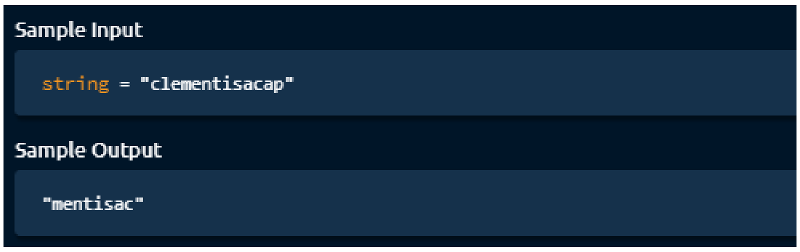
Topic - Strings

Difficulty hard: Longest Substring Without Duplication

Write a function that takes in a string and returns its longest substring without duplicate characters. You can assume that there will only be one longest substring without duplication.



Sample Input

```
string = "clementisacap"
```

Sample Output

```
"mentisac"
```

Hint:

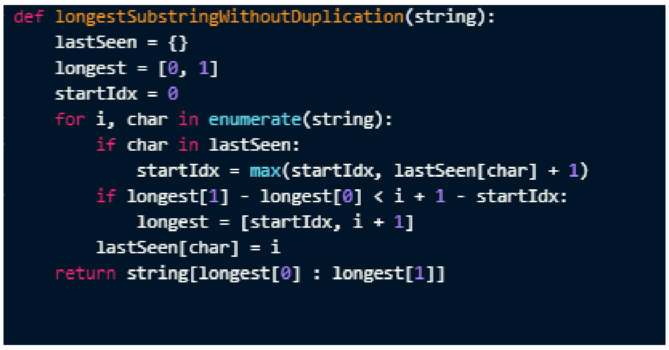
Try traversing the input string and storing the last position at which you see each character in a hash table. How can this help you solve the given problem?

As you traverse the input string, keep track of a starting index variable. This variable, as its name suggests, should represent the most recent index from which you could start a substring with no duplicate characters, ending at your current index. Use the hash table mentioned in Hint #1 to update this variable correctly, and update the longest substring as you go.

Optimal Space & Time Complexity

O(n) time | O(min(n, a)) space - where n is the length of the input string and a is the length of the character alphabet represented in the input string

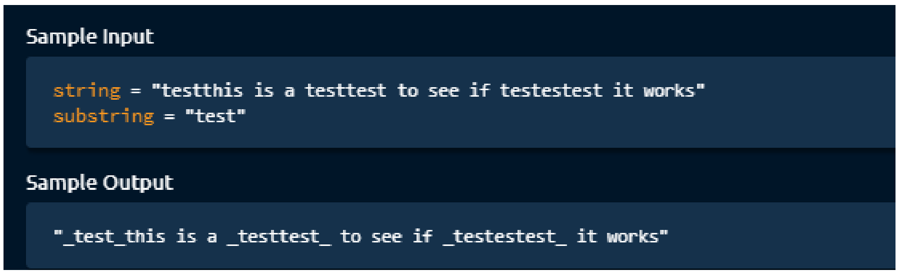
Soluton:



```python
def longestSubstringWithoutDuplication(string):
    lastSeen = {}
    longest = [0, 1]
    startIdx = 0
    for i, char in enumerate(string):
        if char in lastSeen:
            startIdx = max(startIdx, lastSeen[char] + 1)
        if longest[1] - longest[0] < i + 1 - startIdx:
            longest = [startIdx, i + 1]
        lastSeen[char] = i
    return string[longest[0] : longest[1]]
```

Difficulty Medium: Underscorify Substring

Write a function that takes in two strings: a main string and a potential substring of the main string. The function should return a version of the main string with every instance of the substring in it wrapped between underscores. If two or more instances of the substring in the main string overlap each other or sit side by side, the underscores relevant to these substrings should only appear on the far left of the leftmost substring and on the far right of the rightmost substring. If the main string doesn't contain the other string at all, the function should return the main string intact.



Sample Input

```
string = "testthis is a testtest to see if testestest it works"
substring = "test"
```

Sample Output

```
"_test_this is a _testtest_ to see if _testestest_ it works"
```

Hint:

