Topic - Stack and Trie

Difficulty Hard : Suffix Trie Construction

Write a SuffixTrie class for a Suffix-Trie-like data structure. The class should have a root property set to be the root node of the trie and should support:

> Creating the trie from a string; this will be done by calling the populateSuffixTrieFrom method upon class instantiation, which should populate the root of the class.

> Searching for strings in the trie.

Note that every string added to the trie should end with the special endSymbol character: "*".

Sample Input (for creation)

```
string = "babc"
```

Sample Output (for creation)

```
The structure below is the root of the trie.
{
  "c": {"*": true},
  "b": {
    "c": {"*": true},
    "a": {"b": {"c": {"*": true}}},
  },
  "a": {"b": {"c": {"*": true}}},
}
```

Sample Input (for searching in the suffix trie above)

```
string = "abc"
```

Sample Output (for searching in the suffix trie above)

```
true
```

Hint:

Building a suffix-trie-like data structure consists of essentially storing every suffix of a given string in a trie. To do so, iterate through the input string one character at a time and insert every substring starting at each character and ending at the end of the string into the trie. To insert a string into the trie, start by adding the first character of the string into the root node of the trie and mapping it to an empty hash table if it isn't already there. Then, iterate through the rest of the string inserting each of the remaining characters into

the previous character's corresponding node (or hash table) in the trie, making sure to add an endSymbol "*" at the end. Searching the trie for a specific string should follow a nearly identical logic to the one used to add a string in the trie.

Optimal Space & Time Complexity:

Creation: O(n^2) time | O(n^2) space - where n is the length of the input string
Searching: O(m) time | O(1) space - where m is the length of the input string

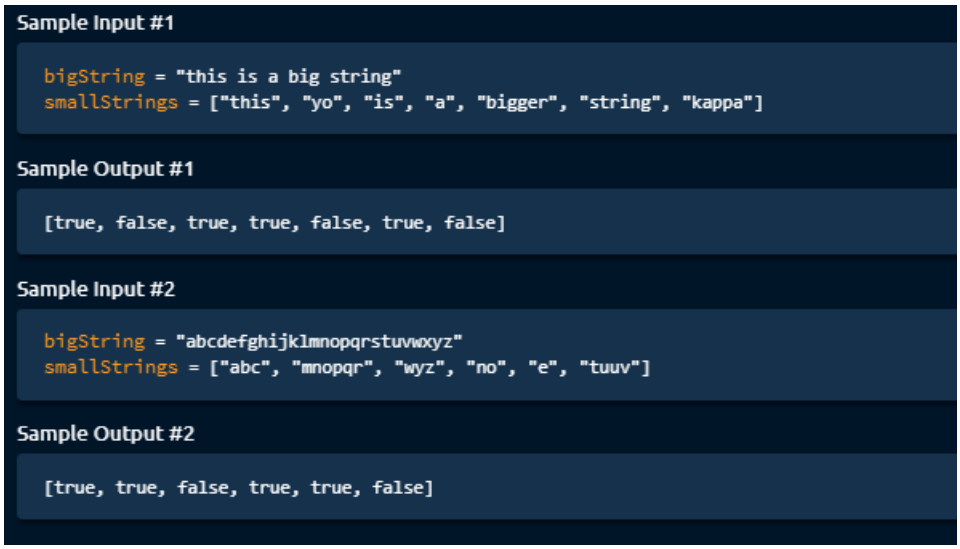
Solution:

```
class SuffixTrie:
    def __init__(self, string):
        self.root = {}
        self.endSymbol = "*"
        self.populateSuffixTrieFrom(string)

    # O(n^2) time | O(n^2) space
    def populateSuffixTrieFrom(self, string):
        for i in range(len(string)):
            self.insertSubstringStartingAt(i, string)

    def insertSubstringStartingAt(self, i, string):
        node = self.root
        for j in range(i, len(string)):
            letter = string[j]
            if letter not in node:
                node[letter] = {}
            node = node[letter]
        node[self.endSymbol] = True

    # O(m) time | O(1) space
    def contains(self, string):
        node = self.root
        for letter in string:
            if letter not in node:
                return False
            node = node[letter]
        return self.endSymbol in node
```

Difficulty Hard : Multi String Search

Write a function that takes in a big string and an array of small strings, all of which are smaller in length than the big string. The function should return an array of booleans, where each boolean represents whether the small string at that index in the array of small strings is contained in the big string.

