<https://leetcode.com/problems/longest-common-prefix/>

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

If there is no common prefix, return an empty string "".

**Example 1:**

Input: strs = ["flower","flow","flight"]

Output: "fl"

**Example 2:**

Input: strs = ["dog","racecar","car"]

Output: ""

Explanation: There is no common prefix among the input strings.

**Constraints:**

1 <= strs.length <= 200

0 <= strs[i].length <= 200

strs[i] consists of only lowercase English letters.

**Attempt 1: 2023-08-13**

**Solution 1: Trie (30 min)**

class Solution {

    TrieNode root = new TrieNode();

    public String longestCommonPrefix(String[] strs) {

        for(String str : strs) {

            addStr(str);

        }

        TrieNode node = root;

        // To find longest common prefix, randomly pick any string

        // among strs array is fine, because the longest common prefix

        // won't exceed any string

        int len = strs.length;

        return search(strs[0], len);

    }

    public void addStr(String str) {

        TrieNode node = root;

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

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

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

            }

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

            node.count++;

        }

        //node.isEnd = true;

    }

    public String search(String str, int len) {

        StringBuilder sb = new StringBuilder();

        TrieNode node = root;

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

            // Only when current character's corresponding TrieNode' count

            // match total string number in the array(len), we will add

            // current character into final result, because it means this

            // character shared between all strings

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

                sb.append(c);

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

            } else {

                // If count not match total string number

                break;

            }

        }

        return sb.toString();

    }

}

class TrieNode {

    TrieNode[] children;

    // No need 'isEnd' since to find longest common prefix

    // for any string in the given string array we at most

    // can go through all characters in that string, 'isEnd'

    // will have no usage during go through process

    //boolean isEnd;

    // Used to judge if the character shared between all

    // strings in the array

    int count;

    public TrieNode() {

        this.children = new TrieNode[26];

        //this.isEnd = false;

        this.count = 0;

    }

}

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.

**Refer to**

<https://leetcode.com/problems/longest-common-prefix/editorial/>

**Further Thoughts / Follow up**

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

Given a set of keys S = [S1,S2…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:

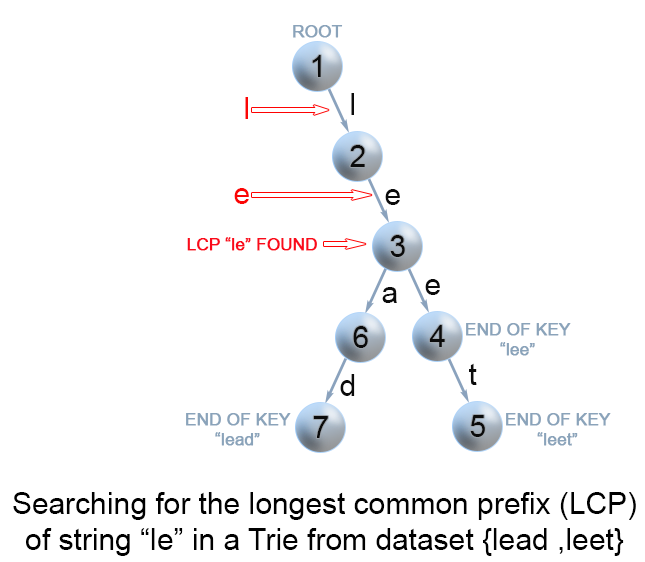
it is prefix of query string q

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 [S1…Sn] strings. Then find the prefix of query string qin 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*

public String longestCommonPrefix(String q, String[] strs) {

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

        return "";

    if (strs.length == 1)

        return strs[0];

    Trie trie = new Trie();

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

        trie.insert(strs[i]);

    }

    return trie.searchLongestPrefix(q);

}

class TrieNode {

    // R links to node children

    private TrieNode[] links;

    private final int R = 26;

    private boolean isEnd;

    // number of children non null links

    private int size;

    public void put(char ch, TrieNode node) {

        links[ch -'a'] = node;

        size++;

    }

    public int getLinks() {

        return size;