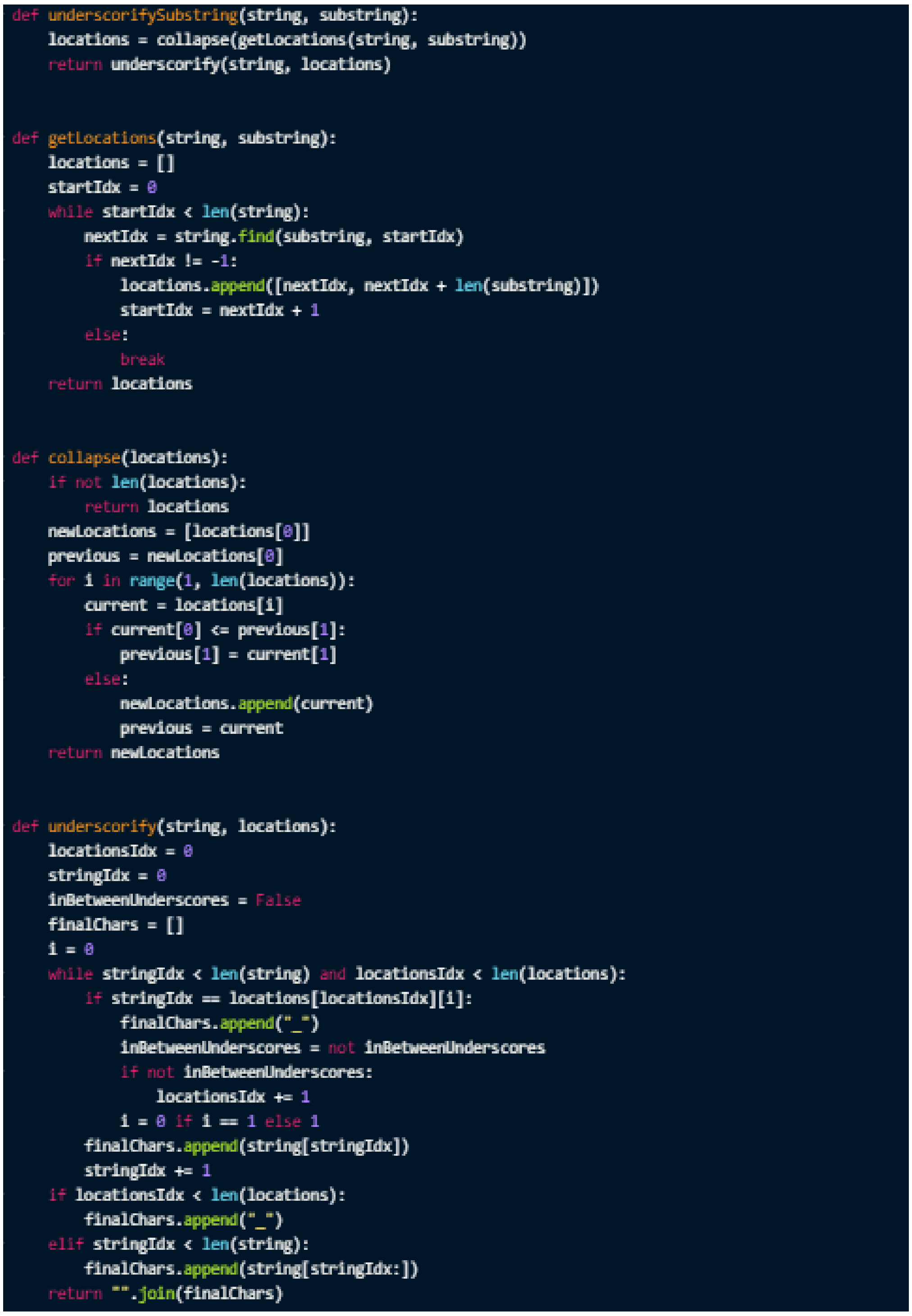
The first thing you need to do to solve this question is to get the locations of all instances of the substring in the main string. Try traversing the main string one character at a time and calling whatever substring-matching function is built into the language you're working in. Store a 2D array of locations, where each subarray holds the starting and ending indices of a specific instance of the substring in the main string. The second thing you need to do is to "collapse" the 2D array mentioned in Hint . In essence, you need to merge the locations of substrings that overlap each other or sit next to each other. Traverse the 2D array mentioned in Hint and build a new 2D array that holds these "collapsed" locations.Finally, you need to create a new string with

underscores added in the correct positions. Construct this new string by traversing the main string and the 2D array mentioned in Hint at the same time. You might have to keep track of when you are "in between" underscores in order to correctly traverse the 2D array.

Optimal Space & Time Complexity

Average case: O(n + m) | O(n) space - where n is the length of the main string and m is the length of the substring