Note that you can't use language-built-in string-matching methods.



Hint:

A simple way to solve this problem is to iterate through all of the small strings, checking if each of them is contained in the big string by iterating through the big string's characters and comparing them to the given small string's characters with a couple of loops. Is this approach efficient from a time-complexity point of view? Try building a suffix-trie-like data structure containing all of the big string's suffixes. Then, iterate through all of the small strings and check if each of them is contained in the data structure you've created. What are the time-complexity ramifications of this approach? Try building a trie containing all of the small strings. Then, iterate through the big string's characters and check if any part of the big string is a string contained in the trie you've created. Is this approach better than the one described in from a time-complexity point of view?

Optimal Space & Time Complexity:

O(ns + bs) time | O(ns) space - where n is the number of small strings, s is the length of longest small string, and b is the length of the big string

Solution:

```python
def multiStringSearch(bigString, smallStrings):
    return [isInBigString(bigString, smallString) for smallString in smallStrings]


def isInBigString(bigString, smallString):
    for i in range(len(bigString)):
        if i + len(smallString) > len(bigString):
            break
        if isInBigStringHelper(bigString, smallString, i):
            return True
    return False


def isInBigStringHelper(bigString, smallString, startIdx):
    leftBigIdx = startIdx
    rightBigIdx = startIdx + len(smallString) - 1
    leftSmallIdx = 0
    rightSmallIdx = len(smallString) - 1
    while leftBigIdx <= rightBigIdx:
        if bigString[leftBigIdx] != smallString[leftSmallIdx] or bigString[rightBigIdx] != smallString[rightSmallIdx]:
            return False
        leftBigIdx += 1
        rightBigIdx -= 1
        leftSmallIdx += 1
        rightSmallIdx -= 1
    return True
```

Difficulty Easy : Min Max Stack Construction

Write a MinMaxStack class for a Min Max Stack. The class should support: Pushing and popping values on and off the stack. Peeking at the value at the top of the stack. Getting both the minimum and the maximum values in the stack at any given point in time. All class methods, when considered independently, should run in constant time and with constant space.

Sample Usage

```
// All operations below are performed sequentially.
MinMaxStack(): - // instantiate a MinMaxStack
push(5): -
getMin(): 5
getMax(): 5
peek(): 5
push(7): -
getMin(): 5
getMax(): 7
peek(): 7
push(2): -
getMin(): 2
getMax(): 7
peek(): 2
pop(): 2
pop(): 7
getMin(): 5
getMax(): 5
peek(): 5
```

Hint:

You should be able to push values on, pop values off, and peek at values on top of the stack at any time and in constant time, using constant space. What data structure maintains order and would allow you to do this? You should be able to get the minimum and maximum values in the stack at any time and in constant time, using constant space. What data structure would allow you to do this? Since the minimum and maximum values in the stack can change with every push and pop, you will likely need to keep track of all the mins and maxes at every value in the stack.

Optimal Space & Time Complexity:

All methods: O(1) time | O(1) space

Solution:

```
class MinMaxStack:
    def __init__(self):
        self.minMaxStack = []
        self.stack = []

    # O(1) time | O(1) space
    def peek(self):
        return self.stack[len(self.stack) - 1]

    # O(1) time | O(1) space
    def pop(self):
        self.minMaxStack.pop()
        return self.stack.pop()

    # O(1) time | O(1) space
    def push(self, number):
        newMinMax = {"min": number, "max": number}
        if len(self.minMaxStack):
            lastMinMax = self.minMaxStack[len(self.minMaxStack) - 1]
            newMinMax["min"] = min(lastMinMax["min"], number)
            newMinMax["max"] = max(lastMinMax["max"], number)
        self.minMaxStack.append(newMinMax)
        self.stack.append(number)

    # O(1) time | O(1) space
    def getMin(self):
        return self.minMaxStack[len(self.minMaxStack) - 1]["min"]

    # O(1) time | O(1) space
    def getMax(self):
        return self.minMaxStack[len(self.minMaxStack) - 1]["max"]
```

Difficulty Easy : Balanced Brackets

Write a function that takes in a string made up of brackets ((, [, {,),], and }) and other optional characters. The function should return a boolean representing whether the string is balanced with regards to brackets. A string is said to be balanced if it has as many opening brackets of a certain type as it has closing brackets of that type and if no bracket is unmatched. Note that an opening bracket can't match a corresponding closing bracket that comes before it, and similarly, a closing bracket can't match a corresponding opening bracket that comes after it. Also, brackets can't overlap each other as in [(]).