    }

    //assume methods containsKey, isEnd, get, put are implemented as it is described

  //in  https://leetcode.com/articles/implement-trie-prefix-tree/)

}

public class Trie {

    private TrieNode root;

    public Trie() {

        root = new TrieNode();

    }

//assume methods insert, search, searchPrefix are implemented as it is described

//in  https://leetcode.com/articles/implement-trie-prefix-tree/)

    private String searchLongestPrefix(String word) {

        TrieNode node = root;

        StringBuilder prefix = new StringBuilder();

        for (int i = 0; i < word.length(); i++) {

            char curLetter = word.charAt(i);

            if (node.containsKey(curLetter) && (node.getLinks() == 1) && (!node.isEnd())) {

                prefix.append(curLetter);

                node = node.get(curLetter);

            }

            else

                return prefix.toString();

        }

        return prefix.toString();

    }

}

**Complexity Analysis**

In the worst case query q has length 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.

**Solution 2: Vertical Scanning (10 min)**

class Solution {

    public String longestCommonPrefix(String[] strs) {

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

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

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

                // Either strs[j] reach the end or same position character

                // in strs[j] not equal to strs[0]'s one, break out

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

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

                }

            }

        }

        return strs[0];

    }

}

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 and the algorithm performs S=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⋅minLen comparisons where minLen is the length of the shortest string in the array.

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

**Refer to**

<https://leetcode.com/problems/longest-common-prefix/editorial/>

**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 SS*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.

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), where S is the sum of all characters in all strings.

In the worst case there will be n equal strings with length 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/editorial/#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.

**Solution 3: Horizontal  Scanning (10 min)**

class Solution {

    public String longestCommonPrefix(String[] strs) {

        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;

    }

}

**Refer to**

<https://leetcode.com/problems/longest-common-prefix/editorial/>

**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)

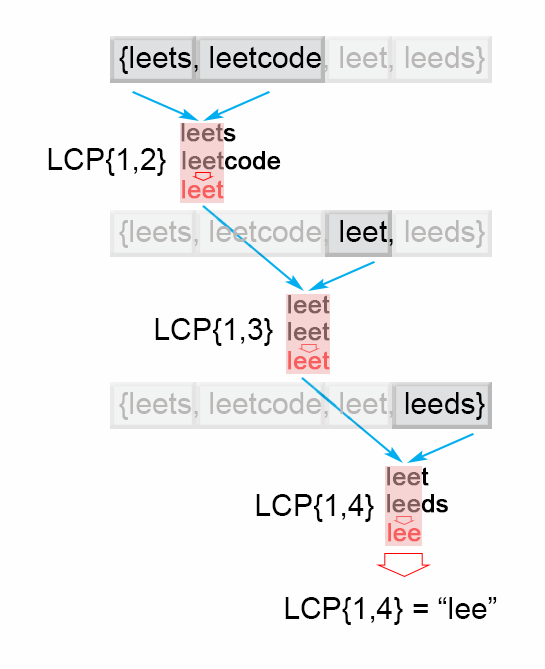
We will use the observation that :

LCP(S1…Sn)=LCP(LCP(LCP(S1,S2),S3),…Sn)

**Algorithm**

To employ this idea, the algorithm iterates through the strings [S1…Sn], finding at each iteration ii*i*the longest common prefix of strings LCP(S1…Si)

When LCP(S1…Si) is an empty string, the algorithm ends. Otherwise after n iterations, the algorithm returns LCP(S1…Sn)



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

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), where S is the sum of all characters in all strings.

In the worst case all n strings are the same. The algorithm compares the string S1 with the other strings [S2…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.

**Solution 4: Binary Search (10 min)**

class Solution {

    public String longestCommonPrefix(String[] strs) {

        int minLen = Integer.MAX\_VALUE;

        for(String str : strs) {

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

        }

        int lo = 1;

        int hi = minLen;

        while(lo <= hi) {

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

            // Use auxiliary helper method to determine if current

            // length of substring of any string (e.g strs[0]) is

            // a common prefix for all other strings

            if(isCommonPrefix(strs, mid)) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        // Use 'hi' is following pattern 'Find Upper Boundary'

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

    }

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

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

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

            if(!strs[i].startsWith(str)) {

                return false;

            }

        }

        return true;

    }

}

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.