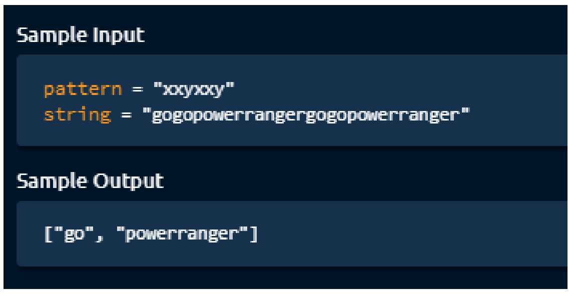
Solution



```python
def underscorifySubstring(string, substring):
    locations = collapse(getLocations(string, substring))
    return underscorify(string, locations)


def getLocations(string, substring):
    locations = []
    startIdx = 0
    while startIdx < len(string):
        nextIdx = string.find(substring, startIdx)
        if nextIdx != -1:
            locations.append([nextIdx, nextIdx + len(substring)])
            startIdx = nextIdx + 1
        else:
            break
    return locations


def collapse(locations):
    if not len(locations):
        return locations
    newLocations = [locations[0]]
    previous = newLocations[0]
    for i in range(1, len(locations)):
        current = locations[i]
        if current[0] <= previous[1]:
            previous[1] = current[1]
        else:
            newLocations.append(current)
            previous = current
    return newLocations


def underscorify(string, locations):
    locationsIdx = 0
    stringIdx = 0
    inBetweenUnderscores = False
    finalChars = []
    i = 0
    while stringIdx < len(string) and locationsIdx < len(locations):
        if stringIdx == locations[locationsIdx][i]:
            finalChars.append("_")
            inBetweenUnderscores = not inBetweenUnderscores
            if not inBetweenUnderscores:
                locationsIdx += 1
            i = 0 if i == 1 else 1
        finalChars.append(string[stringIdx])
        stringIdx += 1
    if locationsIdx < len(locations):
        finalChars.append("_")
    elif stringIdx < len(string):
        finalChars.append(string[stringIdx:])
    return "".join(finalChars)
```

Difficulty Medium: Pattern Matcher

You're given two non-empty strings. The first one is a pattern consisting of only "x"s and / or "y"s; the other one is a normal string of alphanumeric characters. Write a function that checks whether the normal string matches the pattern. A string S0 is said to match a pattern if replacing all "x"s in the pattern with some non-empty substring S1 of S0 and replacing all "y"s in the pattern with some non-empty substring S2 of S0 yields the same string S0. If the input string doesn't match the input pattern, the function should return an empty array; otherwise, it should return an array holding the strings S1 and S2 that represent "x" and "y" in the normal string, in that order. If the pattern doesn't contain any "x"s or "y"s, the respective letter should be represented by an empty string in the final array that you return. You can assume that there will never be more than one pair of strings S1 and S2 that appropriately represent "x" and "y" in the normal string.


Sample Input

```
pattern = "xxyxxy"
string = "gogopowerrangergogopowerranger"
```

Sample Output

```
["go", "powerranger"]
```

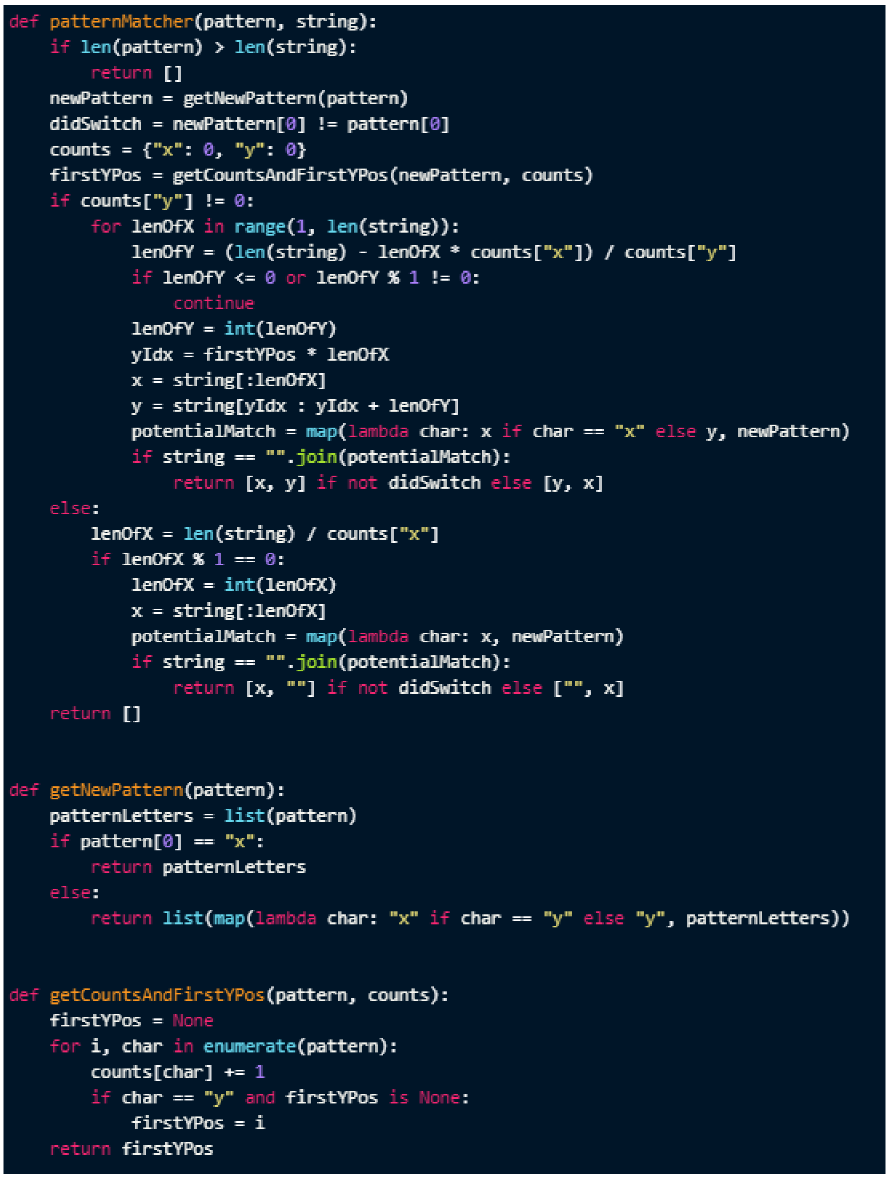
Hint:

