Python Programming

Problem 1.1: [Easy] Maximum Average Subarray

```
You are given an integer array nums consisting of n elements, and an
integer k. Find a contiguous subarray whose length is equal to k that
has the maximum average value and return this value.

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000
Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

Example 2:
Input: nums = [5], k = 1
Output: 5.00000

Constraints:
n == nums.length
1 <= k <= n <= 10^5
-10^4 <= nums[i] <= 10^4
```

Solution:

1. **Given:** An array of numbers (like a list of scores, temperatures, or any other numerical data) and a number `k`.
2. **Goal:** Find the average value of any `k` consecutive numbers in the array. We're trying to find the highest average possible from all possible groups of `k` consecutive numbers.

Understanding the question:

- **Example Input:** `nums = [1, 12, -5, -6, 50, 3], k = 4`.
  - Here, we're asked to find the group of 4 consecutive numbers that has the highest average.
- **Breaking It Down:**
  - The first group of 4 numbers is `[1, 12, -5, -6]`.
  - The sum of these 4 numbers is `1 + 12 - 5 - 6 = 2`.
  - The average of this group is `2 / 4 = 0.5`.
  - The next group starts from the second number (`12`), and the next four numbers are `[12, -5, -6, 50]`.
  - The sum of this group is `12 - 5 - 6 + 50 = 51`.
  - The average of this group is `51 / 4 = 12.75`.

    - The final group starts from the third number (`-5`), and the next four numbers are `[-5, -6, 50, 3]`.
    - The sum of this group is `-5 - 6 + 50 + 3 = 42`.
    - The average of this group is `42 / 4 = 10.5`.
- The highest average of any 4-number group in this array is `12.75`.

Concept of "Sliding Window"

Instead of recalculating each sum entirely from scratch, we can save time by using the **sliding window** technique:

1. **Initial Sum:** Start by finding the sum of the first `k` numbers.
2. **Slide the Window:** For each subsequent group:
    - Subtract the first number that just "left" the window.
    - Add the new number that just "entered" the window.
3. **Track the Maximum Sum:** Keep a record of the highest sum seen so far.
4. **Compute the Average:** Once the largest sum is found, divide by `k` to get the highest average.

Example:

Example Input: `nums = [1, 12, -5, -6, 50, 3]`, `k = 4`.

1. **Initial Subarray:**
   Start by computing the sum of the first `k` elements.

```
Initial subarray (first 4 numbers): [1, 12, -5, -6]
Initial sum = 1 + 12 - 5 - 6 = 2
Initial average = 2 / 4 = 0.5
```

2. **Slide the Window Right by One Position:** To slide the window to the right:
    - Remove the leftmost number (`1` in this case) from the sum.
    - Add the next number (`50` in this case) to the sum.

```
Sliding Window (new subarray): [12, -5, -6, 50]
New sum = 2 - 1 + 50 = 51
New average = 51 / 4 = 12.75
```

3. Here, we've updated the sum by subtracting `1` (number no longer in the window) and adding `50` (new number in the window).
4. **Slide the Window Again:** Repeat this process by moving the window one step further to the right:
    - Remove the number `12`.
    - Add the number `3`.

Sliding Window (new subarray): [-5, -6, 50, 3]

```
New sum = 51 - 12 + 3 = 42
```

5. New average = 42 / 4 = 10.5

Implementation in Python:

```python
def find_max_average(nums, k):
    # Step 1: Calculate the sum of the first 'k' numbers
    current_sum = sum(nums[:k])  # Sum of the first 'k' numbers
    max_sum = current_sum  # This is the largest sum we've seen so far

    # Step 2: Slide the window one position at a time
    for i in range(k, len(nums)):
        # Adjust the sum by removing the first number of the previous group
        # and adding the next number in the sequence
        current_sum += nums[i] - nums[i - k]
        # Update the maximum sum if we find a larger sum
        max_sum = max(max_sum, current_sum)

    # Step 3: Divide by 'k' to get the highest average and return it
    return max_sum / k
```

Breakdown:

1. `def find_max_average(nums, k):`
    - This line defines a function called `find_max_average` that accepts two inputs:
        - `nums` (a list of numbers).
        - `k` (the number of consecutive numbers to consider).
2. `current_sum = sum(nums[:k])`
    - This calculates the sum of the first `k` numbers.
    - `nums[:k]` is a slice that selects the first `k` numbers from the list.
3. `max_sum = current_sum`
    - This initializes `max_sum` to hold the maximum sum we encounter. Initially, it's just the sum of the first `k` numbers.
4. `for i in range(k, len(nums)):`
    - A `for` loop starts at index `k` and goes up to the end of the list `nums`.
    - This loop moves the window of `k` numbers one step to the right each time.
5. `current_sum += nums[i] - nums[i - k]`

    - This line adjusts `current_sum` by:
        - Adding `nums[i]`, the next number in the list.
        - Subtracting `nums[i - k]`, which is the number that has "moved out" of the window.
6. `max_sum = max(max_sum, current_sum)`
    - This compares the new sum (`current_sum`) with the largest sum we've seen so far (`max_sum`), updating `max_sum` if necessary.
7. `return max_sum / k`
    - Once the loop ends, we return the highest average by dividing the largest sum (`max_sum`) by `k`.

Problem 1.2: [Medium] Reverse Words in a String

```
Given an input string s, reverse the order of the words.

A word is defined as a sequence of non-space characters. The words in s
will be separated by at least one space.

Return a string of the words in reverse order concatenated by a single
space.

Note that s may contain leading or trailing spaces or multiple spaces
between two words. The returned string should only have a single space
separating the words. Do not include any extra spaces.

Example 1:
Input: s = "the sky is blue"
Output: "blue is sky the"

Example 2:
Input: s = " hello world "
Output: "world hello"
Explanation: Your reversed string should not contain leading or trailing
spaces.

Example 3:
Input: s = "a good   example"
Output: "example good a"
Explanation: You need to reduce multiple spaces between two words to a
single space in the reversed string.

Constraints:
1 <= s.length <= 10^4
s contains English letters (upper-case and lower-case), digits, and
spaces ' '.
There is at least one word in s.
```

Solution:

Understanding the question:

- **Input:** A string `s` that contains words and spaces.
- **Goal:** Reverse the order of the words. Words are separated by spaces and should be joined back together with a single space between each word.
- **Example:**
    - `Input: "the sky is blue"`
    - `Output: "blue is sky the"`

Easy Solution

A straightforward solution involves using Python's built-in functions. We'll use these steps:

1. **Trim Extra Spaces:**
    - Remove leading and trailing spaces.
    - Replace multiple consecutive spaces between words with a single space.
2. **Split into Words:**
    - Split the string into words based on spaces.
3. **Reverse the List of Words:**
    - Reverse the order of the words.
4. **Join the Words:**
    - Concatenate the words back together, separated by a single space.

```
def reverse_words(s):
    # Step 1: Strip and split the string into words, removing extra
spaces
    words = s.strip().split()
    # Step 2: Reverse the list of words
    reversed_words = words[::-1]
    # Step 3: Join the words with a single space
    return ' '.join(reversed_words)
```

More Complex Approach

1. **Read Characters Backward:**
    - Start from the end of the string and read each character backward.
2. **Collect Words in Reverse Order:**
    - Skip over spaces and collect characters until a space is found, indicating a word boundary.
    - Once a word is collected, add it to a result list.
3. **Join Words Together:**
    - Use a loop to read backward through the string and maintain a result list of words in reverse order.
    - Join the words back together using a single space.

```
def reverse_words_manual(s):
# Initialize variables
result = []
length = len(s)
i = length - 1

# Main loop to collect words in reverse order
while i >= 0:
    # Skip any trailing spaces
```

```
    while i >= 0 and s[i] == ' ':
        i -= 1
    if i < 0:
        break

    # Find the start of the word
    word_end = i
    while i >= 0 and s[i] != ' ':
        i -= 1
    word_start = i + 1

    # Add the word to the result list
    result.append(s[word_start:word_end + 1])

# Join the words with a single space
return ' '.join(result)
```

Breakdown:

1. **Initialize Variables:**
    - `result`: An empty list to collect words in reverse order.
    - `length`: The total length of the input string `s`.
    - `i`: An index pointing to the last character of the string (`length - 1`).
2. **Main Loop to Collect Words in Reverse Order:**
    - The main loop uses `while i >= 0` to iterate backward from the last character to the first.
    - **Skipping Trailing Spaces:**
        i. Inside the loop, the first inner loop (`while i >= 0 and s[i] == ' '`) skips any trailing spaces.
        ii. Each iteration decrements `i` by one until a non-space character is found.
        iii. If `i` becomes less than `0` after skipping spaces, it means the entire string was empty or contained only spaces, so we break out of the loop.
    - **Identifying a Word's Start and End Indices:**
        i. Once a non-space character is encountered, the variable `word_end` is set to `i`.
        ii. A second inner loop (`while i >= 0 and s[i] != ' '`) continues to decrement `i` until a space is found or `i` becomes less than `0`.
        iii. This loop identifies the start of the word (next non-space character) and stops when a space is found.
        iv. The variable `word_start` is set to `i + 1` because `i` points to the space just before the word's start.

- **Add the Word to the Result List:**
    i. Extract the substring from `word_start` to `word_end + 1` and append it to the `result` list.
    ii. This means we've successfully collected one word and added it to the `result` in reverse order.

