LEET CODE CHALLENGES

Autor: *Leonardo Costa Passos*

Table of Contents

1. STRINGs

1.1. validPalindrome (LeetCode: 125)

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Example 1:

- Input: s = "A man, a plan, a canal: Panama"
- Output: true
- Explanation: "amanaplanacanalpanama" is a palindrome.

Example 2:

- Input: s = "race a car"
- Output: false
- Explanation: "raceacar" is not a palindrome.

Example 3:

- Input: s = " "
- Output: true
- Explanation: s is an empty string "" after removing non-alphanumeric characters.

Since an empty string reads the same forward and backward, it is a palindrome.

1.1.1. JavaScript

BootCamp JS > LC125_isPalindrome > JS index.js > ...

```javascript
function isPalindrome(s) {
    s = s.toLowerCase().replace(/[\W_]/g, "");
    console.log(s);
    let left = 0;
    let right = s.length -1;
    while (left < right){
        if (s[left] != s[right]){
            return false;
        }
        left ++;
        right --;
    }
    return true;

}
```

1.1.2. Python

BootCamp PY > valid_Palindrome.py > ...

```python
import re

def is_palindrome(s):
    s = s.lower()
    s = re.sub(r'[\W_]', '', s)
    print(s)
    left = 0
    right = len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

```

1.1.3. Logic Explained:

- Import Regular Expression Module: The re module is imported to provide support for regular expression operations.
- Define Function: A function called is_palindrome is defined, which takes one argument, a string s.
- Convert to Lowercase: Inside the function, the string s is converted to all lowercase using the lower() method. This ensures the comparison is case-insensitive.
- Remove Non-Alphanumeric Characters: Using the regular expression r'[\W_]', all non-alphanumeric characters, including underscores, are replaced with an empty string ''. This leaves only alphanumeric characters for comparison.
- \W matches any character that is not a letter, digit, or underscore. _ matches the underscore character specifically.
- re.sub(pattern, replacement, string) replaces occurrences of the pattern in the string with the replacement.
- Print Cleaned String: The cleaned and filtered string s is printed.
- Initialize Pointers: Two pointers, left and right, are initialized at the beginning and the end of the string, respectively.
- Check for Palindrome: A while loop is used to compare characters from the beginning and end of the string, moving towards the center.
- If the characters pointed to by left and right are not the same, the function returns False.
- If they are the same, left is incremented and right is decremented, and the loop continues.
- Final Return Value: If the loop completes without finding any mismatched characters, the function returns True, indicating that the string is a palindrome.
- Test Example: The function is called with the example string "A man, a plan, a canal: Panama", and the result is printed.

The overall effect of this code is to determine if the input string s is a palindrome when considering only alphanumeric characters and ignoring case. In the given example, the function will return True, as the cleaned version of the string "amanaplanacanalpanama" is a palindrome.

1.2. isAnagram (LeetCode: 242)

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

- Input: s = "anagram", t = "nagaram"
- Output: true

Example 2:

- Input: s = "rat", t = "car"
- Output: false

1.2.1. JavaScript

BootCamp JS > LC242_isAnagram > JS index.js > ...

```javascript
function isAnagram(s, t) {
    if (s.length != t.length){
        return false;
    }
    const sCharCounts = {};

    for (let i = 0; i < s.length; i++){
        const sChar = s[i];
        sCharCounts[sChar] = sCharCounts[sChar] + 1 || 1;

    }
    for (let i = 0; i < t.length; i++){
        const tChar = t[i];

        if (!sCharCounts[tChar]){
            return false
        } else{
            sCharCounts[tChar] --
        }
    }
return true
}
```

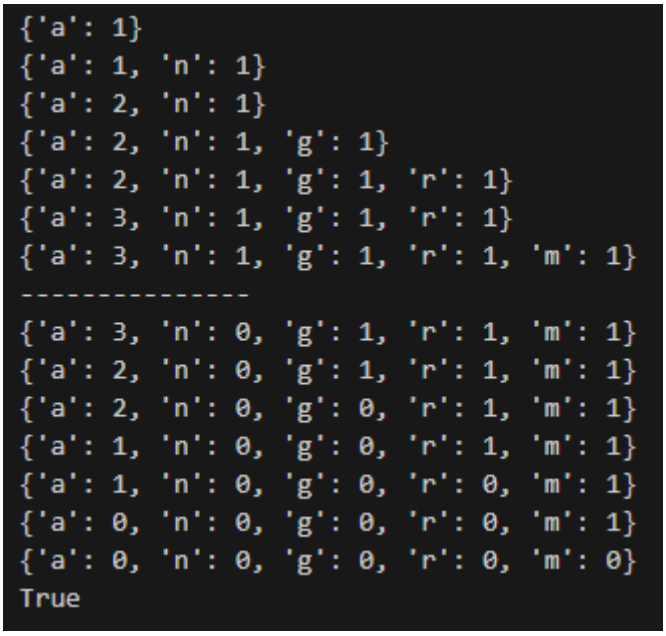
1.2.2. Python

BootCamp PY > isAnagram.py > ...

```python
def isAnagram(s, t):
    if len(s) != len(t):
        return False
    sCharCounts = {}

    for sChar in s:
        sCharCounts[sChar] = sCharCounts.get(sChar, 0) + 1

    for tChar in t:
        if tChar not in sCharCounts or sCharCounts[tChar] == 0:
            return False
        else:
            sCharCounts[tChar] -= 1

    return True
```

1.2.3. Logic Explained:

- If the lengths of s and t are different, it returns False;
- It then counts the occurrences of each character in string s using a dictionary sCharCounts;
- It then iterates through string t, decrementing the count of each character in sCharCounts.
    - If a character in t is not found in sCharCounts or its count is zero, the function returns False;
- If all counts are valid, it returns True.



1.3. validParentheses (LeetCode: 20)

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets;
- Open brackets must be closed in the correct order;
- Every close bracket has a corresponding open bracket of the same type.