Start by checking if the pattern starts with an "x". If it doesn't, consider generating a new pattern that swaps all "x"s for "y"s and vice versa; this might greatly simplify the rest of your algorithm. Make sure to keep track of whether or not you do this swap, as your final answer will be affected by it. Use a hash table to store the number of "x"s and "y"s that appear in the pattern, and keep track of the position of the first "y". Knowing how many "x"s and "y"s appear in the pattern, paired with the length of the main string which you have access to, will allow you to quickly test out various possible lengths for "x" and "y". Knowing where the first "y" appears in the pattern will allow you to actually generate potential substrings. Traverse the main string and try different combinations of substrings that could represent "x" and "y". For each potential combination, map the new pattern mentioned in Hint and see if it matches the main string.

Optimal Space & Time Complexity

O(n^2 + m) time | O(n + m) space - where n is the length of the main string and m is the length of the pattern

Solution:



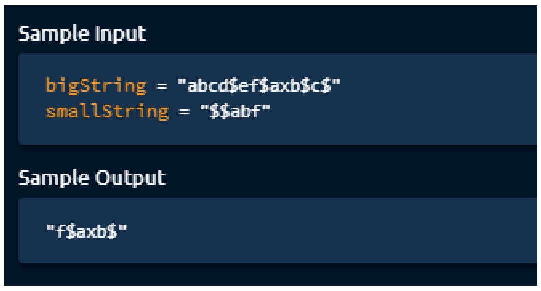
```python
def patternMatcher(pattern, string):
    if len(pattern) > len(string):
        return []
    newPattern = getNewPattern(pattern)
    didSwitch = newPattern[0] != pattern[0]
    counts = {"x": 0, "y": 0}
    firstYPos = getCountsAndFirstYPos(newPattern, counts)
    if counts["y"] != 0:
        for lenOfX in range(1, len(string)):
            lenOfY = (len(string) - lenOfX * counts["x"]) / counts["y"]
            if lenOfY <= 0 or lenOfY % 1 != 0:
                continue
            lenOfY = int(lenOfY)
            yIdx = firstYPos * lenOfX
            x = string[:lenOfX]
            y = string[yIdx : yIdx + lenOfY]
            potentialMatch = map(lambda char: x if char == "x" else y, newPattern)
            if string == "".join(potentialMatch):
                return [x, y] if not didSwitch else [y, x]
    else:
        lenOfX = len(string) / counts["x"]
        if lenOfX % 1 == 0:
            lenOfX = int(lenOfX)
            x = string[:lenOfX]
            potentialMatch = map(lambda char: x, newPattern)
            if string == "".join(potentialMatch):
                return [x, ""] if not didSwitch else ["", x]
    return []


def getNewPattern(pattern):
    patternLetters = list(pattern)
    if pattern[0] == "x":
        return patternLetters
    else:
        return list(map(lambda char: "x" if char == "y" else "y", patternLetters))


def getCountsAndFirstYPos(pattern, counts):
    firstYPos = None
    for i, char in enumerate(pattern):
        counts[char] += 1
        if char == "y" and firstYPos is None:
            firstYPos = i
    return firstYPos
```

Difficulty Medium: Smallest Substring Containing

You're given two non-empty strings: a big string and a small string. Write a function that returns the smallest substring in the big string that contains all of the small string's characters.

Note that:

The substring can contain other characters not found in the small string. The characters in the substring don't have to be in the same order as they appear in the small string. If the small string has duplicate characters, the substring has to contain those duplicate characters (it can also contain more, but not fewer). You can assume that there will only be one relevant smallest substring.



Sample Input

```
bigString = "abcd$ef$axb$c$"
smallString = "$$abf"
```

Sample Output

```
"f$axb$"
```

Hint:

Try storing all of the small string's character counts in a hash table where each character maps to the number of times that it appears in the small string. Try using two pointers (a left pointer and a right pointer) to traverse through the big string. How can this help you find the relevant smallest substring? With the two pointers mentioned in Hint, move the right pointer to the right in the big string, keeping track of all the characters you visit in a hash table identical to the one mentioned in Hint #1, until you've found all of the characters contained in the small string. At that point, move the left pointer to the right in the big string, keeping track of all the characters you "lose", and stop once you no longer have all of the small string's characters in between the left and