3. **Join the Collected Words:**
    - After the main loop completes, all words have been collected in reverse order inside the `result` list.
    - The `result` list is then converted into a final string using `' '.join(result)`.
    - This concatenates all words with a single space between them.
4. **Return the Reversed String:**
    - The final string is returned as the reversed version of the original input string.

Problem 1.3: [Easy] Unique Number of Occurrences

```
Given an array of integers arr, return true if the number of occurrences
of each value in the array is unique or false otherwise.

Example 1:
Input: arr = [1,2,2,1,1,3] Output: true
Explanation: The value 1 has 3 occurrences, 2 has 2 and 3 has 1. No two
values have the same number of occurrences.

Example 2:
Input: arr = [1,2] Output: false

Example 3:
Input: arr = [-3,0,1,-3,1,1,1,-3,10,0] Output: true

Constraints:
1 <= arr.length <= 1000 -1000 <= arr[i] <= 1000
```

Solution:

Overall Concept:

1. **Counting occurrences**: First, we need to count how often each number appears in the array.
2. **Checking uniqueness**: Then, we check if the counts of these occurrences are unique.

Methodology:

1. **Create a dictionary to count occurrences**: Loop through each element in the array and use a dictionary to keep track of how many times each element appears.
2. **Create another dictionary or set to check for uniqueness**: Loop through the counts and store each count in a set. If you find a count that is already in the set, return false. If the loop completes without duplicates, return true.

```python
def uniqueOccurrences(arr):
    # Step 1: Count occurrences
    occurrence_dict = {}
    for num in arr:
        if num in occurrence_dict:
            occurrence_dict[num] += 1
        else:
            occurrence_dict[num] = 1

    # Step 2: Check for unique occurrences
```

```
    occurrence_set = set()
    for count in occurrence_dict.values():
        if count in occurrence_set:
            return False
        occurrence_set.add(count)

    return True
```

Problem 1.4: [Medium] kth Largest Element in an Array

Given an integer array **nums**, return **true** if there exists a triple of indices (i, j, k) such that i < j < k and nums[i] < nums[j] < nums[k].

If no such indices exists, return **false**.

Example 1:

Input: nums = [1,2,3,4,5] Output: true

Explanation: Any triplet where i < j < k is valid.

Example 2:

Input: nums = [5,4,3,2,1] Output: false

Explanation: No triplet exists.

Example 3:

Input: nums = [2,1,5,0,4,6] Output: true

Explanation: The triplet (3, 4, 5) is valid because

nums[3] == 0 < nums[4] == 4 < nums[5] == 6.

Constraints:

1 <= nums.length <= 5 * 10^5

-2^31 <= nums[i] <= 2^31 - 1

Concept and Approach

The key idea is to traverse the array while maintaining the smallest and second smallest values found so far as potential first and second elements of the increasing triplet. By the end of the traversal, if we find a value that is greater than both, then an increasing triplet subsequence exists.

Steps

1. **Initialize two variables** to infinity (`first` and `second`) which will represent the smallest and second smallest elements of a potential triplet found so far.
2. **Traverse the array**: For each element (`num`) in the array:
    - If `num` is smaller than `first`, update `first` with `num`.
    - If `num` is greater than `first` but smaller than `second`, update `second` with `num`.
    - If `num` is greater than `second`, an increasing triplet is found.
3. **Return the result**: If a value greater than `second` is found during the traversal, return `true`. If the end of the array is reached without finding such a value, return `false`.

```
def increasingTriplet(nums):
    # Initialize two variables with infinity
    first, second = float('inf'), float('inf')

    # Traverse through the array
    for num in nums:
        if num <= first:
            # Update first if num is smaller than first
            first = num
        elif num <= second:
            # Update second if num is smaller than second but greater
than first
            second = num
        else:
            # If we find a number greater than both first and second, we
found a triplet
            return True

    # If we finish the loop without returning True, no triplet exists
    return False
```

Problem 1.5: [Easy] Valid Palindrome

```
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.

Constraints:
1 <= s.length <= 2 * 10^5
s consists only of printable ASCII characters.
```

Solution:

Steps:

Step 1: Normalize the String

To normalize the string, you can use a combination of Python's string methods and comprehension. The `str.isalnum()` method checks if a character is alphanumeric (i.e., either a letter or a number). You can iterate over each character in the string, convert it to lowercase with `str.lower()`, and keep it if it's alphanumeric.

Step 2: Check for Palindrome

A string is a palindrome if the string equals its reverse. You can reverse a string in Python using slicing (`s[::-1]`).

```python
def is_palindrome(s):
    """
    Check if the given string s is a palindrome, considering only alphanumeric characters and ignoring cases.
    """
    # Normalize the string: convert to lowercase and filter out non-alphanumeric characters
    filtered_s = ''.join([char.lower() for char in s if char.isalnum()])

    # Check if the normalized string is a palindrome
    return filtered_s == filtered_s[::-1]
```

Problem 1.6: [Medium] Maximum Subarray

```
Given an integer array nums, find the subarray with the largest sum, and
return its sum.

Example 1:
Input: nums = [-2,1,-3,4,-1,2,1,-5,4] Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:
Input: nums = [1] Output: 1
Explanation: The subarray [1] has the largest sum 1.

Example 3:
Input: nums = [5,4,-1,7,8] Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:
1 <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
```

Solution:

Given an integer array `nums`, the task is to find the subarray (i.e., contiguous elements) that has the largest sum and return this maximum sum. We can solve this problem using Kadane's Algorithm.

Kadane's Algorithm: Concept

Kadane's Algorithm works by iterating through the array and maintaining two values during the iteration:

- **`current_sum`**: The maximum sum of the subarray that ends at the current position.
- **`max_sum`**: The maximum sum encountered so far across all subarrays considered.

Steps of Kadane's Algorithm

1. **Initialize** `current_sum` to 0 and `max_sum` to negative infinity (or the smallest possible integer value) to handle arrays containing all negative numbers.
2. **Iterate through the array**:
    - Update `current_sum` by adding the current element (`nums[i]`).
    - If `current_sum` is better than `max_sum`, update `max_sum` to `current_sum`.
    - If `current_sum` becomes negative, reset it to 0. This step effectively ignores any previous subarray whose sum would decrease the sum of a future subarray.
3. **Return** `max_sum` as it contains the maximum subarray sum.

```python
def maxSubArray(nums):
    """
    Finds the maximum sum of a contiguous subarray in the given integer array `nums`.
    """
    # Initialize current_sum to 0 and max_sum to the smallest possible integer
    current_sum = 0
    max_sum = float('-inf')  # or use min(nums) to start with the smallest element in nums

    for num in nums:
        # If current_sum is negative, start new subarray from the current element
        if current_sum < 0:
            current_sum = 0

        # Add the current element to current_sum
        current_sum += num

        # Update max_sum if current_sum is better
        if current_sum > max_sum:
            max_sum = current_sum

    return max_sum
```

- **Initialization**: We start with `current_sum` set to 0 to begin the summing from the first element, and `max_sum` is initialized to the smallest possible value to ensure it gets updated correctly during the iteration.
- **Iteration**: We add each number to `current_sum` and check if it improves the `max_sum`. If `current_sum` becomes negative at any point, it indicates that starting a new subarray from the next position might yield a better result, so we reset `current_sum` to zero.
- **Comparison and Reset**: The comparison `current_sum > max_sum` ensures that we always have the best possible sum tracked in `max_sum`.

Problem 1.7 [Medium] Method Chaining (pandas)

You are provided with a DataFrame called “Animals” with the following columns:

Column Name	Type
name	Object
species	object
age	int
weight	int

Write a solution to list the names of animals that weigh strictly more than 100 kilograms.

Return the animals sorted by weight in descending order.

The result format is in the following example.

Example 1:

Input:

name	species	age	weight
Tatiana	Snake	98	464
Khaled	Giraffe	50	41
Alex	Leopard	6	328
Jonathan	Monkey	45	463
Stefan	Bear	100	50
Tommy	Panda	26	349

Output:

name

Tatiana
Jonathan
Tommy
Alex

Explanation:

All animals weighing more than 100 should be included in the results table.

Tatiana's weight is 464, Jonathan's weight is 463, Tommy's weight is 349, and Alex's weight is 328.

The results should be sorted in descending order of weight.

Solution:

```
import pandas as pd
# Step 1: Filter the DataFrame to include only animals weighing more
than 100 kg
heavy_animals = animals[animals['weight'] > 100]

# Step 2: Sort the filtered DataFrame by weight in descending order
sorted_heavy_animals = heavy_animals.sort_values('weight',
ascending=False)

# Step 3: Select the 'name' column
animal_names = sorted_heavy_animals['name']

# Step 4: Convert the resulting Series to a DataFrame (if needed)
animal_names_df = animal_names.to_frame()

print(animal_names_df)
```

Explanation of Each Step

1. **Filtering**: Create a new DataFrame `heavy_animals` that includes only those animals whose weight exceeds 100 kilograms. This is achieved by applying a boolean condition directly to the DataFrame.

2. **Sorting**: The `heavy_animals` DataFrame is then sorted based on the `weight` column. The `sort_values` function is used with the `ascending=False` parameter to ensure the data is sorted in descending order.
3. **Selecting Column**: Extract only the `name` column from the sorted DataFrame. This results in a pandas Series.
4. **DataFrame Conversion**: Convert the Series `animal_names` back into a DataFrame. This step might be required if the output specification demands a DataFrame format.