Example 1:

- Input: s = "()"
- Output: true

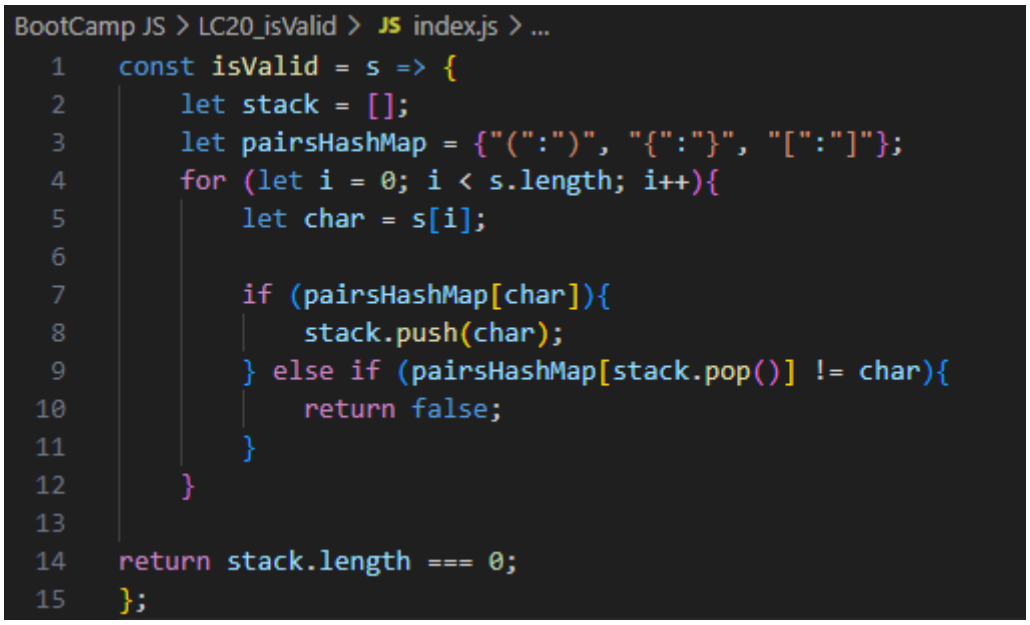
Example 2:

- Input: s = "()[]{}"
- Output: true

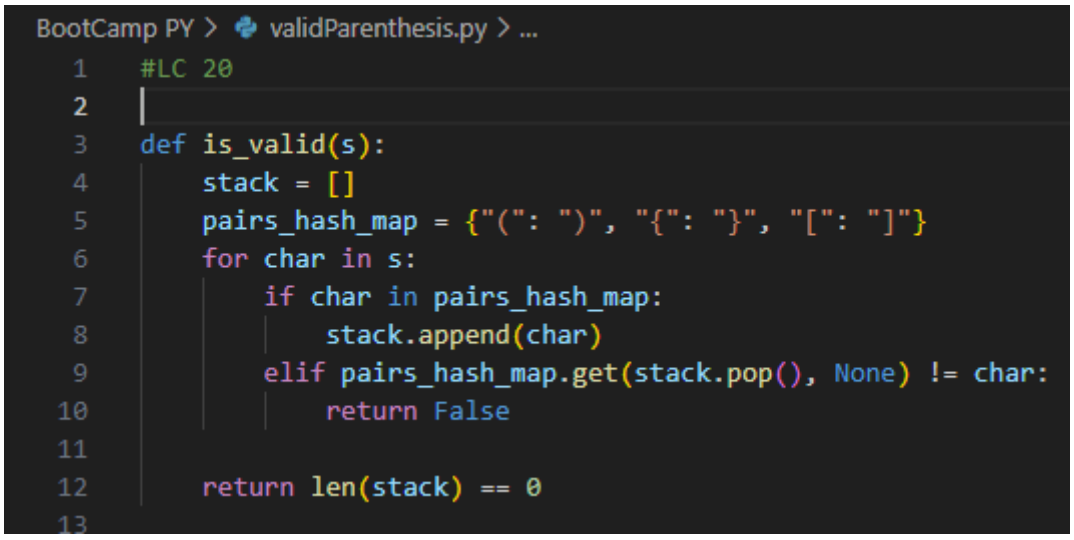
Example 3:

- Input: s = "(]"
- Output: false

1.3.1. JavaScript



1.3.2. Python



1.3.3. Logic Explained

Overall, this code determines whether the input string contains a valid sequence of balanced and nested parentheses, braces, and brackets.

- Function Definition: The function is_valid is defined, which takes a string s as an argument. The purpose of this function is to determine if the input string contains properly balanced and nested parentheses, braces, and brackets.
- Initialize Stack: An empty list stack is initialized, which will be used to track the opening parentheses, braces, or brackets as they are encountered.
- Define Pairs Mapping: The dictionary pairs_hash_map is defined, mapping the opening symbols "(", "{", and "[" to their corresponding closing symbols ")", "}", and "]", respectively.
- Iterate Through Characters: A for-loop is used to iterate through each character in the input string s.
- Check for Opening Symbols: If the current character char is an opening symbol, it is pushed onto the stack. This is done by checking if char is a key in pairs_hash_map.
- Check for Closing Symbols: If the current character char is not an opening symbol, it must be a closing symbol. In this case:
- stack.pop() is called to remove the last opening symbol from the stack.
- The corresponding closing symbol is retrieved from pairs_hash_map.
- If the retrieved closing symbol does not match the current character char, the function returns False.
- The get method is used to handle the case where stack.pop() returns None (when the stack is empty), in which case the comparison still results in False.
- Final Return Value: After iterating through all the characters, the function returns True if the stack is empty (i.e., len(stack) == 0) and False otherwise.
    - An empty stack means that every opening symbol was properly matched with a closing symbol, so the input string is considered valid;
    - A non-empty stack means that there were unmatched opening symbols, so the input string is considered invalid.

2. Dynamic Programming

2.1. House Robbers (LeetCode: 198):

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1:

- Input: nums = [1,2,3,1];
- Output: 4;
- Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3);
- Total amount you can rob = 1 + 3 = 4.

Example 2:

- Input: nums = [2,7,9,3,1];
- Output: 12;
- Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1);
- Total amount you can rob = 2 + 9 + 1 = 12.

Approach for solution:

The House Robber problem is typically solved using Dynamic Programming (DP), not a Greedy Algorithm. Here's a breakdown of why:

- Dynamic Programming Solution: This problem naturally lends itself to a DP approach because the optimal solution at each step depends on the previous steps. You want to keep track of the maximum amount you can rob up until each house, considering whether to rob the current house (and add its value to the amount from two houses ago) or skip it (and keep the amount from the previous house). The time complexity of this approach is O(n), where n is the number of houses.

- Greedy Algorithm Solution: A Greedy Algorithm typically works by making the locally optimal choice at each step in the hopes of finding a global optimum. In the case of the House Robber problem, making locally optimal choices (like always robbing the house with the most money available) doesn't necessarily lead to the global optimum. This is because the constraint of not robbing adjacent houses means you can't simply choose the highest value at each step without considering the overall impact on your strategy.