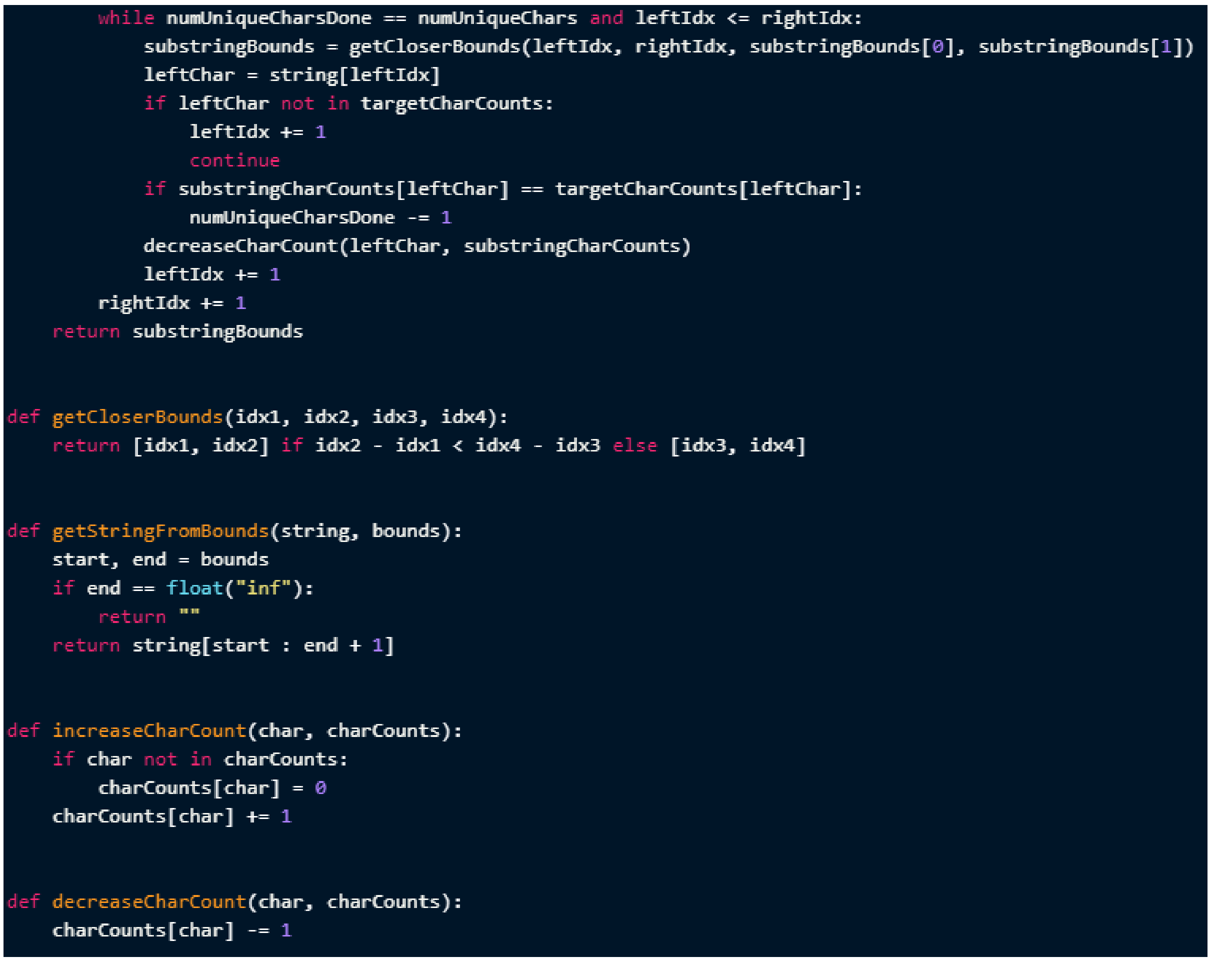
right pointers. Then, repeat the process by moving the right pointer forward and implementing the same logic described in this Hint.

Optimal Space & Time Complexity

O(b + s) time | O(b + s) space - where b is the length of the big input string and s is the length of the small input string

Solution:

```python
def smallestSubstringContaining(bigString, smallString):
    targetCharCounts = getCharCounts(smallString)
    substringBounds = getSubstringBounds(bigString, targetCharCounts)
    return getStringFromBounds(bigString, substringBounds)


def getCharCounts(string):
    charCounts = {}
    for char in string:
        increaseCharCount(char, charCounts)
    return charCounts


def getSubstringBounds(string, targetCharCounts):
    substringBounds = [0, float("inf")]
    substringCharCounts = {}
    numUniqueChars = len(targetCharCounts.keys())
    numUniqueCharsDone = 0
    leftIdx = 0
    rightIdx = 0
    # Move the rightIdx to the right in the string until you've counted
    # all of the target characters enough times.
    while rightIdx < len(string):
        rightChar = string[rightIdx]
        if rightChar not in targetCharCounts:
            rightIdx += 1
            continue
        increaseCharCount(rightChar, substringCharCounts)
        if substringCharCounts[rightChar] == targetCharCounts[rightChar]:
            numUniqueCharsDone += 1
```