Problem 1.8: [Medium] Dot Product of Two Sparse Vectors

Compute the dot product of two sparse vectors.

Definition: A sparse vector is an array where most of the elements are zero. For efficiency, we only store the non-zero elements.

Operations:

1. **Initialization**: Create a sparse vector from a list of numbers.
2. **Dot Product**: Calculate the dot product of two sparse vectors.

Explanation:

- **Dot Product**: Multiply corresponding elements of the two vectors and sum the results. Only non-zero elements contribute to the final result since multiplying by zero always results in zero.

1. **Example 1**:
    - **Input**: nums1 = [1,0,0,2,3], nums2 = [0,3,0,4,0]
    - **Output**: 8
    - **Explanation**: Compute as follows:
        - Multiply corresponding elements: (1×0) + (0×3) + (0×0) + (2×4) + (3×0)
        - Sum the products: 0 + 0 + 0 + 8 + 0 = 8
2. **Example 2**:
    - **Input**: nums1 = [0,1,0,0,0], nums2 = [0,0,0,0,2]
    - **Output**: 0
    - **Explanation**: All products of corresponding elements are zero:
        - (0×0) + (1×0) + (0×0) + (0×0) + (0×2) = 0
3. **Example 3**:
    - **Input**: nums1 = [0,1,0,0,2,0,3], nums2 = [1,0,0,0,3,0,4]
    - **Output**: 18
    - **Explanation**: Compute non-zero contributions:
        - (0×1) + (1×0) + (0×0) + (0×0) + (2×3) + (0×0) + (3×4) = 0 + 0 + 0 + 0 + 6 + 0 + 0 = 18

Solution:

Step 1: Representation of Sparse Vector

We will represent the sparse vector in a hash map (or dictionary in Python) where:

- The **key** is the index of a non-zero element.
- The **value** is the value of the element at that index.

For instance, a vector `[1,0,0,2,3]` would be stored as `{0: 1, 3: 2, 4: 3}`. This representation allows us to quickly access the non-zero elements of the vector without having to iterate through potentially many zero elements.

Step 2: Computing Dot Product

To compute the dot product of two sparse vectors efficiently:

- We iterate over the non-zero elements of the smaller vector (in terms of non-zero elements).
- For each non-zero element, we check if the corresponding index in the other vector also has a non-zero value.
- If both indices have non-zero values, we multiply these values and add the result to a cumulative sum.

Edge Case Handling

- If one of the vectors is empty or entirely zero, the dot product is naturally zero.

```
def convert_to_sparse_vector(nums):
    """
    Convert a list of numbers into a sparse vector represented by a
dictionary.
    Only non-zero elements are stored, with the index as the key and the
element as the value.
    """
    sparse_vector = {}
    for index, value in enumerate(nums):
        if value != 0:
            sparse_vector[index] = value
    return sparse_vector

def dot_product(sparse_vec1, sparse_vec2):
    """
    Compute the dot product of two sparse vectors represented as
dictionaries.
    Iterate over the smaller dictionary to optimize performance.
```

```
    """
    if len(sparse_vec1) > len(sparse_vec2):
        sparse_vec1, sparse_vec2 = sparse_vec2, sparse_vec1  # Swap to
ensure sparse_vec1 is smaller

    result = 0
    for index, value in sparse_vec1.items():
        if index in sparse_vec2:
            result += value * sparse_vec2[index]
    return result
```

Problem 1.9: [Medium] Buildings With an Ocean View

```
There are n buildings in a line. You are given an integer array heights
of size n that represents the heights of the buildings in the line.

The ocean is to the right of the buildings. A building has an ocean view
if the building can see the ocean without obstructions. Formally, a
building has an ocean view if all the buildings to its right have a
smaller height.

Return a list of indices (0-indexed) of buildings that have an ocean
view, sorted in increasing order.

Example 1:
Input: heights = [4,2,3,1] Output: [0,2,3]
Explanation: Building 1 does not have an ocean view because building 2
is taller.

Example 2:
Input: heights = [4,3,2,1] Output: [0,1,2,3]
Explanation: All the buildings have an ocean view.

Example 3:
Input: heights = [1,3,2,4] Output: [3]
Explanation: Only building 3 has an ocean view.

Constraints:
1 <= heights.length <= 10^5
1 <= heights[i] <= 10^9
```

Solution:

Each building can be visualized as part of a skyline when looking from left to right. A building has an unobstructed view of the ocean if there are no taller buildings between it and the ocean (to the right of it).

Strategy

To efficiently find the buildings with an ocean view, we can start checking from the rightmost building towards the left. This way, we can easily keep track of the tallest building we've seen so far, and determine if the current building has a taller one to its right.

Steps

1. **Initialize Variables**:
    - `max_height_so_far`: to store the maximum height observed as we move from right to left. Initially set to `-1` (or `0` if all heights are positive), as we haven't seen any building yet.
    - `ocean_view_buildings`: a list to store the indices of buildings that have an ocean view.
2. **Traverse from Right to Left**:
    - Start from the last building (rightmost) and move towards the first building (leftmost).
    - Compare the current building's height with `max_height_so_far`.
3. **Check for Ocean View**:
    - If the current building's height is greater than `max_height_so_far`, it means this building has an ocean view because it is taller than all the buildings to its right.
    - Update `max_height_so_far` to this building's height if it's greater.
    - Add the current building's index to `ocean_view_buildings`.
4. **Final Adjustment**:
    - Since we have stored indices starting from the last building to the first, we need to reverse the `ocean_view_buildings` list to provide the answer in increasing order of indices.

```
def findBuildingsWithOceanView(heights):
    n = len(heights)  # Total number of buildings
    max_height_so_far = 0  # To track the maximum height seen so far
from the right
    ocean_view_buildings = []  # To collect indices of buildings with
ocean views

    # Traverse from right to left
    for i in range(n-1, -1, -1):
        if heights[i] > max_height_so_far:
            ocean_view_buildings.append(i)  # This building has an ocean
view
            max_height_so_far = heights[i]  # Update the maximum height
seen so far

    # Since we collected buildings from right to left, we need to
reverse the list
    ocean_view_buildings.reverse()

    return ocean_view_buildings
```

Problem 1.10: [Medium] Generate Parentheses

```
Given n pairs of parentheses, write a function to generate all
combinations of well-formed parentheses.

Example 1:
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]

Example 2:
Input: n = 1
Output: ["()"]
```

Solution:

1. **Understanding the Problem:** For n pairs of parentheses, we need to generate strings that have n open brackets (and n close brackets) that form a valid sequence. The constraint is that at no point in the sequence can the number of close brackets) exceed the number of open brackets (.
2. **Using Recursion:** We'll define a helper function that takes parameters to track the number of open brackets and close brackets used so far. The function will also take the current string of parentheses formed. The recursion will proceed by adding an open bracket if not all n open brackets are used, and adding a close bracket if the number of close brackets used is less than the number of open brackets.
3. **Base Case:** The base case for the recursion will be when the length of the current string is 2n, which means we have used n open and n close brackets. At this point, we add the current string to the result list.
4. **Recursive Calls:**
    - Add an open bracket (if the count of open brackets is less than n.
    - Add a close bracket) if the count of close brackets is less than the count of open brackets.

```
def generateParenthesis(n):
    def backtrack(S='', left=0, right=0):
```

```
        # Base case: If the length of S is 2 * n, we append the current
string S to the result
        if len(S) == 2 * n:
            result.append(S)
            return
        # If the number of '(' used is less than n, we can still place
'('
        if left < n:
            backtrack(S + '(', left + 1, right)
        # If the number of ')' used is less than '(', we can place ')'
        if right < left:
            backtrack(S + ')', left, right + 1)

    result = []
    backtrack()
    return result
```

Explanation::

- `generateParenthesis(n)`: This is the main function that initializes the process.
- `backtrack(S, left, right)`: This is a helper function defined inside the main function. `S` holds the current sequence of parentheses, `left` counts the number of `(` used, and `right` counts the number of `)` used.
- We use conditional statements to decide whether to add more `(` or `)` to `S`.
- The function recursively constructs the string and backtracks when a potential sequence reaches an invalid state or completes a valid sequence.
- The use of `left` and `right` ensures that no sequence ever has more `)` before having a matching number of `(`, which is crucial for maintaining well-formedness.

SQL Programming

Problem 2.1: [Medium] Employees Earning More Than Their Managers

Given an `Employee` table with the following schema:

Column Name	Type
id	int
name	varchar
salary	int
managerId	int

- `id` is the primary key for this table.
- Each row of this table indicates the ID of an employee, their name, salary, and the ID of their manager.

Write a solution to find the employees who earn more than their managers.

Output: Return the result table with the names of employees who earn more than their managers. The result can be returned in any order.

Example

Input

Employee table:

id	name	salary	managerId
1	Joe	70000	3
2	Henry	80000	4
3	Sam	60000	Null

4	Max	90000	Null

Output

Employee
Joe

Explanation

- Joe earns $70,000 and his manager (Sam) earns $60,000. Therefore, Joe earns more than his manager.
- Henry and Max do not have managers earning less than them.
- Sam and Max do not have managers.