The array [3, 50, 100, 2] shows how a Greedy Algorithm would fail to find the optimal solution. By choosing the locally optimal choice (the second house with 50), you would miss the opportunity to rob the third house (100), whereas a Dynamic Programming approach would recognize that robbing the first and third houses (3 + 100) would lead to the highest total.

2.1.1. JavaScript:

BootCamp JS > LC198_rob > JS index.js > ...

```javascript
function rob(nums) {
    if (nums.length === 0) return 0;
    if (nums.length === 1) return num[0];
    if (nums.length === 2) return Math.max(nums[0], nums[1]);

    let masLootAtNth = [nums[0], Math.max(nums[0], nums[1])];

    for (let i =2; i < nums.length; i++){
        masLootAtNth.push (Math.max(nums[i] + masLootAtNth[i-2], masLootAtNth[i-1]));
    }

    return masLootAtNth.pop();
}
```

2.1.2. Python

BootCamp PY > rob.py > ...

```python
#LC: 198
def rob(nums):
    if len(nums) == 0:
        return 0
    if len(nums) == 1:
        return nums[0]
    if len(nums) == 2:
        return max(nums[0], nums[1])

    max_loot_at_nth = [nums[0], max(nums[0], nums[1])]

    for i in range(2, len(nums)):
        max_loot_at_nth.append(max(nums[i] + max_loot_at_nth[i - 2], max_loot_at_nth[i - 1]))

    return max_loot_at_nth.pop()
```

2.1.3. Logic Explained:

By applying dynamic programming, the function efficiently computes the maximum loot a robber can obtain without violating the constraint of not robbing adjacent houses.

Base Cases:

- If there are no houses (len(nums) == 0), the result is 0;
- If there is only one house (len(nums) == 1), the result is the value of that house;
- If there are two houses (len(nums) == 2), the result is the maximum of the two values;
- Initialization of Dynamic Programming Array: The array max_loot_at_nth is initialized with two values:
    - The value of the first house (nums[0]).
    - The maximum value of the first two houses (max(nums[0], nums[1])).

Dynamic Programming Iteration:

- Iterate through the houses starting from the third house (index 2);
- For each house, calculate the maximum loot by considering two scenarios:
    - Robbing the current house and the house two steps behind (nums[i] + max_loot_at_nth[i - 2]);
    - Not robbing the current house and considering the loot up to the previous house (max_loot_at_nth[i - 1]);
- Append the maximum of these two scenarios to the max_loot_at_nth array;
- Final Result: The last value in the max_loot_at_nth array represents the maximum loot considering all the houses, so the result is max_loot_at_nth.pop().

This approach avoids redundant computations and leads to a time complexity of O(n), where n is the number of houses.

2.2. Jump Game (LeetCode: 55)

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return true if you can reach the last index, or false otherwise.

Example 1:

- Input: nums = [2,3,1,1,4]
- Output: true
- Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

- Input: nums = [3,2,1,0,4]
- Output: false
- Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

2.2.1. JavaScript (Dynamic Programming):

Dynamic Programming Solution: Utilizes a table to keep track of reachability for each position and builds up the solution iteratively.

BootCamp JS > LC55_canJump > JS index.js > ...

```
function canJump(nums) {
    let dpPositions = new Array(nums.length).fill(false);
    dpPositions[0] = true;

    for (let j = 1; j < nums.length; j++){
        for (let i = 0; i < j; i++){
            if (dpPositions[i] && i + nums[i] >= j){
                dpPositions[j] = true;
                break
            }
        }
    }

    return dpPositions[dpPositions.length - 1]

}
```

2.2.2. Python (Dynamic Programming):

BootCamp PY > jumpGame.py > ...

```python
#LC: 55
def can_jump(nums):
    dp_positions = [False] * len(nums)
    dp_positions[0] = True

    for j in range(1, len(nums)):
        for i in range(j):
            if dp_positions[i] and i + nums[i] >= j:
                dp_positions[j] = True
                break

    return dp_positions[-1]

```

2.2.3. Logic Explained (Dynamic Programming Solution):

- Initialization: An array dp_positions is created with the same length as nums, initialized with False. The first element is set to True, as the start position is always reachable;
- Outer Loop: Iterates through the nums from index 1 onward. This represents the target position we're trying to reach;
- Inner Loop: Iterates from the beginning of the array up to the current target position (j). This represents the possible starting positions;
- Checking Reachability: For each starting position i, the code checks if it is reachable (dp_positions[i] is True) and if a jump from that position can reach or surpass the target position (i + nums[i] >= j). If so, the target position j is marked as reachable (True), and the inner loop breaks;
- Final Result: The function returns whether the last position in the array is reachable (dp_positions[-1]);

An optimized approach using a greedy algorithm could solve this problem in linear time (O(n)), but the given code prioritizes clarity and follows the pattern of dynamic programming. Therefore, while it's an instructive example for learning purposes, it may not be the best choice for handling large input sizes in a production environment.

2.2.4. JavaScript (Greedy Algo):

JS > JS jumpGame2.js > ...

```javascript
function canJump(nums) {
    let maxReach = 0;
    for (let i = 0; i < nums.length; i++) {
        if (i > maxReach) {
            return false;
        }
        maxReach = Math.max(maxReach, i + nums[i]);
    }
    return true;
}
```

2.2.5. Python (Greedy Algo):

BootCamp PY > jumpGame2.py > ...

```python
def can_jump(nums):
    max_reach = 0
    for i, jump in enumerate(nums):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + jump)
    return True
```

2.2.6. Logic Explained (Greedy Algo):

- Initialize max_reach, the furthest index that can be reached, to 0;
- Iterate through the nums array, where i is the current index, and jump (or nums[i] in JS) is the maximum jump length at that position;
- If i exceeds max_reach, return False because the current position is unreachable;
- Otherwise, update max_reach to the maximum of its current value and i + jump;
- Return True if the loop completes, as this means the last index is reachable.

This solution leverages a greedy approach by always attempting to maximize the reachable index at each step. Its time complexity is O(n), and its space complexity is O(1), making it more efficient than the dynamic programming approach.

2.3. Longest Increasing Subsequence (LeetCode: 300)

Given an integer array nums, return the length of the longest strictly increasing Subsequence.

Example 1:

- Input: nums = [10,9,2,5,3,7,101,18]
- Output: 4
- Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:

- Input: nums = [0,1,0,3,2,3]
- Output: 4
- Explanation: The longest increasing subsequence is [0,1,2,3], therefore the length is 4.

