ECS 140A Homework 1 – Problem 2

1 Python

Step 1: Algorithm/Pseudocode

```
find_lcp(arr):
    # if arr is empty, return empty string
    # treat first item in arr as temp longest common prefix
    # iterate over rest of the array items
        # compare temp lcp and current item in iteration and track common prefix between them
        # if common prefix is shorter, update temp lcp to the shorter one
    # return lcp
```

Step 2: Actual Code

```
def find_lcp(arr):
    if len(arr) == 0:
        return ""
    lcp = arr[0]
    for s in arr[1:]:
        shorter_str = s if len(s) <= len(lcp) else lcp
        temp = ""
        for i in range(0, len(shorter_str)):
            if shorter_str[i] == lcp[i]:
                temp += shorter_str[i]
            else:
                break
        if len(temp) < len(lcp):
            lcp = temp
    return lcp
```

Step 3: Working Code

There were no syntax errors, so the initial working code was the same as the previous step.

Step 4: Debug Process

```
(base) Annas-MacBook-Pro-2:hw1 annachen$ python3 lcp.py
['apple', 'app', 'aple', 'appl']: app
[]:
['']:
['abc']: abc
['abc', 'xyz']:
['zzzzz', 'zz', 'zzzz']: zz
['bamboo', 'bamboozled']: bamboo
['bamboo', 'bamboozled', 'bambam']: bamb
```

Note that the second test case is an empty array whereas the third test case is a 1-item array of an empty string. All results are correct except for the first test case. The first test case results in `app` due to a lapse in logic. Originally, when I compared the 2 strings (next string in array vs lcp) in the inner for loop, I thought to use the shorter length string to iterate over for the indices. However, a major flaw appears if the current lcp is shorter than the next string to compare (or every string left to compare). It ends up being the case that the current lcp is compared against itself instead of the next string in the array. To fix this, I iterated the indices over the next string in the array regardless of length and checked that the index to be used is valid in the current lcp.

```
def find_lcp(arr):
    if len(arr) == 0:
        return ""
    lcp = arr[0]
    for s in arr[1:]:
        temp = ""
        for i in range(0, len(s)):
            if i >= len(lcp) or s[i] != lcp[i]:
                break
            else:
                temp += s[i]
        if len(temp) < len(lcp):
            lcp = temp
    return lcp
```

The following shows the correct output after running the fixed code:

```
achen00@ad3.ucdavis.edu@pc20:~/ecs140a/hw1$ python3 lcp.py
['apple', 'app', 'aple', 'appl']: ap
[]:
['']:
['abc']: abc
['abc', 'xyz']:
['zzzzz', 'zz', 'zzzz']: zz
['bamboo', 'bamboozled']: bamboo
['bamboo', 'bamboozled', 'bambam']: bamb
```

Step 5: Add Documentation

```
def find_lcp(arr):
    '''
    Takes an array of strings as input and finds the longest common prefix of all of the strings
    '''
    if len(arr) == 0:
        return ""
    lcp = arr[0] # treat first item as current longest common prefix
    for s in arr[1:]:
        temp = "" # to keep track of common prefix in current 2-string comparison
        for i in range(0, len(s)):
            if i >= len(lcp) or s[i] != lcp[i]: # if index out of range or characters not the same
                break
            else:
                temp += s[i]
        if len(temp) < len(lcp): # update lcp if new word compared results in a shorter one
            lcp = temp
    return lcp
```

Step 6: Extra Test Cases Used

- `[]`
- `[""]`
- `["abc"]`
- `["abc", "xyz"]`
- `["zzzzz", "zz", "zzzz"]`
- `["bamboo", "bamboozled"]`
- `["bamboo", "bamboozled", "bambam"]`

2 C++

Step 2: Actual Code

```
#include <iostream>
#include <string>
#include <vector>

std::string find_lcp(std::vector<std::string>& arr) {
    if (arr.size() == 0)
        return "";
    std::string lcp = arr[0];
    for (std::string s : arr) {
        std::string temp = "";
        for (size_t i = 0; i < s.length(); i++) {
            if (i >= lcp.length() || s[i] != lcp[i])
                break;
            else
                temp += s[i];
        }
        if (temp.length() < lcp.length())
            lcp = temp;
    }
    return lcp;
}
```

Step 3: Working Code

There were no syntax errors, so initial working code was the same as the previous step.

Step 4: Debug Process

All test cases (given and extra) passed and program behaved as expected.

```
achen00@ad3.ucdavis.edu@pc20:~/ecs140a/hw1$ ./lcp
[apple, app, aple, appl]: ap
[]:
[]:
[abc]: abc
[abc, xyz]:
[zzzzz, zz, zzzz]: zz
[bamboo, bamboozled]: bamboo
[bamboo, bamboozled, bambam]: bamb
```

Note that the second test case is an empty array whereas the third test case is a 1-item array of an empty string.