Solution

Steps:

1. **Select the necessary columns:** We need to select the employee's name and compare their salary with their manager's salary.
2. **Self-join the `Employee` table:** We will join the `Employee` table with itself to match each employee with their manager.
3. **Compare salaries:** Filter the results to find employees whose salary is greater than their manager's salary.
4. **Return the employee names:** Select the names of the employees who meet the condition.

```
SELECT e1.name AS Employee
FROM Employee e1
JOIN Employee e2 ON e1.managerId = e2.id
WHERE e1.salary > e2.salary;
```

Explanation:

1. **Self-join the `Employee` table:**
    - We alias the `Employee` table as `e1` for employees and `e2` for managers.
    - The condition `e1.managerId = e2.id` matches each employee (`e1`) with their manager (`e2`).

2. **Filter employees who earn more than their managers:**
    - The condition `e1.salary > e2.salary` ensures that only the employees who earn more than their managers are selected.
3. **Select the employee names:**
    - We select the `name` column from the `e1` alias to get the names of the employees who meet the condition.

Example

Given the `Employee` table:

id	name	salary	managerId
1	Joe	70000	3
2	Henry	80000	4
3	Sam	60000	Null
4	Max	90000	Null

Execution of the query:

1. **Self-join:**
    - `e1` table (employees): `(1, Joe, 70000, 3)`, `(2, Henry, 80000, 4)`, `(3, Sam, 60000, Null)`, `(4, Max, 90000, Null)`
    - `e2` table (managers): `(1, Joe, 70000, 3)`, `(2, Henry, 80000, 4)`, `(3, Sam, 60000, Null)`, `(4, Max, 90000, Null)`
2. **Matching employees with managers:**
    - Joe's manager is Sam.
    - Henry's manager is Max.
3. **Compare salaries:**
    - Joe earns 70000, Sam earns 60000 (Joe earns more than his manager).
    - Henry earns 80000, Max earns 90000 (Henry does not earn more than his manager).
4. **Select names:**
    - Joe is the only employee who earns more than their manager.

Output:

Employee
Joe

Problem 2.2: [Medium] Second Highest Salary

Given an `Employee` table with the following schema:

Column Name	Type
id	int
salary	int

- `id` is the primary key for this table.
- Each row of this table contains information about the salary of an employee.

Write a solution to find the second highest salary from the `Employee` table. If there is no second highest salary, return null.

Example 1

Input:

Employee table:

id	salary
1	100
2	200
3	300

Output:

SecondHighestSalary
200

Example 2

Input:

Employee table:

id	salary
1	100

Output:

SecondHighestSalary
null

Solution:

To find the second highest salary from the `Employee` table, we can use a subquery to first determine the highest salary, then find the highest salary that is less than this maximum salary. If no such salary exists, we should return `null`.

```
SELECT
    (SELECT DISTINCT salary
     FROM Employee
     ORDER BY salary DESC
     LIMIT 1 OFFSET 1) AS SecondHighestSalary;
```

Explanation:

1. **Subquery to find the second highest salary:**
    - `SELECT DISTINCT salary FROM Employee`: Selects all distinct salaries from the `Employee` table.
    - `ORDER BY salary DESC`: Orders the salaries in descending order.
    - `LIMIT 1 OFFSET 1`: Skips the highest salary (offset 1) and then takes the next highest salary (limit 1).
2. **Handle case when there is no second highest salary:**
    - If the `LIMIT 1 OFFSET 1` does not return any rows (because there is only one unique salary or no salaries at all), the result will be `null`.

Problem 2.3: [Hard] Consecutive Transactions With Increasing Amounts

Given a `Transactions` table with the following schema:

Column Name	Type
transaction_id	int
customer_id	int
transaction_date	date
amount	int

- `transaction_id` is the primary key for this table.
- Each row contains information about transactions that includes unique `(customer_id, transaction_date)` pairs along with the corresponding `customer_id` and `amount`.

Write an SQL query to find the customers who have made consecutive transactions with increasing amounts for at least three consecutive days. Include the `customer_id`, start date of the consecutive transactions period, and the end date of the consecutive transactions period. There can be multiple sets of consecutive transactions by a customer.

Output:Return the result table ordered by `customer_id` in ascending order.

Example:

Input

Transactions table:

transaction_id	customer_id	transaction_date	amount
1	101	2023-05-01	100
2	101	2023-05-02	150
3	101	2023-05-03	200
4	102	2023-05-01	50
5	102	2023-05-03	100
6	102	2023-05-04	200

7	105	2023-05-01	100
8	105	2023-05-02	150
9	105	2023-05-03	200
10	105	2023-05-04	300
11	105	2023-05-12	250
12	105	2023-05-13	260
13	105	2023-05-14	270

Output

customer_id	consecutive_start	consecutive_end
101	2023-05-01	2023-05-03
105	2023-05-01	2023-05-04
105	2023-05-12	2023-05-14

Explanation

- `customer_id` 101 has made consecutive transactions with increasing amounts from May 1st, 2023, to May 3rd, 2023.
- `customer_id` 102 does not have any consecutive transactions for at least 3 days.
- `customer_id` 105 has two sets of consecutive transactions: from May 1st, 2023, to May 4th, 2023, and from May 12th, 2023, to May 14th, 2023.

`customer_id` is sorted in ascending order.

Solution:

We have a `Transactions` table that records transactions made by customers. Each transaction has:

- `transaction_id`: Unique identifier for the transaction.
- `customer_id`: Identifier for the customer who made the transaction.
- `transaction_date`: Date when the transaction occurred.
- `amount`: Amount of the transaction.

Our goal is to find customers who have made consecutive transactions with increasing amounts for at least three consecutive days. We need to return the `customer_id`, the start date of the consecutive period (`consecutive_start`), and the end date of the consecutive period (`consecutive_end`). The result should be ordered by `customer_id`.

Steps:

1. **Assign Row Numbers to Transactions**: Use the `ROW_NUMBER()` function to uniquely identify the order of transactions for each customer.
2. **Identify Consecutive and Increasing Transactions**: Use the `LAG()` function to compare each transaction with the previous one to check if the transaction dates are consecutive and if the amounts are increasing.
3. **Group Consecutive Transactions**: Use the `SUM()` function to group transactions into sequences of consecutive increasing transactions.
4. **Filter for Valid Sequences**: Select sequences that are at least three days long.
5. **Output the Results**: Select the required information for valid sequences and order by `customer_id`.

Steps with code snippets:

Step 1: Assign Row Numbers to Transactions

We start by assigning a row number to each transaction per customer, ordered by the transaction date. This helps in comparing each transaction with the previous one.

```
WITH NumberedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
transaction_date) AS row_num
    FROM
        Transactions
)
```

- **`ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY transaction_date)`:** This assigns a unique row number to each transaction within each `customer_id` group, ordered by `transaction_date`.

Step 2: Identify Consecutive and Increasing Transactions

Next, we use the `LAG()` function to compare each transaction with the previous one. We create a flag (`new_sequence`) to mark the start of a new sequence if the transactions are not consecutive or the amounts are not increasing.

```
, FlaggedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
```

```
        amount,
        row_num,
        CASE
            WHEN LAG(transaction_date) OVER (PARTITION BY customer_id
ORDER BY transaction_date) = transaction_date - INTERVAL 1 DAY
                 AND LAG(amount) OVER (PARTITION BY customer_id ORDER BY
transaction_date) < amount
            THEN 0
            ELSE 1
        END AS new_sequence
    FROM
        NumberedTransactions
)
```

- **`LAG(transaction_date) OVER (PARTITION BY customer_id ORDER BY transaction_date)`:** This gets the transaction date of the previous transaction for the same customer.
- **`LAG(amount) OVER (PARTITION BY customer_id ORDER BY transaction_date)`:** This gets the amount of the previous transaction for the same customer.
- **`CASE WHEN ... THEN 0 ELSE 1 END`:** This creates a `new_sequence` flag:
    - `0` if the previous transaction date is consecutive (`transaction_date - INTERVAL 1 DAY`) and the amount is less than the current amount.
    - `1` otherwise, indicating the start of a new sequence.

Step 3: Group Consecutive Transactions

We then use the `SUM()` function to create a group identifier for each sequence of consecutive transactions.

```
, GroupedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        row_num,
        SUM(new_sequence) OVER (PARTITION BY customer_id ORDER BY
transaction_date) AS sequence_group
    FROM
        FlaggedTransactions
)
```

- **`SUM(new_sequence) OVER (PARTITION BY customer_id ORDER BY transaction_date)`:** This calculates a running total of the `new_sequence` flag, effectively grouping transactions into sequences. Each change from `1` to `0` in the `new_sequence` flag starts a new group.

Step 4: Filter for Valid Sequences

Next, we filter out the sequences that are less than three days long.

```
, FilteredSequences AS (
    SELECT
        customer_id,
        MIN(transaction_date) AS consecutive_start,
        MAX(transaction_date) AS consecutive_end,
        COUNT(*) AS sequence_length
    FROM
        GroupedTransactions
    GROUP BY
        customer_id,
        sequence_group
    HAVING
        sequence_length >= 3
)
```

- **`MIN(transaction_date) AS consecutive_start, MAX(transaction_date) AS consecutive_end`:** These get the start and end dates of each sequence.
- **`COUNT(*) AS sequence_length`:** This counts the number of transactions in each sequence.
- **`HAVING sequence_length >= 3`:** This filters the sequences to include only those with at least three transactions.