Example 3:

- Input: nums = [7,7,7,7,7,7,7]
- Output: 1

2.3.1. JavaScript:

BootCamp JS > LC300_lengthOfLIS > JS index.js > ...

```javascript
function lengthOfLIS(nums) {
    if (nums.length === 0) return 0;

    let dpSubsequence = new Array(nums.length).fill(1);
    let maxSoFar = 1;

    for (let j = 1; j < nums.length; j++){
        for (let i = 0; i < j; i++){
            if (nums[j] > nums[i]){
                dpSubsequence[j] = Math.max(dpSubsequence[i] + 1, dpSubsequence[j]);
            }
        }
        maxSoFar = Math.max(maxSoFar, dpSubsequence[j]);
    }
    return maxSoFar;
}
```

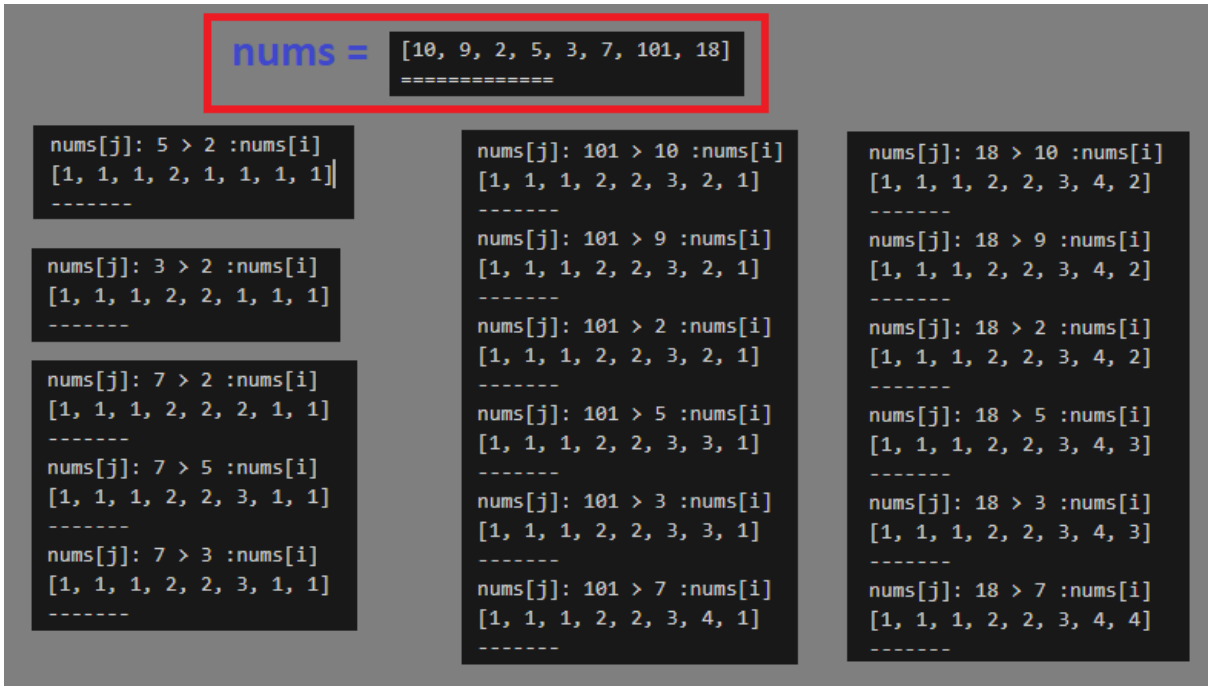
2.3.2. Python:

BootCamp PY > LIS.py > ...

```python
#LC:300

def lengthOfLIS(nums):
    if len(nums) == 0:
        return 0

    dpSubsequence = [1] * len(nums)
    maxSoFar = 1

    for j in range(1, len(nums)):
        for i in range(0, j):
            if nums[j] > nums[i]:
                dpSubsequence[j] = max(dpSubsequence[i] + 1, dpSubsequence[j])
        maxSoFar = max(maxSoFar, dpSubsequence[j])

    return maxSoFar

```

2.3.3. Logic Explained:

- Initialization: If the input array is empty, return 0, as no subsequence exists. Then, create a dpSubsequence array filled with 1, which represents the initial length of the LIS ending at each index. Initialize maxSoFar to 1 as well, as the minimum length will be at least 1.
- Nested Loop: Iterate through each element j of the input array, starting from the second element, and compare it to all previous elements i. If the current element j is greater than the element i, then it can be part of an increasing subsequence ending at j.
- Update: Update dpSubsequence[j] with the maximum value between the current value of dpSubsequence[j] and dpSubsequence[i] + 1, reflecting the fact that a new longer subsequence may have been found.
- Find Maximum Length: Keep track of the maximum length found so far by comparing maxSoFar with dpSubsequence[j].
- Return Result: Finally, return the value of maxSoFar, which represents the length of the longest increasing subsequence.



index	0	1	2	3	4	5	6	7
value	10	9	2	5	3	7	101	18

1. **Initialization**: `dpSubsequence = [1, 1, 1, 1, 1, 1, 1, 1]`, `maxSoFar = 1`
2. **First Iteration (j = 1)**: Since `nums[1]` is less than `nums[0]`, there is no update.
3. **Second Iteration (j = 2)**: `nums[2]` is less than `nums[0]` and `nums[1]`, so there is no update.
4. **Third Iteration (j = 3)**: `nums[3] > nums[2]`, so `dpSubsequence[3] = max(2, 1) = 2`.
5. **Fourth Iteration (j = 4)**: `nums[4] > nums[2]`, so `dpSubsequence[4] = max(2, 1) = 2`.
6. **Fifth Iteration (j = 5)**: `nums[5] > nums[3]`, so `dpSubsequence[5] = max(3, 1) = 3`.
7. **Sixth Iteration (j = 6)**: `nums[6] > nums[5]`, so `dpSubsequence[6] = max(4, 1) = 4`.
8. **Seventh Iteration (j = 7)**: `nums[7] > nums[5]`, so `dpSubsequence[7] = max(3, 1) = 3`.