Sample Input

```
string = "([])(){}(())()()"
```

Sample Output

```
true // it's balanced
```

Hint:

If you iterate through the input string one character at a time, there are two scenarios in which the string will be unbalanced: either you run into a closing bracket with no prior matching opening bracket or you get to the end of the string with some opening brackets that haven't been matched. Can you use an auxiliary data structure to keep track of all the brackets and efficiently check if you run into a unbalanced scenario at every iteration? Consider using a stack to store opening brackets as you traverse the string. The Last-In-First-Out property of the stack should allow you to match any closing brackets that you run into against the most recent opening bracket, if one exists, in which case you can simply pop it out of the stack. How can you check that there are no unmatched opening bracket once you've finished traversing through the string?

Optimal Space & Time Complexity:

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

Solution:

```python
def balancedBrackets(string):
    openingBrackets = "([{"
    closingBrackets = ")]}"
    matchingBrackets = {")": "(", "]": "[", "}": "{"}
    stack = []
    for char in string:
        if char in openingBrackets:
            stack.append(char)
        elif char in closingBrackets:
            if len(stack) == 0:
                return False
            if stack[-1] == matchingBrackets[char]:
                stack.pop()
            else:
                return False
    return len(stack) == 0
```

Difficulty Easy : Sunset Views

Given an array of buildings and a direction that all of the buildings face, return an array of the indices of the buildings that can see the sunset. A building can see the sunset if it's strictly taller than all of the buildings that come after it in the direction that it faces. The input array named buildings contains positive, non-zero integers representing the heights of the buildings. A building at index i thus has a height denoted by buildings[i]. All of the buildings face the same direction, and this direction is either east or west, denoted by the input string named direction, which will always be equal to either "EAST" or "WEST". In relation to the input array, you can interpret these directions as right for east and left for west. Important note: the indices in the ouput array should be sorted in ascending order.

Sample Input #1

```
buildings = [3, 5, 4, 4, 3, 1, 3, 2]
direction = "EAST"
```

Sample Output #1

```
[1, 3, 6, 7]

// Below is a visual representation of the sample input.
//    _
//   | |_ _
//  _| | | |_  _
// | | | | | | |_
// | | | | | |_| | |
// |_|_|_|_|_|_|_|_|
```

Sample Input #2

```
buildings = [3, 5, 4, 4, 3, 1, 3, 2]
direction = "WEST"
```

Sample Output #2

```
[0, 1]

// The buildings are the same as in the first sample
// input, but their direction is reversed.
```

Hint:

Is there a way to solve this problem in one loop? How does your solution change based on the direction that the buildings are facing? You can use the same approach for each direction by simply changing the direction in which you traverse the array of buildings. There are multiple ways to solve this problem, but one is to maintain a running maximum of building heights. Loop in the opposite direction that the buildings are facing, and keep track of the

maximum building height that you've seen. At each iteration, compare the height of the current building to the running maximum; if the current building is taller, then it can see the sunset; otherwise, it can't. Finally, at each iteration, update the running maximum. Another way to solve this problem is to use a stack. Loop in the direction that the buildings are facing, and add the index of the current building to the stack at the end of each iteration. Before adding elements to the stack, compare the current building height to buildings at the top of the stack. Pop off the top of the stack until the current building height is shorter than the height of the building at the top of the stack. This will remove all buildings that are blocked from seeing the sunset by the current building. At the end of the algorithm, the stack will only contain elements that can see the sunset.

Optimal Space & Time Complexity:

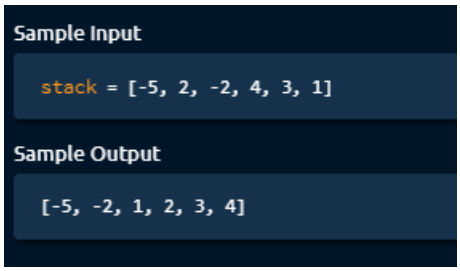
O(n) time | O(n) space - where n is the length of the input array