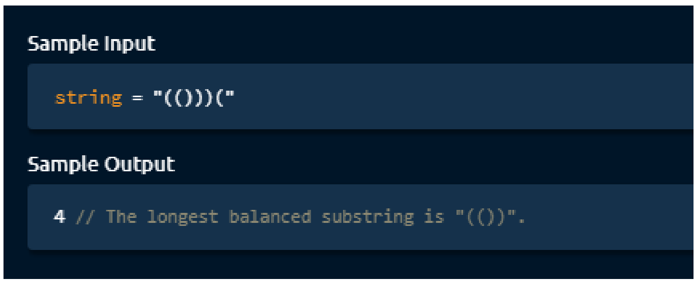


```python
        while numUniqueCharsDone == numUniqueChars and leftIdx <= rightIdx:
            substringBounds = getCloserBounds(leftIdx, rightIdx, substringBounds[0], substringBounds[1])
            leftChar = string[leftIdx]
            if leftChar not in targetCharCounts:
                leftIdx += 1
                continue
            if substringCharCounts[leftChar] == targetCharCounts[leftChar]:
                numUniqueCharsDone -= 1
            decreaseCharCount(leftChar, substringCharCounts)
            leftIdx += 1
        rightIdx += 1
    return substringBounds


def getCloserBounds(idx1, idx2, idx3, idx4):
    return [idx1, idx2] if idx2 - idx1 < idx4 - idx3 else [idx3, idx4]


def getStringFromBounds(string, bounds):
    start, end = bounds
    if end == float("inf"):
        return ""
    return string[start : end + 1]


def increaseCharCount(char, charCounts):
    if char not in charCounts:
        charCounts[char] = 0
    charCounts[char] += 1


def decreaseCharCount(char, charCounts):
    charCounts[char] -= 1
```

Note both images are combined one solution

Difficulty Medium: Longest Balanced Substring

Write a function that takes in a string made up of parentheses ((and)). The function should return an integer representing the length of the longest balanced substring with regards to parentheses. A string is said to be balanced if it has as many opening parentheses as it has closing parentheses and if no parenthesis is unmatched. Note that an opening parenthesis can't match a closing parenthesis that comes before it, and similarly, a closing parenthesis can't match an opening parenthesis that comes after it.


Sample Input

```
string = "(()))("
```

Sample Output

```
4 // The longest balanced substring is "(())".
```

Hint:

With a brute-force style approach, you can iterate through all substrings of the input string, check if they're balanced, and keep track of the longest balanced

one. This approach will require using an auxiliary method to check whether a substring is balanced.

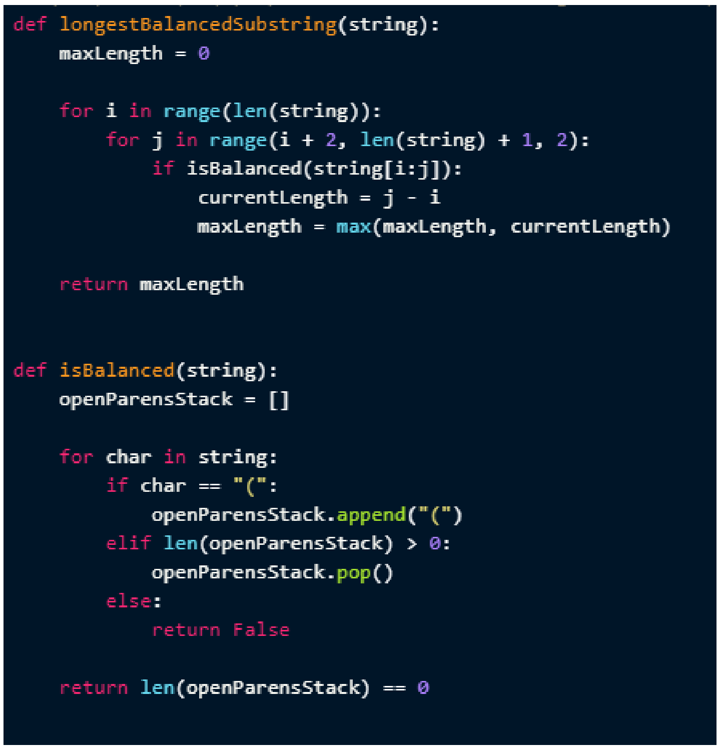
A more efficient approach to solving this problem is to iterate through the input string only once, using a stack to track the indices of all unmatched opening parentheses. Whenever a closing parenthesis is encountered, you check if the stack has a corresponding opening-parenthesis index, and you pop that index off the stack if it does. If the stack doesn't have a corresponding opening-parenthesis index, then the closing parenthesis is unmatched, and its own index in the input string denotes the start of a new, potentially balanced substring. With this approach, you'll have to figure out a way to keep track of how long a balanced substring is.