2.4. Coin Change (LeetCode: 322)

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

- Input: coins = [1,2,5], amount = 11
- Output: 3
- Explanation: 11 = 5 + 5 + 1

Example 2:

- Input: coins = [2], amount = 3
- Output: -1

Example 3:

- Input: coins = [1], amount = 0
- Output: 0

2.4.1. JavaScript:

BootCamp JS > LC322_coinChange > JS index.js > ...

```javascript
function coinChange(coins, amount) {
    let dpMinCoins = new Array(amount + 1).fill(Infinity);
    dpMinCoins[0] = 0;

    for (let i = 1; i < dpMinCoins.length; i++) {
        for (let j = 0; j < coins.length; j++) {
            const coinValue = coins[j];
            if (coinValue <= i) {
                dpMinCoins[i] = Math.min(dpMinCoins[i - coinValue] + 1, dpMinCoins[i]);
            }
        }
    }

    const answer = dpMinCoins[dpMinCoins.length - 1];
    return answer === Infinity ? -1 : answer;
}
```

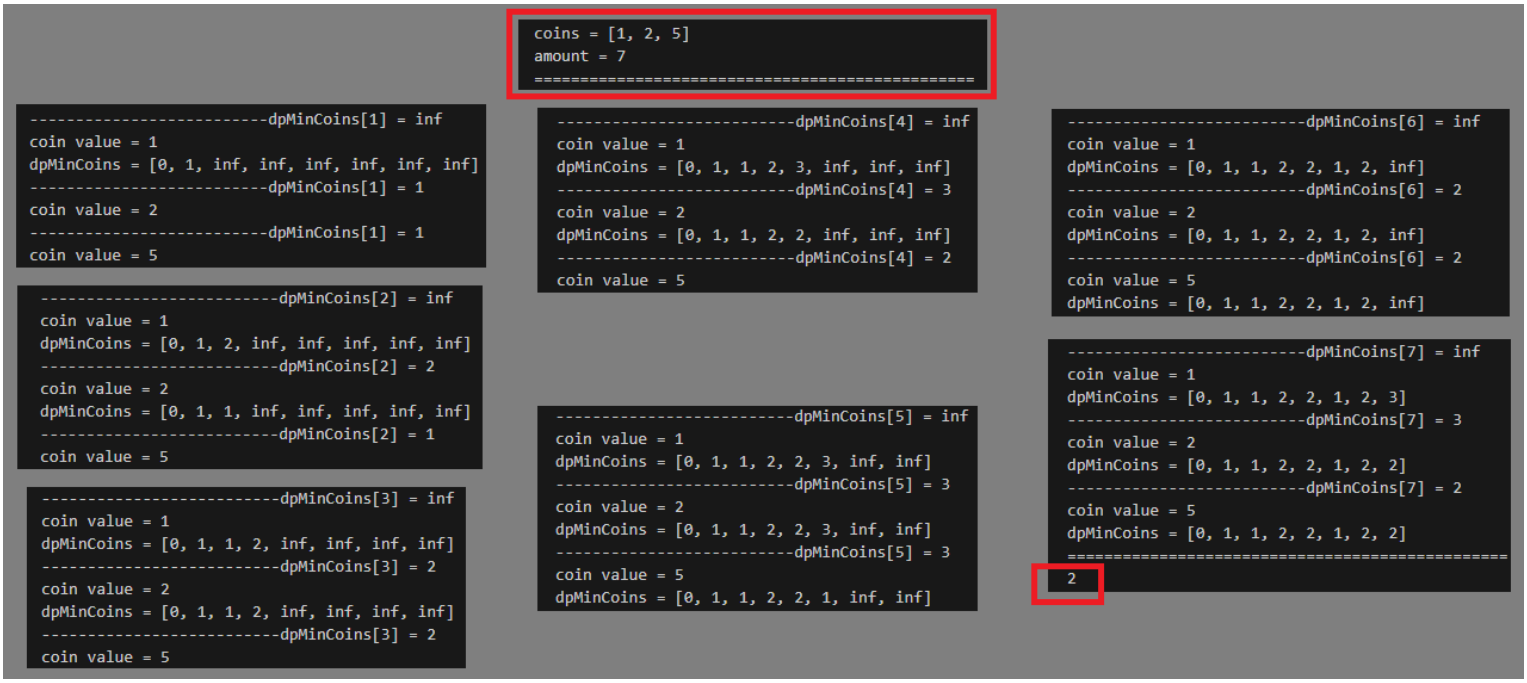
2.4.2. Python:

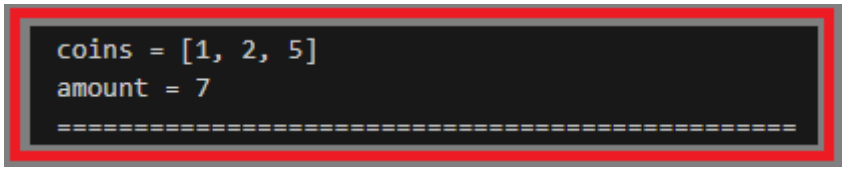
BootCamp PY > coinChange.py > ...

```python
#LC: 322
def coinChange(coins, amount):
    dpMinCoins = [float('inf')] * (amount + 1)
    dpMinCoins[0] = 0

    for i in range(1, len(dpMinCoins)):
        for coinValue in coins:
            if coinValue <= i:
                dpMinCoins[i] = min(dpMinCoins[i - coinValue] + 1, dpMinCoins[i])

    answer = dpMinCoins[-1]
    return -1 if answer == float('inf') else answer

```

2.4.3. Logic Explained:

- Initialization: An array dpMinCoins is initialized with size amount + 1 and filled with infinity, except for the first element dpMinCoins[0], which is set to 0, representing that 0 coins are needed for amount 0.
- Building the DP array: For each amount from 1 to the target amount, the code iterates through all coin denominations, updating the minimum number of coins needed for the current amount.
    - If the coin value is less than or equal to the current amount, the corresponding DP value is updated with the minimum of its current value and the value for the remaining amount plus one.
- Final result: The final element in the dpMinCoins array represents the minimum number of coins needed for the target amount. If it's still infinity, then the amount is not reachable, and the function returns -1. Otherwise, it returns the value.





For the given input coins = [1, 2, 5] and amount = 7, the function coinChange will perform the following steps:

- Initialization: Initialize a dynamic programming (DP) table dpMinCoins of length amount + 1 (i.e., 8) with all values set to Infinity, except the first value dpMinCoins[0], which is set to 0. This table represents the minimum number of coins needed to make up every amount from 0 to 7.

  dpMinCoins = [0, Infinity, Infinity, Infinity, Infinity, Infinity, Infinity, Infinity]

- Iterating through the DP Table: The outer loop iterates through the DP table starting from 1, up to the amount. For each amount i, it checks all possible coin denominations and updates the minimum number of coins needed.

