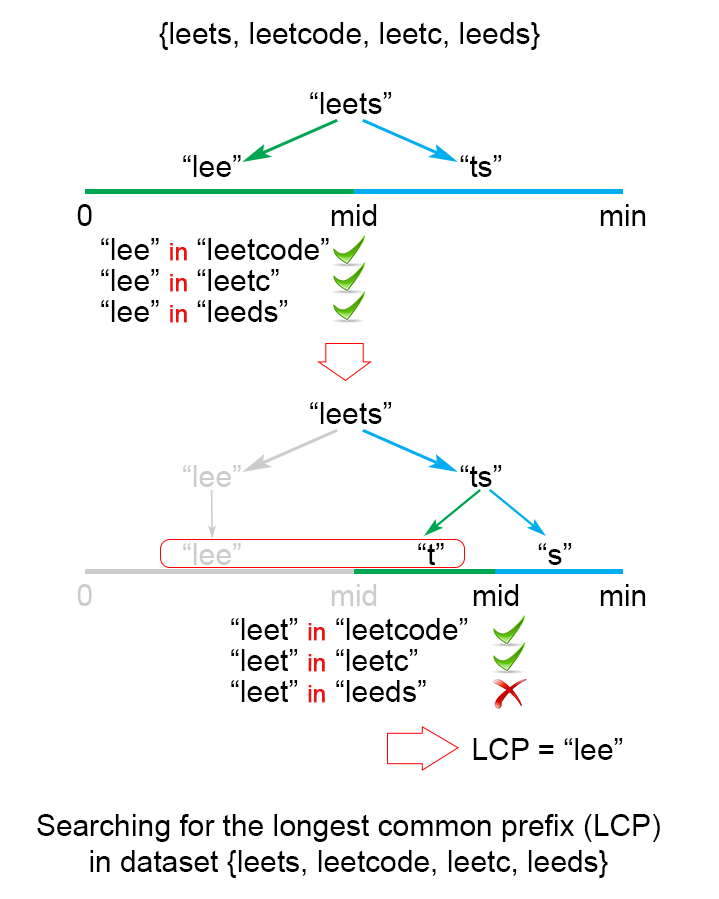
return left.substring(0, min);

}

Approach 4: Binary search

The idea is to apply binary search method to find the string with maximum value L, which is common prefix of all of the strings. The algorithm searches space is the interval (0…*minLen*), where minLen is minimum string length and the maximum possible common prefix. Each time search space is divided in two equal parts, one of them is discarded, because it is sure that it doesn't contain the solution. There are two possible cases:

* S[1...mid] is not a common string. This means that for each j > i S[1..j] is not a common string and we discard the second half of the search space.
* S[1...mid] is common string. This means that for for each i < j S[1..i] is a common string and we discard the first half of the search space, because we try to find longer common prefix.



*Figure 3. Finding the longest common prefix of strings using binary search technique*

**Complexity Analysis**

In the worst case we have *n* equal strings with length *m*

* Time complexity :  *O*(*S*⋅log*m*), where *S* is the sum of all characters in all strings.

The algorithm makes log*m* iterations, for each of them there are  *S*=*m*⋅*n* comparisons, which gives in total  *O*(*S*⋅log*m*) time complexity.

* Space complexity :  *O*(1). We only used constant extra space.

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0)

return "";

int minLen = Integer.MAX\_VALUE;

for (String str : strs)

minLen = Math.min(minLen, str.length());

int low = 1;

int high = minLen;

while (low <= high) {

int middle = (low + high) / 2;

if (isCommonPrefix(strs, middle))

low = middle + 1;

else

high = middle - 1;

}

return strs[0].substring(0, (low + high) / 2);

}

private boolean isCommonPrefix(String[] strs, int len){

String str1 = strs[0].substring(0,len);

for (int i = 1; i < strs.length; i++)

if (!strs[i].startsWith(str1))

return false;

return true;

}

Further Thoughts / Follow up

Let's take a look at a slightly different problem:

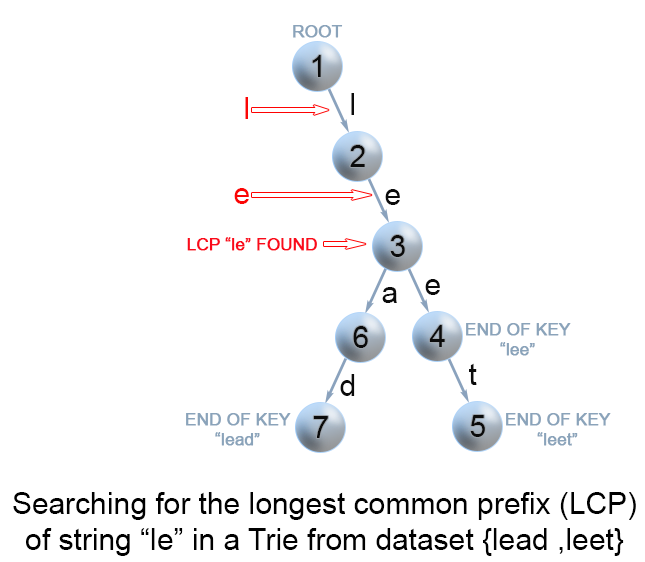
Given a set of keys [*S*1​,*S*2​…*Sn*​], find the longest common prefix among a string q and S. This LCP query will be called frequently.

We could optimize LCP queries by storing the set of keys S in a Trie. For more information about Trie, please see this article [Implement a trie (Prefix trie)](https://leetcode.com/articles/implement-trie-prefix-tree/). In a Trie, each node descending from the root represents a common prefix of some keys. But we need to find the longest common prefix of a string q and all key strings. This means that we have to find the deepest path from the root, which satisfies the following conditions:

* it is prefix of query string q
* each node along the path must contain only one child element. Otherwise the found path will not be a common prefix among all strings.
* the path doesn't comprise of nodes which are marked as end of key. Otherwise the path couldn't be a prefix a of key which is shorter than itself.

**Algorithm**

The only question left, is how to find the deepest path in the Trie, that fulfills the requirements above. The most effective way is to build a trie from  [*S*1​…*Sn*​] strings. Then find the prefix of query string q in the Trie. We traverse the Trie from the root, till it is impossible to continue the path in the Trie because one of the conditions above is not satisfied.



*Figure 4. Finding the longest common prefix of strings using Trie*

**Complexity Analysis**

In the worst case query *q* has length m*m* and it is equal to all *n* strings of the array.

* Time complexity : preprocessing *O*(*S*), where *S* is the number of all characters in the array, LCP query *O*(*m*).

Trie build has  *O*(*S*) time complexity. To find the common prefix of *q* in the Trie takes in the worst case *O*(*m*).

* Space complexity : *O*(*S*). We only used additional *S* extra space for the Trie.