Solution:

```
def sunsetViews(buildings, direction):
    buildingsWithSunsetViews = []

    startIdx = 0 if direction == "WEST" else len(buildings) - 1
    step = 1 if direction == "WEST" else -1

    idx = startIdx
    runningMaxHeight = 0
    while idx >= 0 and idx < len(buildings):
        buildingHeight = buildings[idx]

        if buildingHeight > runningMaxHeight:
            buildingsWithSunsetViews.append(idx)

        runningMaxHeight = max(runningMaxHeight, buildingHeight)

        idx += step

    if direction == "EAST":
        return buildingsWithSunsetViews[::-1]

    return buildingsWithSunsetViews
```

Difficulty Medium : Sort Stack

Write a function that takes in an array of integers representing a stack, recursively sorts the stack in place (i.e., doesn't create a brand new array), and returns it. The array must be treated as a stack, with the end of the array as the top of the stack. Therefore, you're only allowed to Pop elements from the top of the stack by removing elements from the end of the array using the built-in .pop() method in your programming language of choice. Push elements to the top of the stack by appending elements to the end of the array using the built-in .append() method in your programming language of choice. Peek at the element on top of the stack by accessing the last element in the array. You're not allowed to perform any other operations on the input array, including accessing elements (except for the last element), moving elements, etc.. You're also not allowed to use any other data structures, and your solution must be recursive.



Hint:

If you had to insert a single item into an already sorted stack, all the while abiding by the constraints of this problem, how would you do that? Inserting a single item in an already sorted stack is fairly simple: you can pop elements off of the stack until you find an element that's smaller than or equal to the value that you want to add. Then, you can push that value on top of the stack and reinsert all the previously popped items back on top of the stack in the reverse order in which you popped them off. The resulting stack will still be sorted. You can easily insert multiple items in an already sorted stack by just repeatedly performing above hint. However, you'll need to have an already sorted stack. To get an already sorted stack, you'll need to pop all of the elements off the unsorted stack until it's eventually empty, and then you'll need to push all of the items back on the stack, inserting them in their sorted order one at a time. recursively, the steps are the following: Pop an item from the top of the stack, and hold onto it in memory. Sort the rest of the stack. To do so, repeat step #1 until the stack is empty, at which point you've reached

the base case since an empty stack is always sorted. Insert the most recently popped off item from step #1 back into the now sorted stack but in its proper sorted position. The first time that you reinsert an item, it'll be inserted in an empty stack.

Optimal Space & Time Complexity:

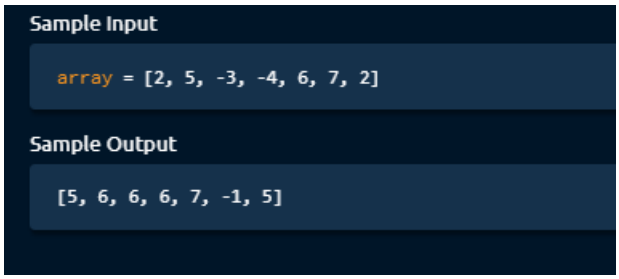
O(n^2) time | O(n) space - where n is the length of the stack

Solution:

```python
def sortStack(stack):
    if len(stack) == 0:
        return stack

    top = stack.pop()

    sortStack(stack)

    insertInSortedOrder(stack, top)

    return stack


def insertInSortedOrder(stack, value):
    if len(stack) == 0 or stack[len(stack) - 1] <= value:
        stack.append(value)
        return

    top = stack.pop()

    insertInSortedOrder(stack, value)

    stack.append(top)
```

Difficulty Med : Next Greater Element

Write a function that takes in an array of integers and returns a new array containing, at each index, the next element in the input array that's greater than the element at that index in the input array. In other words, your function should return a new array where outputArray[i] is the next element in the input array that's greater than inputArray[i]. If there's no such next greater element for a particular index, the value at that index in the output array should be -1. For example, given array = [1, 2], your function should return [2, -1]. Additionally, your function should treat the input array as a circular array. A circular array wraps around itself as if it were connected end-to-end. So the next index after the last index in a circular array is the first index. This means that, for our problem, given array = [0, 0, 5, 0, 0, 3, 0 0], the next greater element after 3 is 5, since the array is circular.