Step 5: Add Documentation

```
#include <iostream>
#include <string>
#include <vector>

/*
Takes an array of strings as input and finds the longest common prefix of all of the strings
*/
```

```
std::string find_lcp(std::vector<std::string>& arr) {
    if (arr.size() == 0)
        return "";
    std::string lcp = arr[0]; // treat first item as current longest common prefix
    for (std::string s : arr) {
        std::string temp = ""; // to keep track of common prefix in current 2-string comparison
        for (size_t i = 0; i < s.length(); i++) {
            // if index out of range or characters not the same
            if (i >= lcp.length() || s[i] != lcp[i])
                break;
            else
                temp += s[i];
        }
        if (temp.length() < lcp.length()) // update lcp if new word compared results in a shorter one
            lcp = temp;
    }
    return lcp;
}
```

3 Rust

Step 2: Actual Code

```
fn find_lcp(arr: Vec<String>) -> String {
    if arr.len() == 0 {
        return "".to_string();
    }
    let mut lcp = arr[0];
    for s in arr {
        let mut temp: String = "".to_string();
        for i in 0..s.len() {
            if i >= lcp.len() || s[i] != lcp[i] {
                break;
            } else {
                temp.push(s[i]);
            }
        }
        if temp.len() < lcp.len() {
            lcp = temp;
        }
    }
    return lcp;
}
```

```
error[E0277]: the type `String` cannot be indexed by `usize`
 --> src/main.rs:9:34
  |
9 |             if i >= lcp.len() || s[i] != lcp[i] {
  |                                  ^^^^ `String` cannot be indexed by `usize`
  |
  = help: the trait `Index<usize>` is not implemented for `String`
```

This error was raised because strings cannot be indexed in Rust. I consulted Stack Overflow and found that I would need to use the `.chars()` iterator and index from there:
https://stackoverflow.com/questions/24542115/how-to-index-a-string-in-rust

```
(base) Annas-MacBook-Pro-2:q2-rust annachen$ cargo run
   Compiling q2-rust v0.1.0 (/Users/annachen/ecs140a/hw1/q2-rust)
error[E0507]: cannot move out of index of `Vec<String>`
 --> src/main.rs:5:19
  |
5 |     let mut lcp = arr[0];
  |                   ^^^^^^
  |                   |
  |                   move occurs because value has type `String`, which does not implement the `Copy` trait
  |                   help: consider borrowing here: `&arr[0]`

error[E0382]: borrow of moved value: `arr`
   --> src/main.rs:8:21
    |
1   | fn lcp(arr: Vec<String>) -> String {
    |        --- move occurs because `arr` has type `Vec<String>`, which does not implement the `Copy` trait
...
6   |     for s in arr {
    |              ---
    |              |
    |              `arr` moved due to this implicit call to `.into_iter()`
    |              help: consider borrowing to avoid moving into the for loop: `&arr`
7   |         let mut temp: String = "".to_string();
8   |         for i in 0..arr.len() {
    |                     ^^^^^^^^^ value borrowed here after move
    |
note: this function takes ownership of the receiver `self`, which moves `arr`
```

These errors were raised because we do not have ownership to the variables underlined, yet the code tries moving them from some memory space to another. To fix these, I used `.clone()` to create copies of those variables.

Step 3: Working Code

```
fn lcp(arr: Vec<String>) -> String {
    if arr.len() == 0 {
        return "".to_string();
    }
    let mut lcp = arr[0].clone();
    for s in arr.clone() {
        let mut temp: String = "".to_string();
        for i in 0..s.len() {
            if i >= lcp.len() || s.chars().nth(i).unwrap() != lcp.chars().nth(i).unwrap() {
                break;
            } else {
                temp.push(s.chars().nth(i).unwrap());
            }
        }
        if temp.len() < lcp.len() {
            lcp = temp;
        }
    }
    return lcp;
}
```

Step 4: Debug Process

All test cases (given and extra) passed and program behaved as expected.

```
achen00@ad3.ucdavis.edu@pc20:~/ecs140a/hw1$ ./lcp
[apple, app, aple, appl]: ap
[]:
[]:
[abc]: abc
[abc, xyz]:
[zzzzz, zz, zzzz]: zz
[bamboo, bamboozled]: bamboo
[bamboo, bamboozled, bambam]: bamb
```

Note that the second test case is an empty array whereas the third test case is a 1-item array of an empty string.

Step 5: Add Documentation

```
/// Takes an array of strings as input and finds the longest common prefix of all of the strings
fn find_lcp(arr: Vec<String>) -> String {
    if arr.len() == 0 {
        return "".to_string();
    }
    let mut lcp = arr[0].clone(); // treat first item as current longest common prefix
    for s in arr.clone() {
        let mut temp: String = "".to_string(); // to keep track of common prefix in current 2-string co
        for i in 0..s.len() {
            // if index out of range or characters not the same
```

```
            if i >= lcp.len() || s.chars().nth(i).unwrap() != lcp.chars().nth(i).unwrap() {
                break;
            } else {
                temp.push(s.chars().nth(i).unwrap());
            }
        }
        if temp.len() < lcp.len() { // update lcp if new word compared results in a shorter one
            lcp = temp;
        }
    }
    return lcp;
}
```