  - i = 1: Check coins [1, 2, 5]. Since 1 ≤ 1, update dpMinCoins[1] as the minimum between its current value and dpMinCoins[1 - 1] + 1 = 1. The updated table becomes:

    dpMinCoins = [0, 1, Infinity, Infinity, Infinity, Infinity, Infinity, Infinity]

  - i = 2: Check coins [1, 2, 5]. Update dpMinCoins[2] with the minimum value between current and dpMinCoins[2 - 1] + 1 = 2 and dpMinCoins[2 - 2] + 1 = 1. The updated table becomes:

    dpMinCoins = [0, 1, 1, Infinity, Infinity, Infinity, Infinity, Infinity]

  - i = 3: Following the logic, the table becomes:

    dpMinCoins = [0, 1, 1, 2, Infinity, Infinity, Infinity, Infinity]

  - i = 4: The table becomes:

    dpMinCoins = [0, 1, 1, 2, 2, Infinity, Infinity, Infinity]

  - i = 5: The table becomes:

    dpMinCoins = [0, 1, 1, 2, 2, 1, Infinity, Infinity]

  - i = 6: The table becomes:

    dpMinCoins = [0, 1, 1, 2, 2, 1, 2, Infinity]

  - i = 7: The table becomes:

    - dpMinCoins = [0, 1, 1, 2, 2, 1, 2, 2]

- Final Result: The final answer is dpMinCoins[dpMinCoins.length - 1], which is dpMinCoins[7] = 2. This is the fewest number of coins needed to make up the amount 7, using the given denominations.

3. Arrays

3.1. Contains Duplicate (LeetCode: 217)

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

- Input: nums = [1,2,3,1]
- Output: true

Example 2:

- Input: nums = [1,2,3,4]
- Output: false

Example 3:

- Input: nums = [1,1,1,3,3,4,3,2,4,2]
- Output: true

3.1.1. JavaScript:

BootCamp JS > LC217_containsDuplicate > JS index.js > ...

```javascript
const containsDuplicate = nums => {
    const visitedNums = {}
    for (let i = 0; i < nums.length; i++){
        const num = nums[i];
        if (visitedNums[num]){
            return true;
        } else{
            visitedNums[num] = true;
        }
    }
    return false;
};
```

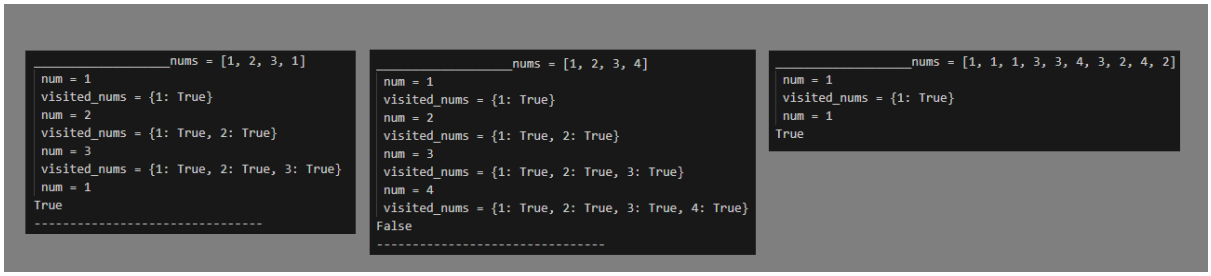
3.1.2. Python:

BootCamp PY > containsDuplicate.py > ...

```python
#LC: 217

def containsDuplicate(nums):
    visited_nums = {}
    for num in nums:
        if num in visited_nums:
            return True
        else:
            visited_nums[num] = True
    return False

```

3.1.3. Logic Explained:

- Initialization: A dictionary (visitedNums) is created to keep track of the numbers encountered so far.
- Iterating through the Array: The code then iterates through the array, and for each number (num), it checks if that number has been seen before by looking up the visitedNums dictionary.
- If Seen Before: If the number is found in the dictionary, that means the number is a duplicate, and the function immediately returns true.
- If Not Seen Before: If the number is not found in the dictionary, it is added to the dictionary with the value true to mark that this number has been seen.
- Final Result: If the loop completes without finding any duplicates, the function returns false, indicating that no duplicates were found.



3.2. Product of Array Except Self (LeetCode: 238)

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

- Input: nums = [1,2,3,4]
- Output: [24,12,8,6]

Example 2:

- Input: nums = [-1,1,0,-3,3]
- Output: [0,0,9,0,0]

3.2.1. JavaScript:

BootCamp JS > LC238_productExceptSelf > JS index.js > ...

```javascript
const productExceptSelf = nums => {
    let output = nums.map(n=> 1);
    let product = 1;

    // Multiply from the LEFT:
    for (let i = 0; i < nums.length; i++){
        output[i] = output[i] * product;
        product = product * nums[i];
    }

    product = 1;
    // Multiply from the RIGHT:
    for (let j = nums.length - 1; j>=0; j--){
        output[j] = output[j] * product;
        product = product * nums[j];
    }
    return output;
};
```

3.2.2. Python:

BootCamp PY > productExceptSelf.py > ...

```python
#LC: 238

def productExceptSelf(nums):
    output = [1 for _ in nums]
    product = 1

    # Multiply from the LEFT
    for i in range(len(nums)):
        output[i] *= product
        product *= nums[i]

    product = 1
    # Multiply from the RIGHT
    for j in range(len(nums) - 1, -1, -1):
        output[j] *= product
        product *= nums[j]

    return output

```

3.2.3. Logic Explained:

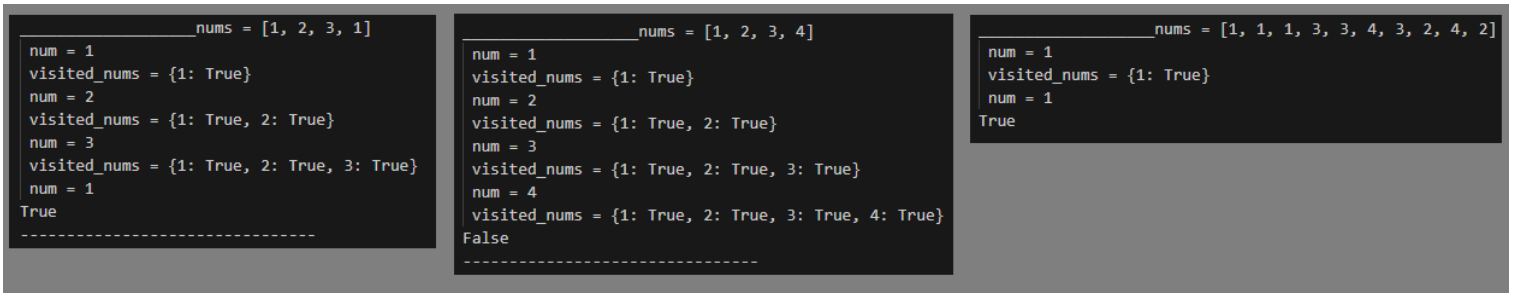
The code consists of two main parts, both involving iteration over the given array nums.