Step 5: Output the Results

Finally, we select the required information and order the results by `customer_id`.

```
SELECT
    customer_id,
    consecutive_start,
    consecutive_end
FROM
    FilteredSequences
ORDER BY
    customer_id;
```

Full SQL Query:

```
WITH NumberedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
transaction_date) AS row_num
    FROM
        Transactions
),
FlaggedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        row_num,
        CASE
            WHEN LAG(transaction_date) OVER (PARTITION BY customer_id
ORDER BY transaction_date) = transaction_date - INTERVAL 1 DAY
                 AND LAG(amount) OVER (PARTITION BY customer_id ORDER BY
transaction_date) < amount
            THEN 0
            ELSE 1
        END AS new_sequence
    FROM
        NumberedTransactions
),
GroupedTransactions AS (
    SELECT
        customer_id,
        transaction_date,
        amount,
        row_num,
        SUM(new_sequence) OVER (PARTITION BY customer_id ORDER BY
transaction_date) AS sequence_group
    FROM
        FlaggedTransactions
),
FilteredSequences AS (
    SELECT
        customer_id,
        MIN(transaction_date) AS consecutive_start,
        MAX(transaction_date) AS consecutive_end,
        COUNT(*) AS sequence_length
    FROM
```

```
        GroupedTransactions
    GROUP BY
        customer_id,
        sequence_group
    HAVING
        sequence_length >= 3
)
SELECT
    customer_id,
    consecutive_start,
    consecutive_end
FROM
    FilteredSequences
ORDER BY
    customer_id;
```

Problem 2.4: [Hard] Confirmation Rate

Given the following tables:

Table: Signups

Column Name	Type
user_id	int
time_stamp	datetime

- `user_id` is the column of unique values for this table.
- Each row contains information about the signup time for the user with ID `user_id`.

Table: Confirmations

Column Name	Type
user_id	int
time_stamp	datetime
action	ENUM

- `(user_id, time_stamp)` is the primary key for this table.
- `user_id` is a foreign key referencing the `Signups` table.
- `action` is an ENUM with values ('confirmed', 'timeout').
- Each row indicates that the user with ID `user_id` requested a confirmation message at `time_stamp` and the message was either confirmed ('confirmed') or expired without confirming ('timeout').

The confirmation rate of a user is calculated as the number of 'confirmed' messages divided by the total number of requested confirmation messages. If a user did not request any confirmation messages, their confirmation rate is 0. Round the confirmation rate to two decimal places.

Output: Return the result table with the confirmation rate of each user. The result can be returned in any order.

Example

Input

Signups table:

user_id	time_stamp
3	2020-03-21 10:16:13
7	2020-01-04 13:57:59
2	2020-07-29 23:09:44
6	2020-12-09 10:39:37

Confirmations table:

user_id	time_stamp	action
3	2021-01-06 03:30:46	timeout
3	2021-07-14 14:00:00	timeout
7	2021-06-12 11:57:29	confirmed
7	2021-06-13 12:58:28	confirmed

7	2021-06-14 13:59:27	confirmed
2	2021-01-22 00:00:00	confirmed
2	2021-02-28 23:59:59	timeout

Output

user_id	confirmation_rate
6	0.00
3	0.00
7	1.00
2	0.50

Explanation

- User 6 did not request any confirmation messages. The confirmation rate is 0.
- User 3 made 2 requests and both timed out. The confirmation rate is 0.
- User 7 made 3 requests and all were confirmed. The confirmation rate is 1.
- User 2 made 2 requests where one was confirmed and the other timed out. The confirmation rate is 1 / 2 = 0.5.

Solution:

We need to calculate the confirmation rate for each user based on the `Signups` and `Confirmations` tables. The confirmation rate is defined as the number of 'confirmed' messages divided by the total number of requested confirmation messages. If a user did not request any confirmation messages, their confirmation rate should be 0. We also need to return the results with the confirmation rate rounded to two decimal places.

Steps:

Step 1: Count Total Number of Confirmation Requests for Each User

We first need to determine how many confirmation requests each user has made. This includes both 'confirmed' and 'timeout' actions. We use the `COUNT(*)` function, which counts the number of rows that match the criteria (grouped by `user_id`).

```
SELECT
    user_id,
    COUNT(*) AS total_requests
FROM
    Confirmations
GROUP BY
    user_id
```

- **`COUNT(*)`:** Counts all rows for each `user_id`.
- **`GROUP BY user_id`:** Groups the results by `user_id` to get a count for each user.

Step 2: Count Number of Confirmed Requests for Each User

Next, we need to count how many of these requests were actually confirmed. This will give us the numerator for our confirmation rate calculation. We filter the rows to include only those where `action = 'confirmed'`.

```
SELECT
    user_id,
    COUNT(*) AS confirmed_requests
FROM
    Confirmations
WHERE
    action = 'confirmed'
GROUP BY
    user_id
```

- **`WHERE action = 'confirmed'`:** Filters the rows to count only confirmed actions.
- **`COUNT(*)`:** Counts the filtered rows for each `user_id`.
- **`GROUP BY user_id`:** Groups the results by `user_id` to get a count for each user.

Step 3: Calculate the Confirmation Rate

We need to join the results of the total requests and confirmed requests to calculate the confirmation rate. We'll use a `LEFT JOIN` to ensure that all users are included, even if they have no confirmation requests.

```
SELECT
    c.user_id,
    COALESCE(cc.confirmed_requests, 0) / COALESCE(c.total_requests, 0)
AS confirmation_rate
FROM
    (SELECT
        user_id,
        COUNT(*) AS total_requests
     FROM
        Confirmations
     GROUP BY
        user_id) c
LEFT JOIN
    (SELECT
        user_id,
        COUNT(*) AS confirmed_requests
     FROM
        Confirmations
     WHERE
        action = 'confirmed'
     GROUP BY
        user_id) cc
ON c.user_id = cc.user_id
```

- **`COALESCE()`:** Replaces `NULL` values with 0. This is necessary because some users may have no confirmed requests or no total requests.
- **`c`:** Alias for the subquery that counts total requests.
- **`cc`:** Alias for the subquery that counts confirmed requests.
- **`LEFT JOIN`:** Ensures that every user from the `total_requests` subquery is included, even if they have no confirmed requests.

Step 4: Handle Users with No Confirmation Requests

We need to join the above result with the `Signups` table to include all users, even those who have not made any confirmation requests. We also calculate the confirmation rate, rounding it to two decimal places.

```
SELECT
    s.user_id,
    ROUND(COALESCE(cc.confirmed_requests, 0) /
```

```
COALESCE(c.total_requests, 0), 2) AS confirmation_rate
FROM
    Signups s
LEFT JOIN
    (SELECT
        user_id,
        COUNT(*) AS total_requests
     FROM
        Confirmations
     GROUP BY
        user_id) c
ON s.user_id = c.user_id
LEFT JOIN
    (SELECT
        user_id,
        COUNT(*) AS confirmed_requests
     FROM
        Confirmations
     WHERE
        action = 'confirmed'
     GROUP BY
        user_id) cc
ON s.user_id = cc.user_id
ORDER BY
    s.user_id;
```

- **`Signups s`**: The main table containing all users.
- **`LEFT JOIN ... ON s.user_id = c.user_id`**: Ensures all users from `Signups` are included, matching with their total request counts.
- **`LEFT JOIN ... ON s.user_id = cc.user_id`**: Ensures all users from `Signups` are included, matching with their confirmed request counts.
- **`ROUND(..., 2)`**: Rounds the confirmation rate to two decimal places.

Final SQL Query

```
SELECT
    s.user_id,
    ROUND(COALESCE(cc.confirmed_requests, 0) /
COALESCE(c.total_requests, 0), 2) AS confirmation_rate
FROM
    Signups s
LEFT JOIN
    (SELECT
        user_id,
```

```
        COUNT(*) AS total_requests
    FROM
        Confirmations
    GROUP BY
        user_id) c
ON s.user_id = c.user_id
LEFT JOIN
    (SELECT
        user_id,
        COUNT(*) AS confirmed_requests
    FROM
        Confirmations
    WHERE
        action = 'confirmed'
    GROUP BY
        user_id) cc
ON s.user_id = cc.user_id
ORDER BY
    s.user_id;
```

Summary:

1. **Calculate total confirmation requests per user**: We count all confirmation requests for each user.
2. **Calculate confirmed requests per user**: We count only the confirmed requests for each user.
3. **Join these counts and handle missing data**: We join the counts using `LEFT JOIN` to ensure all users are included, and use `COALESCE` to handle users with no requests.
4. **Join with `Signups` to include all users**: We join the result with the `Signups` table to include users with no confirmation requests.
5. **Calculate and round the confirmation rate**: We calculate the confirmation rate and round it to two decimal places, ensuring correct formatting.

Problem 2.5: [Medium] Rank Scores

Given a `Scores` table with the following schema:

Column Name	Type
id	int
score	decimal

- `id` is the primary key for this table.
- Each row of this table contains the score of a game. The score is a floating-point value with two decimal places.

Write a solution to find the rank of the scores. The ranking should be calculated according to the following rules:

1. The scores should be ranked from the highest to the lowest.
2. If there is a tie between two scores, both should have the same ranking.
3. After a tie, the next ranking number should be the next consecutive integer value (i.e., there should be no gaps between ranks).

