**14. Longest Common Prefix**

**Question**

Write a function to find the longest common prefix string amongst an array of strings.

**Quick Navigation**

* [Solution](https://leetcode.com/articles/longest-common-prefix/#solution)
  + [Approach #1 (Horizontal scanning)](https://leetcode.com/articles/longest-common-prefix/#approach-1-horizontal-scanning)
  + [Approach #2 (Vertical scanning)](https://leetcode.com/articles/longest-common-prefix/#approach-2-vertical-scanning)
  + [Approach #3 (Divide and conquer)](https://leetcode.com/articles/longest-common-prefix/#approach-3-divide-and-conquer)

**Solution**

**Approach #1 (Horizontal scanning)**

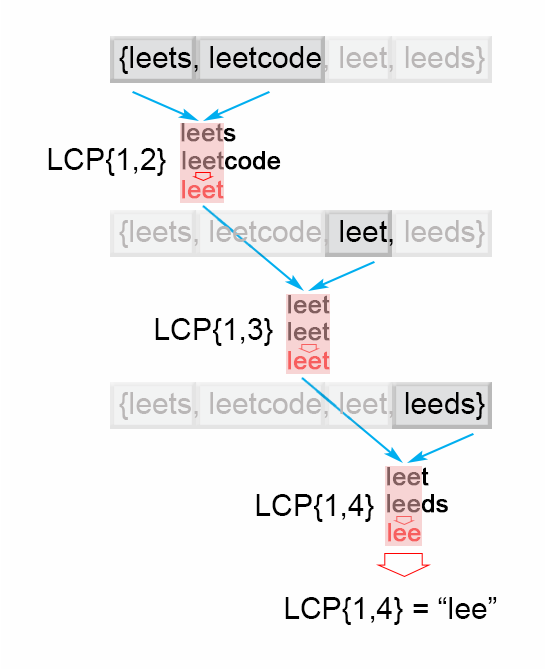
**Intuition**

For a start we will describe a simple way of finding the longest prefix shared by a set of strings LCP(S1…Sn)LCP(S\_1 \ldots S\_n)LCP(S​1​​…S​n​​). We will use the observation that :

LCP(S1…Sn)=LCP(LCP(LCP(S1,S2),S3),…Sn)LCP(S\_1 \ldots S\_n) = LCP(LCP(LCP(S\_1, S\_2),S\_3),\ldots S\_n)LCP(S​1​​…S​n​​)=LCP(LCP(LCP(S​1​​,S​2​​),S​3​​),…S​n​​)

**Algorithm**

To employ this idea, the algorithm iterates through the strings [S1…Sn][S\_1 \ldots S\_n][S​1​​…S​n​​], finding at each iteration iii the longest common prefix of strings LCP(S1…Si)LCP(S\_1 \ldots S\_i)LCP(S​1​​…S​i​​) When LCP(S1…Si)LCP(S\_1 \ldots S\_i)LCP(S​1​​…S​i​​) is an empty string, the algorithm ends. Otherwise after nnn iterations, the algorithm returns LCP(S1…Sn)LCP(S\_1 \ldots S\_n)LCP(S​1​​…S​n​​).



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

**Java**

**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**;**

**}**

**Complexity Analysis**

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

In the worst case all nnn strings are the same. The algorithm compares the string S1S1S1 with the other strings [S2…Sn][S\_2 \ldots S\_n][S​2​​…S​n​​] There are SSS character comparisons, where SSS is the sum of all characters in the input array.

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

**Approach #2 (Vertical scanning)**

**Algorithm**

Imagine a very short string is at the end of the array. The above approach will still do SSS 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.

**Java**

**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**];**

**}**

**Complexity Analysis**

* Time complexity : O(S)O(S)O(S) , where S is the sum of all characters in all strings. In the worst case there will be nnn equal strings with length mmm and the algorithm performs S=m∗nS = m\*nS=m∗n character comparisons. Even though the worst case is still the same as Approach #1, in the best case there are at most n∗minLenn\*minLenn∗minLen comparisons where minLenminLenminLen is the length of the shortest string in the array.
* Space complexity : O(1)O(1)O(1). We only used constant extra space.

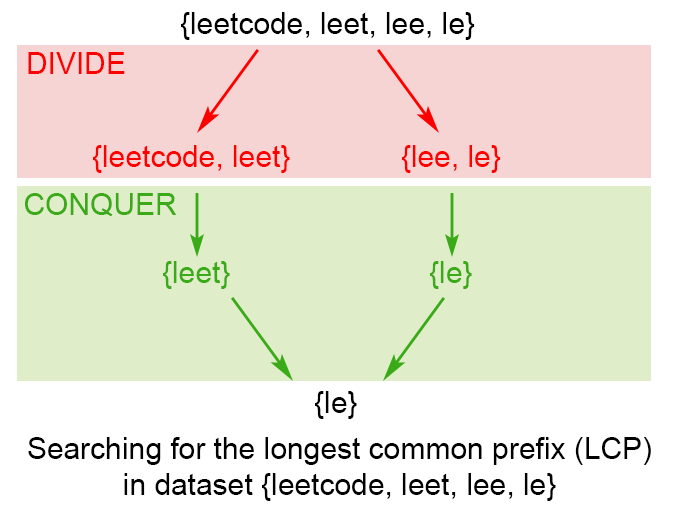
**Approach #3 (Divide and conquer)**

**Intuition**

The idea of the algorithm comes from the associative property of LCP operation. We notice that : LCP(S1…Sn)=LCP(LCP(S1…Sk),LCP(Sk+1…Sn))LCP(S\_1 \ldots S\_n) = LCP(LCP(S\_1 \ldots S\_k), LCP (S\_{k+1} \ldots S\_n))LCP(S​1​​…S​n​​)=LCP(LCP(S​1​​…S​k​​),LCP(S​k+1​​…S​n​​)) , where LCP(S1…Sn)LCP(S\_1 \ldots S\_n)LCP(S​1​​…S​n​​) is the longest common prefix in set of strings [S1…Sn][S\_1 \ldots S\_n][S​1​​…S​n​​] , 1<k<n1 < k < n1<k<n

**Algorithm**

To apply the observation above, we use divide and conquer technique, where we split the LCP(Si…Sj)LCP(S\_i \ldots S\_j)LCP(S​i​​…S​j​​) problem into two subproblems LCP(Si…Smid)LCP(S\_i \ldots S\_{mid})LCP(S​i​​…S​mid​​) and LCP(Smid+1…Sj)LCP(S\_{mid+1} \ldots S\_j)LCP(S​mid+1​​…S​j​​), where mid is i+j2\frac{i + j}{2}​2​​i+j​​. We use their solutions lcpLeft and lcpRight to construct the solution of the main problem LCP(Si…Sj)LCP(S\_i \ldots S\_j)LCP(S​i​​…S​j​​). 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)LCP(S\_i \ldots S\_j)LCP(S​i​​…S​j​​).



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

**Java**

**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**);**

**}**

**Complexity Analysis**

In the worst case we have nnn equal strings with length mmm

* Time complexity : O(S)O(S)O(S), where SSS is the number of all characters in the array, S=m∗nS = m\*nS=m∗n Time complexity is T(n)=2T(n2)+O(m)T(n) = 2 T(\frac{n}{2}) + O(m)T(n)=2T(​2​​n​​)+O(m). Therefore time complexity is O(S)O(S)O(S). In the best case this algorithm performs O(minLen∗n)O(minLen\*n)O(minLen∗n) comparisons, where minLenminLenminLen is the shortest string of the array
* Space complexity : O(m∗log(n))O(m\*log(n))O(m∗log(n))

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