**Refer to**

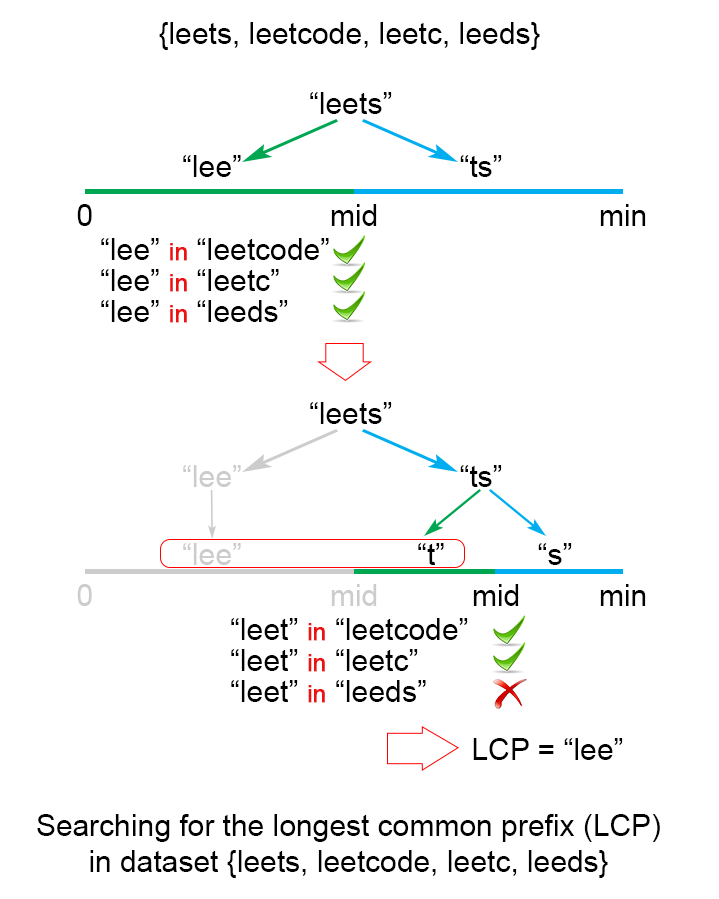
<https://leetcode.com/problems/longest-common-prefix/editorial/>

**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*

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;

}

**Complexity Analysis**

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

**Solution 5: Divide and Conquer (10 min)**

class Solution {

    public String longestCommonPrefix(String[] strs) {

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

    }

    private String helper(String[] strs, int left, int right) {

        // Base condition

        if(left == right) {

            return strs[left];

        }

        // Divide

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

        String lcpleft = helper(strs, left, mid);

        String lcpright = helper(strs, mid + 1, right);

        // Conquer

        return commonPrefix(lcpleft, lcpright);

    }

    private String commonPrefix(String lcpleft, String lcpright) {

        int minLen = Math.min(lcpleft.length(), lcpright.length());

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

            if(lcpleft.charAt(i) != lcpright.charAt(i)) {

                return lcpleft.substring(0, i);

            }

        }

        return lcpleft.substring(0, minLen);

    }

}

Time complexity : O(S), where S is the number of all characters in the array, S=m⋅n

Time complexity is 2⋅T(n/2)+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 space to store the result, so space complexity is O(m⋅log⁡n)