The most efficient way to solve this problem is to use only two variables to keep track of the numbers of opening and closing parentheses, respectively, as you traverse the string. Think about how you can use these two pieces of information alone to find the longest balanced substring. Specifically, how do these two pieces of information help you figure out if a substring is balanced, and how can you use them to calculate the length of such a substring?

Optimal Space & Time Complexity

O(n) time | O(1) space - where n is the length of the input string

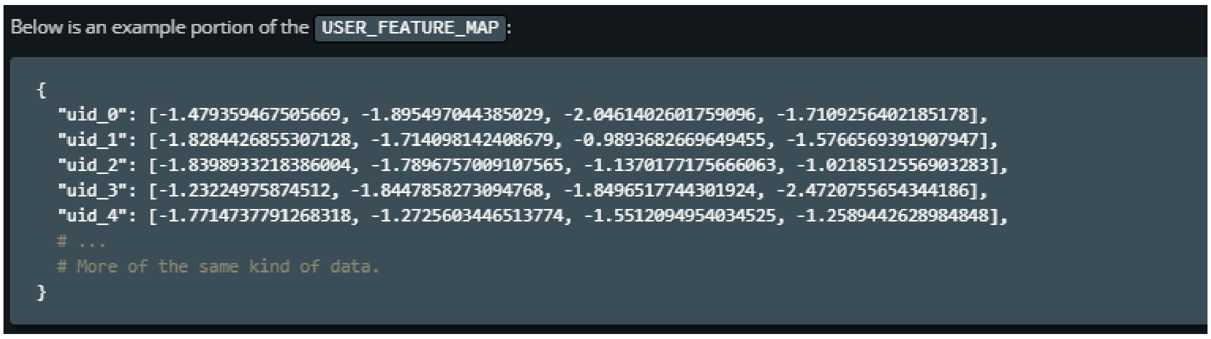
Solution:


```python
def longestBalancedSubstring(string):
    maxLength = 0

    for i in range(len(string)):
        for j in range(i + 2, len(string) + 1, 2):
            if isBalanced(string[i:j]):
                currentLength = j - i
                maxLength = max(maxLength, currentLength)

    return maxLength


def isBalanced(string):
    openParensStack = []

    for char in string:
        if char == "(":
            openParensStack.append("(")
        elif len(openParensStack) > 0:
            openParensStack.pop()
        else:
            return False

    return len(openParensStack) == 0
```

Difficulty V. Hard: K-means

Use the k-means algorithm to return the k means (or centroids) for the provided user features.

These user features are the result of a dimensionality reduction by PCA on some user-app interaction data. You'll have access to a USER_FEATURE_MAP dictionary, mapping each user "uid_i" to a respective list of 4 features associated with the user in question.



Below is an example portion of the `USER_FEATURE_MAP`:

```
{
  "uid_0": [-1.479359467505669, -1.895497044385029, -2.0461402601759096, -1.7109256402185178],
  "uid_1": [-1.8284426855307128, -1.714098142408679, -0.9893682669649455, -1.5766569391907947],
  "uid_2": [-1.8398933218386004, -1.7896757009107565, -1.1370177175666063, -1.0218512556903283],
  "uid_3": [-1.23224975874512, -1.8447858273094768, -1.8496517744301924, -2.4720755654344186],
  "uid_4": [-1.7714737791268318, -1.2725603446513774, -1.5512094954034525, -1.2589442628984848],
  # ...
  # More of the same kind of data.
}
```