First Loop (Multiply from the LEFT):

- Initializes an output array with the same length as nums, all set to 1;
- Iterates over the nums array, and at each step, multiplies the corresponding element in the output array by the current product;
- Multiplies the product variable by the current number in nums, essentially accumulating the product of the numbers to the left of the current position.

Second Loop (Multiply from the RIGHT):

- Resets the product variable to 1;
- Iterates over the nums array in reverse order, and again multiplies the corresponding element in the output array by the current product;
- Multiplies the product variable by the current number in nums, accumulating the product of the numbers to the right of the current position.



3.3. Best Time to Buy and Sell Stock (LeetCode: 121)

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

- Input: prices = [7,1,5,3,6,4]
- Output: 5
- Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
- Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

- Input: prices = [7,6,4,3,1]
- Output: 0
- Explanation: In this case, no transactions are done and the max profit = 0.

3.3.1. JavaScript:

BootCamp JS > LC121_maxProfit > JS index.js > ...

```javascript
var maxProfit = function(prices) {
    let maxProfit = 0;
    let cheapestPrice = prices[0];

    for (let i = 0; i < prices.length; i++){
        const price = prices[i];
        if (price < cheapestPrice) cheapestPrice = price;

        const currentProfit = price - cheapestPrice;
        maxProfit = Math.max(currentProfit, maxProfit);
    }
return maxProfit;
};
```

3.3.2. Python:

BootCamp PY > buyStocks.py > ...

```python
def maxProfit(prices):
    max_profit = 0
    cheapest_price = prices[0]

    for price in prices:
        if price < cheapest_price:
            cheapest_price = price

        current_profit = price - cheapest_price
        max_profit = max(current_profit, max_profit)

    return max_profit

```

3.3.3. Logic Explained:

The logic behind the code involves keeping track of the cheapest price seen so far and the maximum profit that can be made using that cheapest price.

Understanding the Logic:

- Initialization: maxProfit is initialized to 0, and cheapestPrice is initialized to the first price in the array;
- Iteration: The code iterates through the prices array;
- Update Cheapest Price: If a price is found that is less than the current cheapestPrice, it updates cheapestPrice;
- Calculate Current Profit: For each price, the current profit is calculated as the difference between the current price and the cheapestPrice;
- Update Maximum Profit: If the currentProfit is greater than maxProfit, then maxProfit is updated to the value of currentProfit;
- Return Value: The final maxProfit value is returned, representing the maximum profit that can be made from buying and selling the stock.

```
PRICES: [7, 1, 5, 3, 6, 4]
##############################################
current_profit = price - cheapest_price => current_profit = 7 - 7
max_profit = max(current_profit, max_profit) => max_profit = max(0, 0)
----------------------------------------
cheapest_price = price => cheapest_price = 1
current_profit = price - cheapest_price => current_profit = 1 - 1
max_profit = max(current_profit, max_profit) => max_profit = max(0, 0)
----------------------------------------
current_profit = price - cheapest_price => current_profit = 5 - 1
max_profit = max(current_profit, max_profit) => max_profit = max(4, 0)
----------------------------------------
current_profit = price - cheapest_price => current_profit = 3 - 1
max_profit = max(current_profit, max_profit) => max_profit = max(2, 4)
----------------------------------------
current_profit = price - cheapest_price => current_profit = 6 - 1
max_profit = max(current_profit, max_profit) => max_profit = max(5, 4)
----------------------------------------
current_profit = price - cheapest_price => current_profit = 4 - 1
max_profit = max(current_profit, max_profit) => max_profit = max(3, 5)
----------------------------------------
5
```

3.4. Two Sum (LeetCode: 01)

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

- Input: nums = [2,7,11,15], target = 9;
- Output: [0,1];
- Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

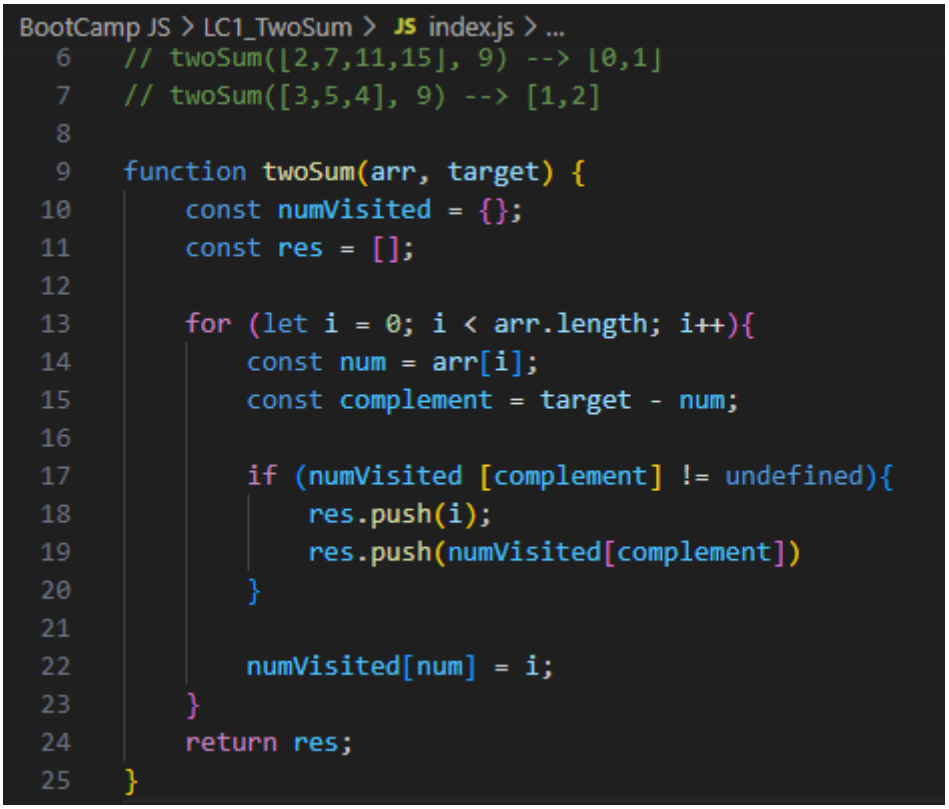
Example 2:

- Input: nums = [3,2,4], target = 6;
- Output: [1,2];

Example 3:

- Input: nums = [3,3], target = 6;
- Output: [0,1];

3.4.1. JavaScript:



3.4.2. Python:

3.4.3. Logic Explained:

The code functions by iterating through the array while keeping track of visited numbers and their indices in a hash table (dictionary in Python). The key idea is to find the complement of each number and check if it has been visited.