**Refer to**

<https://leetcode.com/problems/longest-common-prefix/editorial/>

**Approach 3: Divide and conquer**

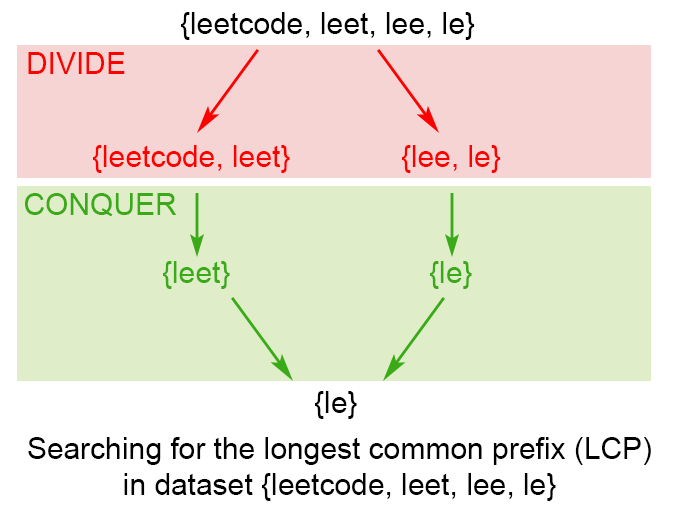
**Intuition**

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)), where LCP(S1…Sn) is the longest common prefix in set of strings [S1…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 (i+j)/2. 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*

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 n equal strings with length m

Time complexity : O(S), where S is the number of all characters in the array, S=m⋅n

Time complexity is 2⋅T(n/2)+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

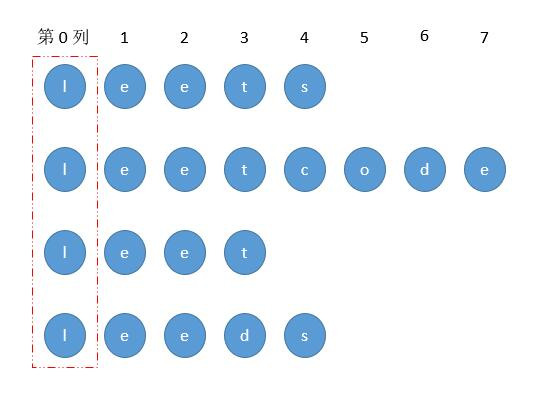
Space complexity : O(m⋅log⁡n)

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

**Also refer to**

<https://leetcode.wang/leetCode-14-Longest-Common-Prefix.html>

**解法一 垂直比较**



我们把所有字符串垂直排列，然后一列一列的比较，直到某一个字符串到达结尾或者该列字符不完全相同。

下边看一下我的代码，看起来比较多

//这个函数判断 index 列的字符是否完全相同

public boolean isSameAtIndex(String[] strs, int index) {

    int i = 0;

    while (i < strs.length - 1) {

        if (strs[i].charAt(index) == strs[i + 1].charAt(index)) {

            i++;

        } else {

            return false;

        }

    }

    return true;

}

public String longestCommonPrefix(String[] strs) {

    if (strs.length == 0)

        return "";

    //得到最短的字符串的长度

    int minLength = Integer.MAX\_VALUE;

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

        if (strs[i].length() < minLength) {

            minLength = strs[i].length();

        }

    }

    int j = 0;

    //遍历所有列

    for (; j < minLength; j++) {

        //如果当前列字符不完全相同，则结束循环

        if (!isSameAtIndex(strs, j)) {

            break;

        }

    }

    return strs[0].substring(0, j);

}

下边看一下，官方的代码

public String longestCommonPrefix(String[] strs) {

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

    //遍历所有列

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

        char c = strs[0].charAt(i); // 保存 i 列第 0 行的字符便于后续比较

        //比较第 1,2,3... 行的字符和第 0 行是否相等

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

            /\*\*

            \* i == strs[j].length() 表明当前行已经到达末尾

            \* strs[j].charAt(i) != c  当前行和第 0 行字符不相等

            \* 上边两种情况返回结果

            \*/

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

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

        }

    }

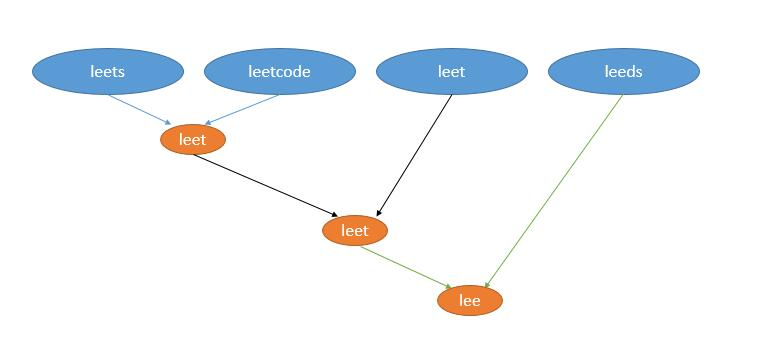
    return strs[0];