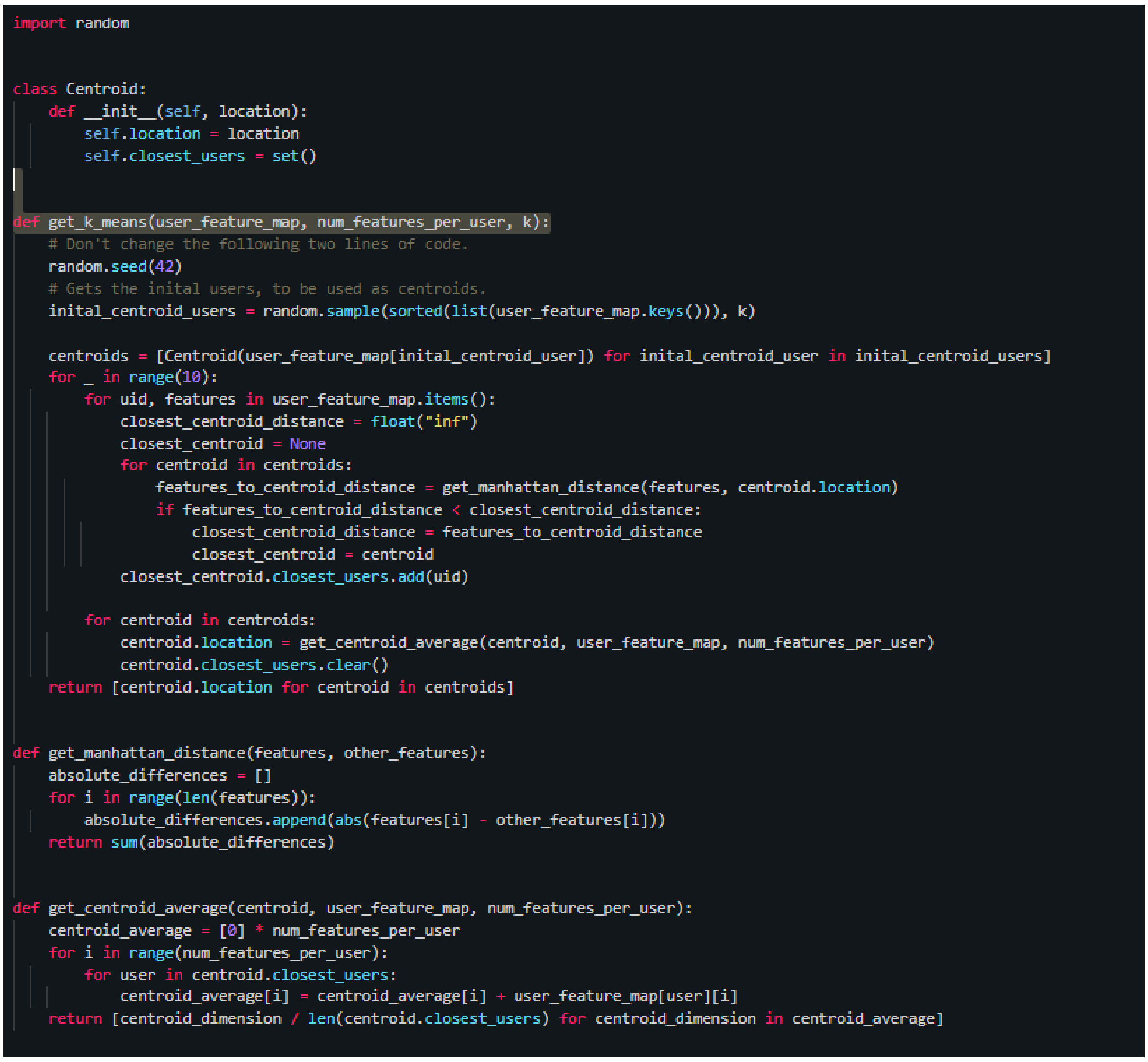
Note that:

The initial centroid locations are selected for you to ensure consistency when verifying your solution. You should execute at least 10 iterations of the k-means algorithm, not including the initialization of the centroids. You should use the Manhattan distance as the distance metric. You shouldn't use any libraries that implement k-means for you. Your output values will automatically be rounded to the fourth decimal.

Hint:

Start by creating k centroids based on the locations of the provided users. After creating the centroids, find the nearest centroid for each point, and assign each point to its nearest centroid. After finding each point's nearest centroid, update the location of each centroid to be the average of each point assigned to it. Calculating the average of n-dimensional data is the same as calculating the average of one-dimensional data; you just handle each dimension separately. Repeat the entire process of updating the centroids at least 10 times to allow for the centroids to converge on the means of the features.

Solution:



```python
import random


class Centroid:
    def __init__(self, location):
        self.location = location
        self.closest_users = set()


def get_k_means(user_feature_map, num_features_per_user, k):
    # Don't change the following two lines of code.
    random.seed(42)
    # Gets the inital users, to be used as centroids.
    inital_centroid_users = random.sample(sorted(list(user_feature_map.keys())), k)

    centroids = [Centroid(user_feature_map[inital_centroid_user]) for inital_centroid_user in inital_centroid_users]
    for _ in range(10):
        for uid, features in user_feature_map.items():
            closest_centroid_distance = float("inf")
            closest_centroid = None
            for centroid in centroids:
                features_to_centroid_distance = get_manhattan_distance(features, centroid.location)
                if features_to_centroid_distance < closest_centroid_distance:
                    closest_centroid_distance = features_to_centroid_distance
                    closest_centroid = centroid
            closest_centroid.closest_users.add(uid)

        for centroid in centroids:
            centroid.location = get_centroid_average(centroid, user_feature_map, num_features_per_user)
            centroid.closest_users.clear()
    return [centroid.location for centroid in centroids]


def get_manhattan_distance(features, other_features):
    absolute_differences = []
    for i in range(len(features)):
        absolute_differences.append(abs(features[i] - other_features[i]))
    return sum(absolute_differences)


def get_centroid_average(centroid, user_feature_map, num_features_per_user):
    centroid_average = [0] * num_features_per_user
    for i in range(num_features_per_user):
        for user in centroid.closest_users:
            centroid_average[i] = centroid_average[i] + user_feature_map[user][i]
    return [centroid_dimension / len(centroid.closest_users) for centroid_dimension in centroid_average]
```