Hint:

Solving this problem in O(n^2) time, where n is the length of the array, is straightforward. Can you solve it with a better time complexity? How can a stack allow you to solve this problem in O(n) time? There are a couple of ways to solve this problem in linear time with a stack. One approach is to push onto the stack the indices of elements for which you haven't yet found the next greater element. If you go with this index approach, you need to loop through the array twice (since it's circular) and compare the value of the current element in the array to the one represented by the index on top of the stack. If the element on the top of the stack is smaller than the current element, then the current element is next greater element for the top-of-stack element, and you can pop the index off the top of the stack and use it to store the current element in the correct position in your result array. You then continue to pop elements off the top of the stack until the current element is no longer greater than the top-of-stack element. At this point, you add the index of the current element to the top of the stack, and you continue iterating through the array, repeating the same process. The approach discussed in Hint #3 assumes that

you loop through the array from left to right. You could loop through the array backwards using a very similar approach, storing the actual values of elements on the stack rather than their indices. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation.

Optimal Space & Time Complexity:

O(n) time | O(n) space - where n is the length of the array

Solution:

```python
def nextGreaterElement(array):
    result = [-1] * len(array)
    stack = []

    for idx in range(2 * len(array)):
        circularIdx = idx % len(array)

        while len(stack) > 0 and array[stack[len(stack) - 1]] < array[circularIdx]:
            top = stack.pop()
            result[top] = array[circularIdx]

        stack.append(circularIdx)

    return result
```

Difficulty Hard : Shorten Path

Write a function that takes in a non-empty string representing a valid Unix-shell path and returns a shortened version of that path. A path is a notation that represents the location of a file or directory in a file system. A path can be an absolute path, meaning that it starts at the root directory in a file system, or a relative path, meaning that it starts at the current directory in a file system. In a Unix-like operating system, a path is bound by the following rules: The root directory is represented by a /. This means that if a path starts with /, it's an absolute path; if it doesn't, it's a relative path. The symbol / otherwise represents the directory separator. This means that the path /foo/bar is the location of the directory bar inside the directory foo, which is itself located inside the root directory. The symbol .. represents the parent directory. This means that accessing files or directories in /foo/bar/.. is equivalent to accessing files or directories in /foo. The symbol . represents the current directory. This means that accessing files or directories in /foo/bar/. is equivalent to accessing files or directories in /foo/bar. The symbols / and . can be repeated sequentially without consequence; the symbol .. cannot, however, because repeating it sequentially means going further up in parent directories. For example, /foo/bar/baz/././. and /foo/bar/baz are equivalent paths, but /foo/bar/baz/../../../ and /foo/bar/baz definitely aren't. The only exception is with the root directory: /../../.. and / are equivalent, because the root directory has no parent directory, which means that repeatedly accessing parent directories does nothing. Note that the shortened version of the path must be equivalent to the original path. In other words, it must point to the same file or directory as the original path.

Sample Input

```
path = "/foo/../test/../test/../foo//bar/./baz"
```

Sample Output

```
"/foo/bar/baz" // This path is equivalent to the input path.
```

Hint:

A path effectively consists of meaningful "tokens" (like directory names and symbols) that have been put together. Try transforming the string version of the path into a list of meaningful tokens that you can then analyze as you see fit. Split the input path around the directory separator "/" using a native "split" function and try eliminating meaningless tokens from the resulting list of tokens. Meaningless tokens will include the empty string and the "." symbol,

since the emptry string will represent sequential "/"s, which are effectively useless, and the "." symbol is also effectively useless. The ".." symbol essentially requires you to remove the previous token in the list of tokens; try using a stack to implement the logic of parsing out ".." symbols and the relevant parent directories. You'll need to handle two edge cases: the case where the path is an absolute one (for this, you'll have to identify if the path starts with a "/" at the beginning of your algorithm and then tweak other logic accordingly) and the case where the path is a relative one that starts with one or multiple ".." symbols (in this case, you'll want to keep these symbols, since they're meaningful to the path).

Optimal Space & Time Complexity:

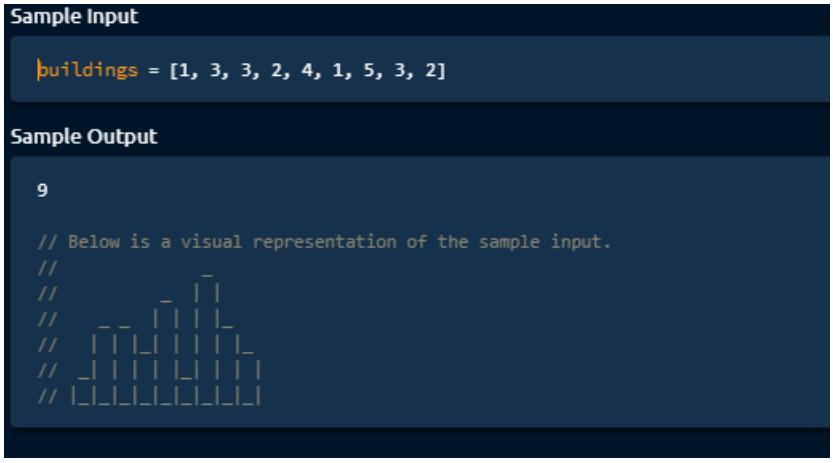
O(n) time | O(n) space - where n is the length of the pathname

Solution:

```
def shortenPath(path):
    startsWithSlash = path[0] == "/"
    tokens = filter(isImportantToken, path.split("/"))
    stack = []
    if startsWithSlash:
        stack.append("")
    for token in tokens:
        if token == "..":
            if len(stack) == 0 or stack[-1] == "..":
                stack.append(token)
            elif stack[-1] != "":
                stack.pop()
        else:
            stack.append(token)

    if len(stack) == 1 and stack[0] == "":
        return "/"
    return "/".join(stack)


def isImportantToken(token):
    return len(token) > 0 and token != "."
```

Difficulty Hard : Largest Rectangle Under Skyline

Write a function that takes in an array of positive integers representing the heights of adjacent buildings and returns the area of the largest rectangle that can be created by any number of adjacent buildings, including just one building. Note that all buildings have the same width of 1 unit. For example, given buildings = [2, 1, 2], the area of the largest rectangle that can be created is 3, using all three buildings. Since the minimum height of the three buildings is 1, you can create a rectangle that has a height of 1 and a width of 3 (the number of buildings). You could also create rectangles of area 2 by using only the first building or the last building, but these clearly wouldn't be the largest rectangles. Similarly, you could create rectangles of area 2 by using the first and second building or the second and third building. To clarify, the width of a created rectangle is the number of buildings used to create the rectangle, and its height is the height of the smallest building used to create it. Note that if no rectangles can be created, your function should return 0.



Hint:

Try treating every building as a pillar of a rectangle that can be created with the height of the building in question. The brute-force approach to solve this problem involves treating every building as a part of a potential rectangle to be created. As you loop through all the buildings, simply expand to the left and right of the current building, and determine the width of the longest rectangle that you can create that has a height of the current building. Calculate the area of this longest rectangle, and update a variable to store the area of the largest rectangle that you've found so far. This approach has a time complexity of O(n^2); can you do better? There's a way to solve this problem in linear (O(n)) time by using a stack. When should you push and pop buildings on and off the

stack if you were to loop through the buildings from left to right? Try to think of each building on the stack as a pillar of a potential rectangle. The stack mentioned in Hint #3 will be used to determine the length of a rectangle that has the height of a building that is currently on top of the stack. Loop through all the buildings, and at each building, compare its height to the height of the building on top of the stack. If the current building's height is smaller than or the same as the height of the building on top of the stack, pop the building off the stack. When you pop the building off the stack, you've determined the rightmost position (your current position) of a rectangle of that height (the height of the building you popped) that uses that building. Then, to determine the leftmost position of that rectangle, you look at the next building on top of the stack. This is the index of the closest building to the left that has a smaller height than that of the building that was just popped off. Now, you can calculate the area of the rectangle that uses this building and update a variable to store the max area. Continue popping buildings off the stack at each iteration until the current building is taller than the one on top of the stack, and don't forget to push each building on top of the stack at each iteration. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation.

Optimal Space & Time Complexity:

O(n) time | O(n) space - where n is the number of buildings

Solution:

```
def largestRectangleUnderSkyline(buildings):
    maxArea = 0
    for pillarIdx in range(len(buildings)):
        currentHeight = buildings[pillarIdx]

        furthestLeft = pillarIdx
        while furthestLeft > 0 and buildings[furthestLeft - 1] >= currentHeight:
            furthestLeft -= 1

        furthestRight = pillarIdx
        while furthestRight < len(buildings) - 1 and buildings[furthestRight + 1] >= currentHeight:
            furthestRight += 1

        areaWithCurrentBuilding = (furthestRight - furthestLeft + 1) * currentHeight
        maxArea = max(areaWithCurrentBuilding, maxArea)

    return maxArea
```