}

时间复杂度：最坏的情况就是 n 个 长度为 m 的完全一样的字符串，假设 S 是所有字符的和，那么 S = m \* n，时间复杂度就是 O（S）。当然正常情况下并不需要比较所有字符串，最多比较 n \* minLen 个字符就可以了。

空间复杂度：O（1），常数个额外空间。

**解法二 水平比较**



我们将字符串水平排列，第 0 个和第 1 个字符串找最长子串，结果为 leet，再把结果和第 2 个字符串比较，结果为 leet，再把结果和第 3 个字符串比较，结果为 lee，即为最终结果。

public String longestCommonPrefix3(String[] strs) {

        if (strs.length == 0)

            return "";

        String prefix = strs[0]; // 保存结果

        // 遍历每一个字符串

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

            // 找到上次得到的结果 prefix 和当前字符串的最长子串

            int minLen = Math.min(prefix.length(), strs[i].length());

            int j = 0;

            for (; j < minLen; j++) {

                if (prefix.charAt(j) != strs[i].charAt(j)) {

                    break;

                }

            }

            prefix = prefix.substring(0, j);

        }

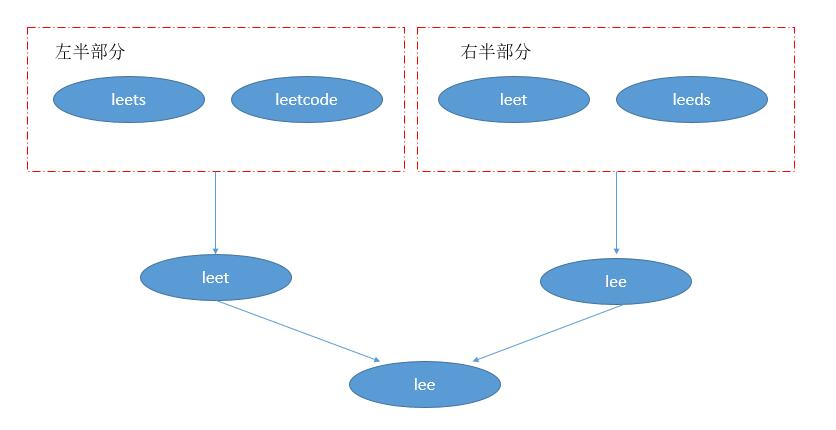
        return prefix;

    }

时间复杂度：最坏情况和解法一是一样，n 个长度为 m 的完全相同的字符，就要比较所有的字符 S，S = n \* m 。但对于正常情况，处于最短字符串前的字符串依旧要比较所有字符，而不是最短字符串个字符，相对于解法一较差。

空间复杂度：O（1）。

**解法三 递归**



我们把原来的数组分成两部分，求出左半部分的最长公共前缀，求出右半部分的最长公共前缀，然后求出的两个结果再求最长公共前缀，就是最后的结果了。

求左半部分的最长公共前缀，我们可以继续把它分成两部分，按照上边的思路接着求。然后一直分成两部分，递归下去。

直到该部分只有 1 个字符串，那么最长公共子串就是它本身了，直接返回就可以了

public String longestCommonPrefix(String[] strs) {

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

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

}

//递归不断分成两部分

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);

  }

}

//求两个结果的最长公共前缀

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);

}

时间复杂度：

空间复杂度：

每次遇到递归的情况，总是有些理不清楚，先空着吧。

**总**

进行了垂直比较和水平比较，又用到了递归，[solution](https://leetcode.com/problems/longest-common-prefix/solution/) 里还介绍了二分查找，感觉这里用二分查找有些太僵硬了，反而使得时间复杂度变高了。还介绍了前缀树，这里后边遇到再总结吧。