Return the result table ordered by score in descending order.

Output:Return the result table with the scores and their corresponding ranks.

Example

Input

Scores table:

id	score
1	3.50
2	3.65

3	4.00
4	3.85
5	4.00
6	3.65

Output

score	rank
4.00	1
4.00	1
3.85	2
3.65	3
3.65	3
3.50	4

Solution:

To solve the problem of ranking scores with ties handled correctly, we will use the `DENSE_RANK()` window function. This function assigns ranks to rows within the partition of the result set without gaps in the ranking, which is exactly what we need for this problem.

Steps:

1. **Select the Scores Table**: We'll start by selecting all columns from the `Scores` table.
2. **Apply the `DENSE_RANK()` Window Function**: This will help us rank the scores from highest to lowest, assigning the same rank to tied scores.
3. **Order the Results by Score**: The final result should be ordered by the score in descending order.

Let's break down each step and construct the SQL query:

Step 1: Select the Scores Table

We first select the necessary columns from the `Scores` table. We will be working with the `id` and `score` columns.

```
SELECT
    id,
    score
FROM
    Scores
```

Step 2: Apply the `DENSE_RANK()` Window Function

We use the `DENSE_RANK()` function to rank the scores. This function is applied over the entire set of scores and ordered by score in descending order.

```
SELECT
    score,
    DENSE_RANK() OVER (ORDER BY score DESC) AS rank
FROM
    Scores
```

- **`DENSE_RANK() OVER (ORDER BY score DESC)`:** This orders the scores in descending order and assigns ranks such that tied scores get the same rank, and the next rank follows consecutively.

Step 3: Order the Results by Score

Finally, we order the results by the score in descending order to meet the requirement.

```
SELECT
    score,
    DENSE_RANK() OVER (ORDER BY score DESC) AS rank
FROM
    Scores
ORDER BY
    score DESC
```

Full SQL Query

```
SELECT
    score,
    DENSE_RANK() OVER (ORDER BY score DESC) AS rank
FROM
    Scores
ORDER BY
    score DESC;
```

Problem 2.6: [Easy] Duplicate Emails

Given a `Person` table with the following schema:

Column Name	Type
id	int
email	varchar

- `id` is the primary key for this table.
- Each row of this table contains an email. The emails will not contain uppercase letters.
- The email field is guaranteed not to be NULL.

Output: Return the result table with all duplicate emails. The result can be returned in any order.

Example

Input

Person table:

id	email
1	a@b.com
2	c@d.com
3	a@b.com

Output

email
a@b.com

Explanation

- The email `a@b.com` is repeated two times.

Solution:

Given a `Person` table with the columns `id` and `email`, we need to:

1. Identify emails that appear more than once in the table.
2. Return these duplicate emails in the result set.

Steps:

Step 1: Select the Emails and Count Occurrences

First, we need to count how many times each email appears in the `Person` table. We will use the `GROUP BY` clause to group the rows by the `email` column and the `COUNT(*)` function to count the occurrences of each email.

```
SELECT
    email,
    COUNT(*) AS email_count
FROM
    Person
GROUP BY
    email
```

- **`GROUP BY email`:** This groups the rows by the `email` column so that we can count the number of occurrences for each email.
- **`COUNT(*) AS email_count`:** This counts the number of rows in each group and gives us the count of each email.

Step 2: Filter Out Emails with Only One Occurrence

Next, we need to filter out emails that appear only once because we are only interested in duplicates. We use the `HAVING` clause to filter the results of the `GROUP BY` clause.

```
SELECT
    email
FROM
    Person
GROUP BY
    email
HAVING
    COUNT(*) > 1
```

- **`HAVING COUNT(*) > 1`:** This condition filters the groups to include only those with a count greater than 1, i.e., emails that appear more than once.

Step 3: Order the Results (Optional)

While the problem does not specify the need to order the results, you can optionally add an `ORDER BY` clause to sort the emails if desired.

```
SELECT
    email
FROM
    Person
GROUP BY
    email
HAVING
    COUNT(*) > 1
ORDER BY
    email;
```

Full SQL Query

```
SELECT
    email
FROM
    Person
GROUP BY
    email
HAVING
    COUNT(*) > 1;
```

Problem 2.7: [Medium] The Last Person to Fit in the Bus

Given a `Queue` table with the following schema:

Column Name	Type
person_id	int
person_name	varchar
weight	int
turn	int

- `person_id` contains unique values.
- This table contains information about all people waiting for a bus.
- The `person_id` and `turn` columns contain all numbers from 1 to n, where n is the number of rows in the table.
- `turn` determines the order in which people will board the bus, where `turn=1` denotes the first person to board and `turn=n` denotes the last person to board.
- `weight` is the weight of the person in kilograms.

There is a queue of people waiting to board a bus, but the bus has a weight limit of 1000 kilograms. Some people may not be able to board due to this limit.

Write a solution to find the `person_name` of the last person that can fit on the bus without exceeding the weight limit. The test cases are generated such that the first person does not exceed the weight limit.

Output: Return the result table with the name of the last person who can board the bus without exceeding the weight limit.

Example

Input

Queue table:

person_id	person_name	weight	turn
5	Alice	250	1
4	Bob	175	5
3	Alex	350	2
6	John Cena	400	3
1	Winston	500	6
2	Marie	200	4

Output

person_name
John Cena

Explanation

The following table is ordered by the turn for simplicity:

Turn	ID	Name	Weight	Total Weight	
1	5	Alice	250	250	
2	3	Alex	350	600	

3	6	John Cena	400	1000	(last person to board)
4	2	Marie	200	1200	(cannot board)
5	4	Bob	175	___	
6	1	Winston	500	___	

Solution:

Steps:

Step 1: Sort People by Their Turn

We need to select and sort the rows by the `turn` column, which determines the order in which people board the bus.

```
SELECT
    person_id,
    person_name,
    weight,
    turn
FROM
    Queue
ORDER BY
    turn;
```

Step 2: Accumulate Weights and Find the Last Person

We will use a running total to accumulate the weights as people board. We'll stop once adding another person's weight would exceed the 1000 kg limit.

To achieve this, we will use a common table expression (CTE) with a running total and a window function to accumulate weights.

```
WITH RunningTotal AS (
```

```
    SELECT
        person_name,
        weight,
        turn,
        SUM(weight) OVER (ORDER BY turn) AS total_weight
    FROM
        Queue
)
SELECT
    person_name
FROM
    RunningTotal
WHERE
    total_weight <= 1000
ORDER BY
    total_weight DESC
LIMIT 1;
```

- **`SUM(weight) OVER (ORDER BY turn) AS total_weight`**: This calculates the running total of weights as we go through the queue in the order of their turns.
- **`WHERE total_weight <= 1000`**: Filters the people who can board without exceeding the weight limit.
- **`ORDER BY total_weight DESC LIMIT 1`**: Retrieves the last person who fits within the weight limit.

Full SQL Query

```
WITH RunningTotal AS (
    SELECT
        person_name,
        weight,
        turn,
        SUM(weight) OVER (ORDER BY turn) AS total_weight
    FROM
        Queue
)
SELECT
    person_name
FROM
    RunningTotal
WHERE
    total_weight <= 1000
ORDER BY
    total_weight DESC
```

```
LIMIT 1;
```

Explanation:

1. **`WITH RunningTotal AS (...)`**:
    - Defines a common table expression (CTE) called `RunningTotal`.
    - This CTE computes the running total of weights as people board the bus in the order of their turns.
2. **`SELECT person_name, weight, turn, SUM(weight) OVER (ORDER BY turn) AS total_weight`**:
    - Selects the `person_name`, `weight`, and `turn` columns.
    - Calculates the cumulative sum of weights (`total_weight`) ordered by `turn`.
3. **`FROM Queue ORDER BY turn`**:
    - Specifies the `Queue` table as the source and orders the results by the `turn` column to ensure the correct boarding order.
4. **`SELECT person_name FROM RunningTotal WHERE total_weight <= 1000 ORDER BY total_weight DESC LIMIT 1`**:
    - Selects the `person_name` from the `RunningTotal` CTE where the `total_weight` is less than or equal to 1000.
    - Orders the filtered results by `total_weight` in descending order and limits the result to the first row, which gives us the last person who can board without exceeding the weight limit.

Problem 2.8: [Medium] Winning Candidate

Given the following tables:

Table: Candidate

Column Name	Type
id	int
name	varchar

- `id` is the column with unique values for this table.
- Each row contains information about the `id` and the `name` of a candidate.

Table: Vote

Column Name	Type
id	int
candidateId	int

- `id` is an auto-increment primary key.
- `candidateId` is a foreign key referencing the `id` from the `Candidate` table.
- Each row determines the candidate who got the ith vote in the elections.

Write a solution to report the name of the winning candidate (i.e., the candidate who received the largest number of votes).

Output: Return the result table with the name of the winning candidate.

Example

Input

Candidate table:

id	name
1	A
2	B
3	C
4	D
5	E

Vote table:

id	candidateId
1	2
2	4
3	3
4	2
5	5

Output

name
B

Explanation

- Candidate B has 2 votes.
- Candidates C, D, and E each have 1 vote.
- The winner is candidate B.