Understanding the Logic:

- Initialization: A dictionary numVisited is initialized to keep track of visited numbers and their indices. A result list res is initialized to store the result;
- Iteration: The code iterates through the arr:
    - Calculate Complement: For each number, its complement with respect to the target is calculated;
    - Check for Complement in Visited Numbers: If the complement is found in numVisited, the current index and the index of the complement are added to the result;
    - Mark Number as Visited: The current number and its index are stored in numVisited;
- Return Value: The final res list is returned, containing the indices of the two numbers that add up to the target.

```
num = 3
complement = target - num // complement = 6 - 3
--------------------------
num = 2
complement = target - num // complement = 6 - 2
--------------------------
num = 4
complement = target - num // complement = 6 - 4
--------------------------
res.append(i) // res.append(2)
res.append(num_visited[complement]) // res.append(1)
res = [2, 1]
```

4. Intervals

4.1. Meeting Rooms (LeetCode: 252)

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

Example 1:

- Input: intervals = [[0,30],[5,10],[15,20]];
- Output: false

Example 2:

- Input: intervals = [[7,10],[2,4]]
- Output: true

4.1.1. JavaScript:

```javascript
const canAttendMeetings = intervals => {
    if (intervals.length === 0) return true;
    const starts = [];
    const ends = [];

    for (let i = 0; i < intervals.length; i++){
        const subArray = intervals[i];
        starts.push(subArray[0]);
        ends.push(subArray[1]);
    }

    starts.sort((a,b) => a - b);
    ends.sort((a,b) => a - b);

    for (let i = 0; i < starts.length; i++){
        if (starts[i+1] < ends[i]) return false;
    }

    return true;

};
```

4.1.2. Python:

BootCamp PY > meetings.py > ...

```python
#LC: 252


def can_attend_meetings(intervals):
    print(" ")
    if len(intervals) == 0:
        return True
    starts = []
    ends = []

    for subArray in intervals:
        starts.append(subArray[0])
        ends.append(subArray[1])

    starts.sort()
    ends.sort()

    for i in range(len(starts) - 1):
        if starts[i + 1] < ends[i]:
            return False

    return True
```

4.1.3. Logic Explained:

The code segregates the start and end times of the meetings into two separate arrays and then sorts them. It then checks for overlaps between consecutive intervals.

Understanding the Logic:

- Initialization: Two lists starts and ends are initialized to store the start and end times of the intervals.
- Special Case Handling: If the intervals list is empty, the function returns True, as there are no meetings to attend.
- Filling the Lists: The code iterates through the given intervals, filling the starts and ends lists with the start and end times, respectively.
- Sorting the Lists: Both the starts and ends lists are sorted in ascending order. This ordering allows for a linear comparison of overlapping intervals.
- Checking Overlaps: The code iterates through the sorted starts list and checks if the start time of the next meeting is less than the end time of the current meeting. If so, it returns False. This comparison checks if any meeting starts before the previous one ends, implying a scheduling conflict.
- Return Value: If no overlaps are found, the function returns True, indicating that the person can attend all the meetings.

The code segregates the start and end times of the meetings into two separate arrays and then sorts them. It then checks for overlaps between consecutive intervals.

```
intervals: [[0, 30], [5, 10], [15, 20]]
##################################
starts = [0]
ends = [30]
----------------------
starts = [0, 5]
ends = [30, 10]
----------------------
starts = [0, 5, 15]
ends = [30, 10, 20]
----------------------
starts SORTED = [0, 5, 15]
ends SORTED = [10, 20, 30]
----------------------
if i + 1 < len(starts) and starts[i + 1] < ends[i]: return FALSE
if 0 + 1 < 3 and 5 < 10
False
```

```
intervals: [[7, 10], [2, 4]]
##################################
starts = [7]
ends = [10]
----------------------
starts = [7, 2]
ends = [10, 4]
----------------------
starts SORTED = [2, 7]
ends SORTED = [4, 10]
----------------------
if i + 1 < len(starts) and starts[i + 1] < ends[i]: return FALSE
if 0 + 1 < 2 and 7 < 4
True
```

4.2. Non-overlapping Intervals (LeetCode: 435)

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:

- Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
- Output: 1
- Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

Example 2:

- Input: intervals = [[1,2],[1,2],[1,2]]
- Output: 2
- Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.

Example 3:

- Input: intervals = [[1,2],[2,3]]
- Output: 0
- Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

4.2.1. JavaScript:

BootCamp JS > LC435_eraseOverlapIntervals > JS index.js > ...

```javascript
const eraseOverlapIntervals = intervals => {
    if (intervals.length === 0) return 0;

    let count = 0;

    intervals.sort(function(a,b) {return a[0] - b[0]});

    let end = intervals[0][1];

    for (let i = 1; i < intervals.length; i++){
        const interval = intervals[i];
        const intervalStart = intervals[0];
        const intervalEnd = intervals[1];

        if (intervalStart < end){
            count++
            end = Math.min(intervalEnd, end);
        } else{
            end = intervalEnd;
        }
    }
    return count;
};
```

4.2.2. Python:

BootCamp PY > eraseOverlapIntervals.py > ...

```python
#LC: 435
def eraseOverlapIntervals(intervals):
    if len(intervals) == 0:
        return 0

    count = 0
    intervals.sort(key=lambda x: x[0])

    end = intervals[0][1]

    for i in range(1, len(intervals)):
        interval = intervals[i]
        intervalStart = interval[0]
        intervalEnd = interval[1]

        if intervalStart < end:
            count += 1
            end = min(intervalEnd, end)
        else:
            end = intervalEnd

    return count
```

4.2.3. Logic Explained:

The code follows a Greedy Algorithm to minimize the number of intervals to remove for making the rest non-overlapping.

Understanding the Logic:

- Special Case Handling: If the intervals list is empty, the function returns 0, as there are no intervals to consider.
- Initialization: Count variable is initialized to store the number of overlaps. Intervals are sorted by their start times to facilitate linear traversal.
- Setting Initial End Time: The end time of the first interval is stored for future comparisons.
- Iterating through Intervals: The code iterates through the sorted intervals, starting from the second one:
    - A) If the start time of the current interval is less than the end time of the previous one, an overlap is found, and the count is incremented.
    - B) The end time is updated to the minimum of the current end time and the previous one.
    - C) If no overlap is found, the end time is updated to the current end time.
- Return Value: The function returns the count, representing the minimum number of intervals to remove.