Solution:

To determine the winning candidate in the election, we need to find the candidate who received the largest number of votes. We will achieve this by following these steps:

1. **Count the votes for each candidate**: We will use the `Vote` table to count how many votes each candidate received.
2. **Determine the candidate with the maximum votes**: Once we have the vote counts, we will find the candidate with the highest count.
3. **Retrieve the candidate's name**: We will join the result with the `Candidate` table to get the name of the candidate with the most votes.

Step-by-Step Solution:

Step 1: Count the Votes for Each Candidate

We will use the `GROUP BY` clause to group the votes by `candidateId` and the `COUNT(*)` function to count the number of votes each candidate received.

```
SELECT
    candidateId,
    COUNT(*) AS vote_count
FROM
    Vote
GROUP BY
    candidateId
```

- **`GROUP BY candidateId`**: This groups the rows by `candidateId` so that we can count the number of votes for each candidate.
- **`COUNT(*) AS vote_count`**: This counts the number of votes for each candidate.

Step 2: Determine the Candidate with the Maximum Votes

Next, we need to find the candidate who received the highest number of votes. We can use a subquery to find the maximum vote count and filter the candidates accordingly.

```
WITH VoteCounts AS (
    SELECT
        candidateId,
        COUNT(*) AS vote_count
    FROM
        Vote
    GROUP BY
        candidateId
)
SELECT
    candidateId
FROM
    VoteCounts
WHERE
    vote_count = (SELECT MAX(vote_count) FROM VoteCounts)
```

- **`WITH VoteCounts AS (...)`**: This common table expression (CTE) calculates the vote count for each candidate.
- **`WHERE vote_count = (SELECT MAX(vote_count) FROM VoteCounts)`**: This filters the candidates to find the one with the maximum vote count.

Step 3: Retrieve the Candidate's Name

Finally, we need to join the result with the `Candidate` table to get the name of the candidate with the most votes.

```
WITH VoteCounts AS (
    SELECT
        candidateId,
        COUNT(*) AS vote_count
    FROM
        Vote
    GROUP BY
        candidateId
)
SELECT
    c.name
FROM
    VoteCounts vc
JOIN
    Candidate c ON vc.candidateId = c.id
```

```
WHERE
    vc.vote_count = (SELECT MAX(vote_count) FROM VoteCounts);
```

- **`JOIN Candidate c ON vc.candidateId = c.id`**: This joins the `VoteCounts` CTE with the `Candidate` table to get the candidate's name.

Full SQL Query

```
WITH VoteCounts AS (
    SELECT
        candidateId,
        COUNT(*) AS vote_count
    FROM
        Vote
    GROUP BY
        candidateId
)
SELECT
    c.name
FROM
    VoteCounts vc
JOIN
    Candidate c ON vc.candidateId = c.id
WHERE
    vc.vote_count = (SELECT MAX(vote_count) FROM VoteCounts);
```

Summary:

1. **`WITH VoteCounts AS (...)`**:
    - Defines a CTE named `VoteCounts` that calculates the number of votes each candidate received.
    - Groups the votes by `candidateId` and counts them.
2. **`SELECT candidateId, COUNT(*) AS vote_count FROM Vote GROUP BY candidateId`**:
    - This part of the query inside the CTE calculates the vote counts.
3. **`SELECT c.name FROM VoteCounts vc JOIN Candidate c ON vc.candidateId = c.id`**:
    - This joins the `VoteCounts` CTE with the `Candidate` table to get the names of the candidates.
4. **`WHERE vc.vote_count = (SELECT MAX(vote_count) FROM VoteCounts)`**:
    - This filters the result to include only the candidate(s) with the highest vote count.

AI/ML

Problem 3.1: What is Principal Component Analysis (PCA)? When do you use it?

Answer::

Principal Component Analysis (PCA) is a technique used in data analysis to simplify complex datasets. If you have a big table of data with lots of features, like height, weight, age, and more for a group of people. Sometimes, having too many columns makes it hard to see patterns or relationships in the data.

PCA helps by transforming this large set of features into a smaller one while keeping as much important information as possible. Here's how it works in simple terms:

1. **Identify Patterns:** PCA looks at the data and identifies the directions (called "principal components") in which the data varies the most.
2. **Create New Features:** These principal components are new features created by combining the original ones in a specific way. Each new feature captures a significant part of the data's variation.
3. **Reduce Dimensions:** Often, the first few principal components capture most of the important information. So, instead of using all the original features, you can use these few new features. This reduces the number of columns while keeping the essence of the data.

For example, if you had data with 10 columns, PCA might tell you that just 2 or 3 new columns (principal components) can represent the data almost as well as the original 10.

Why Use PCA?

- **Simplification:** It makes large datasets easier to understand and work with.
- **Visualization:** With fewer dimensions, you can create simpler visualizations to see patterns and relationships.
- **Noise Reduction:** By focusing on the main components, you can reduce the impact of noisy or less important data.

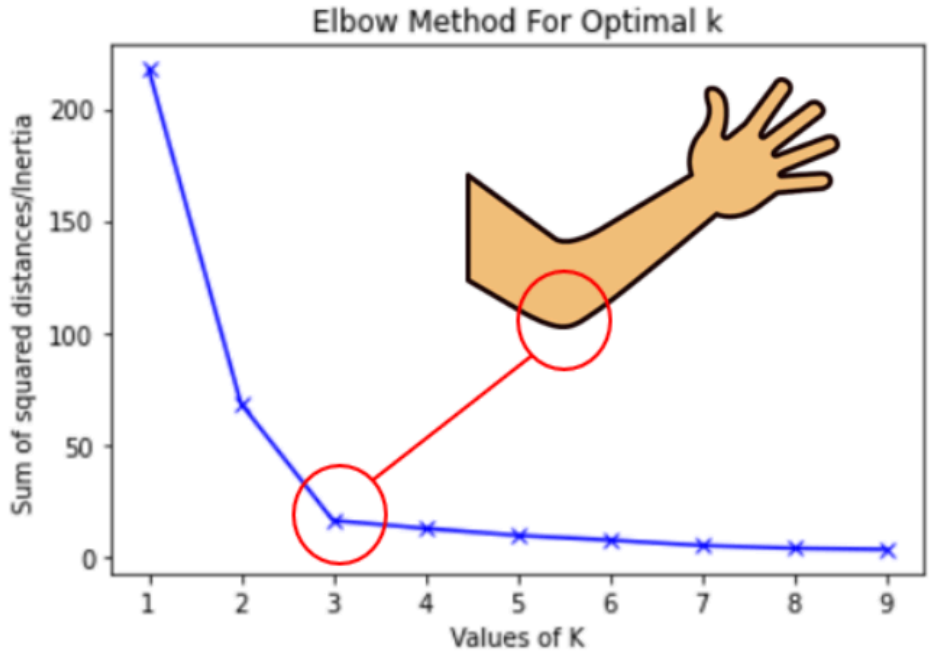
Problem 3.2: How do you determine the optimal number of clusters (k) in the K-means algorithm?

Answer:

Question: How do you determine the optimal number of clusters (k) in the K-means algorithm?

Answer:

1. **Elbow Method:**
    - **How It Works:**
        - Run the K-means algorithm for different values of k (e.g., from 1 to 10).
        - For each k, calculate the sum of squared distances between data points and their corresponding cluster centroids (called the within-cluster sum of squares or WCSS).
        - Plot these WCSS values against the number of clusters.
    - **Interpretation:**
        - Look for a "bend" or "elbow" in the plot where the WCSS starts to decrease more slowly. This point indicates that adding more clusters beyond this value of k doesn't provide a significant improvement in explaining the data's variance.
        - The optimal k is at this elbow point.

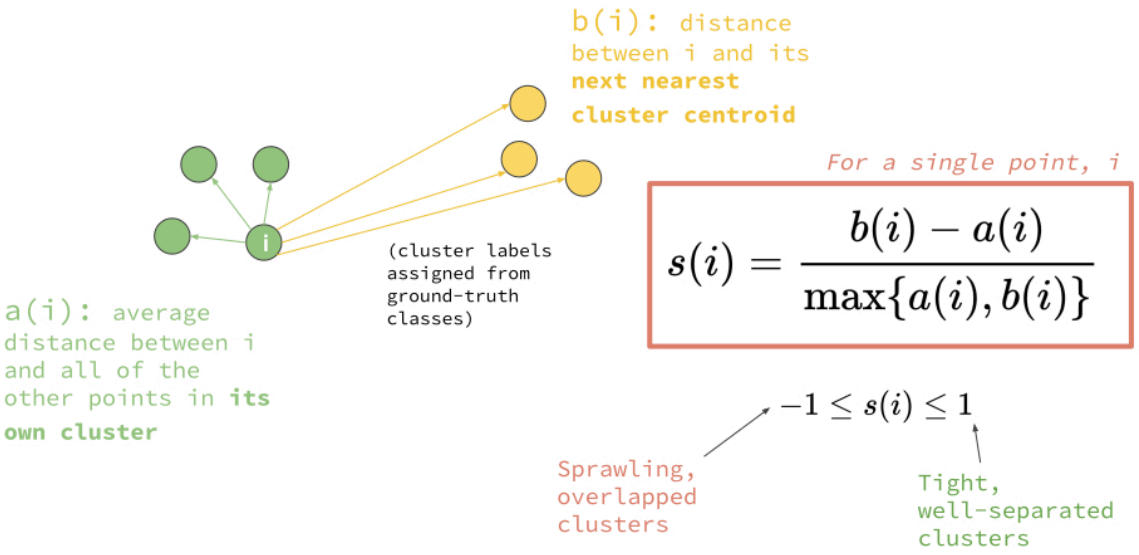


Line plot between K and inertia

2. **Silhouette Method:**
    - **How It Works:**
        - Run the K-means algorithm for different values of k.

- Calculate the silhouette score for each k, which measures how similar each point is to its own cluster compared to other clusters. The silhouette score ranges from -1 to 1, where higher values indicate better clustering.

- **Interpretation:**
  - Plot the silhouette score against the number of clusters.
  - The optimal k is the value with the highest silhouette score, indicating the best-defined clusters.



3. **Cross-Validation:**
   - **How It Works:**
     - Use techniques like cross-validation to evaluate the performance of K-means for different values of k.
     - This involves splitting the data into training and validation sets and assessing the clustering quality on the validation set.
   - **Interpretation:**
     - Choose the k that provides the best performance on the validation set, balancing good cluster separation with the model's generalization ability.

Problem 3.3: What is regularization in linear models, and why is it important? Can you explain the common types of regularization techniques used in linear models?

Solution:

Regularization is a technique used to prevent overfitting, which occurs when a model learns the noise in the training data instead of the underlying pattern. Overfitting leads to poor generalization to new, unseen data. Regularization introduces a penalty for larger coefficients in the model, which helps to keep the model simpler and more generalizable.

Why is Regularization Important?

- Prevents Overfitting: By penalizing large coefficients, regularization ensures the model doesn't fit the noise in the training data.
- Improves Generalization: A simpler model with smaller coefficients is more likely to perform well on new, unseen data.
- Enhances Stability: Regularized models are less sensitive to small changes in the training data, making them more robust.

Common Types of Regularization Techniques:

1. L1 Regularization (Lasso Regression):

$$Cost\ Function = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x)^i - y^i\right)^2 + \lambda \sum_{i=1}^{n} |slope|$$

$$\lambda = Hyperparameter$$

Cost Function Lasso Regression

   - Lambda is the regularization parameter that controls the strength of the penalty.
   - Effect:
     - Encourages sparsity in the model, meaning it can reduce some coefficients to exactly zero, effectively performing feature selection.
2. L2 Regularization (Ridge Regression):

$$Cost\ Function\ = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta(x)^i - y^i\right)^2 + \lambda\ (slope)^2$$

$$\lambda = Hyperparameter$$

Cost Function Ridge Regression

- Adds a penalty equal to the square of the coefficients to the loss function.
- Effect:
  - Shrinks the coefficients but does not force them to be zero. This means all features are kept in the model but with smaller magnitudes.

3. Elastic Net Regularization:
   - Combines both L1 and L2 regularization penalties.
   - Effect:
     - Balances the benefits of both Lasso and Ridge regression, encouraging sparsity while also shrinking coefficients.

Implementation: You are building a linear regression model to predict house prices based on features like size, number of rooms, and location. Without regularization, the model might overfit, capturing noise from features that aren't very predictive.

1. Using Lasso Regression:
   - The model might set the coefficient for an irrelevant feature (like the color of the front door) to zero, effectively removing it from the model.
2. Using Ridge Regression:
   - The model would shrink the coefficients for less important features, but none would be eliminated.
3. Using Elastic Net:
   - The model would achieve a balance, possibly reducing some coefficients to zero while shrinking others.

Problem 3.4: Can you explain what the F1 score is and why it is important in evaluating the performance of a classification model?

Solution:

The F1 score is a metric used to evaluate the performance of a classification model, particularly when dealing with imbalanced datasets. It combines precision and recall into a single score, providing a balance between the two.

Why is the F1 Score Important?

- **Balances Precision and Recall:** It gives a single metric that considers both false positives (errors where the model incorrectly predicts the positive class) and false negatives (errors where the model fails to predict the positive class).
- **Useful for Imbalanced Data:** When the dataset has an unequal distribution of classes (e.g., more negative instances than positive ones), accuracy can be misleading. The F1 score provides a more informative measure in such cases.
- **Better Evaluation:** It helps in scenarios where you want to ensure that both precision and recall are reasonably high, which is crucial in many real-world applications.

Components of the F1 Score:

1. **Precision:**
    - Precision is the ratio of correctly predicted positive observations to the total predicted positives.

$$\frac{TP}{TP + FP}$$

    -
    - Here, TP is True Positives, and FP is False Positives.
2. **Recall (Sensitivity or True Positive Rate):**
    - Recall is the ratio of correctly predicted positive observations to all observations in the actual positive class.

$$\frac{TP}{TP + FN}$$

- Here, FN is False Negatives.

3. **F1 Score:**
   - The F1 score is the harmonic mean of precision and recall.

$$\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

   - The harmonic mean penalizes extreme values more than the arithmetic mean, ensuring that both precision and recall are reasonably high.

Example: You have a model that predicts whether an email is spam (positive class) or not spam (negative class). Here's how the F1 score helps:

1. **Precision:**
   - Your model predicts 100 emails as spam, but only 80 of them are actually spam. So, precision = 80/100 = 0.8 or 80%.
2. **Recall:**
   - Out of 90 actual spam emails, your model correctly identifies 80. So, recall = 80/90 = 0.89 or 89%.
3. **F1 Score:**
   - The F1 score combines these two measures:

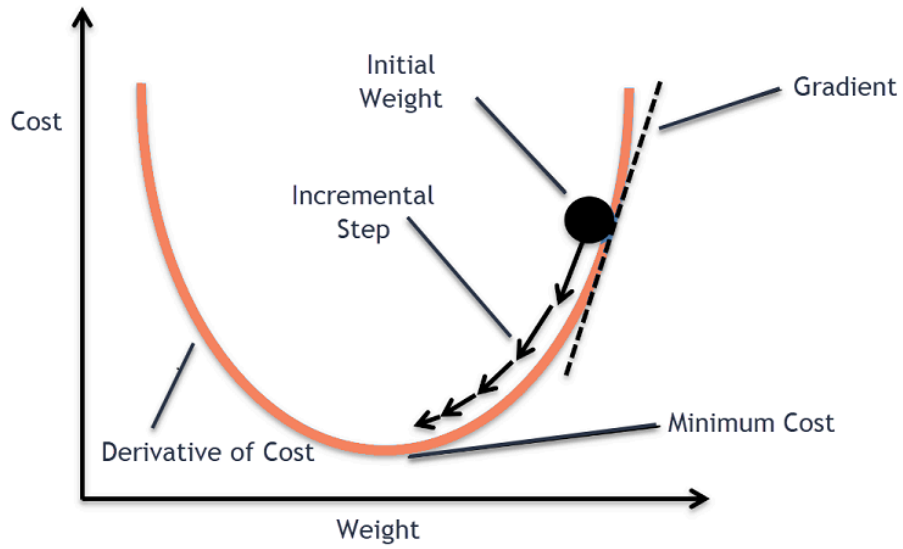
$$2 \times \frac{0.8 \times 0.89}{0.8 + 0.89} \approx 0.84$$

Problem 3.5: Can you explain what gradient descent is and how it is used in training machine learning models?

Solution:

Gradient descent is an optimization algorithm used to minimize the loss function in machine learning models. It is a fundamental technique for training models, especially those involving neural networks and linear regression.

How Gradient Descent Works:



1. **Initialize Parameters:**
   - Start with initial values for the model parameters (weights and biases). These can be set to zero, random values, or other starting points.
2. **Calculate the Gradient:**
   - Compute the gradient of the loss function with respect to each parameter. The gradient is a vector of partial derivatives indicating the direction and rate of the steepest increase in the loss function.
3. **Update Parameters:**
   - Adjust the parameters in the direction opposite to the gradient to decrease the loss function. This step is controlled by the learning rate, which determines the size of the step.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

   -
     - θ represents the parameters.
     - α is the learning rate.

        - $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the parameters.
4. **Iterate:**
    - Repeat the process of calculating the gradient and updating the parameters until the loss function converges to a minimum value or a predefined number of iterations is reached.

Types of Gradient Descent:

1. **Batch Gradient Descent:**
    - Uses the entire dataset to compute the gradient in each iteration.
    - Pros: Accurate gradient estimates.
    - Cons: Computationally expensive and slow for large datasets.
2. **Stochastic Gradient Descent (SGD):**
    - Uses one data point at a time to compute the gradient and update the parameters.
    - Pros: Faster and more efficient for large datasets.
    - Cons: Noisy updates can cause the loss function to fluctuate.
3. **Mini-Batch Gradient Descent:**
    - Uses a small subset (mini-batch) of the dataset to compute the gradient.
    - Pros: Balance between the efficiency of SGD and the accuracy of batch gradient descent.
    - Cons: Requires selecting an appropriate mini-batch size.

Example: Imagine you're training a linear regression model to predict house prices based on features like size and number of rooms. The loss function (e.g., Mean Squared Error) measures how well the model's predictions match the actual prices.

1. **Initialize Parameters:**
    - Set initial weights and biases.
2. **Calculate the Gradient:**
    - Compute the gradient of the loss function with respect to the weights and biases.
3. **Update Parameters:**
    - Adjust the weights and biases by taking a step in the opposite direction of the gradient.
4. **Iterate:**
    - Repeat the gradient calculation and parameter update until the loss function reaches a minimum.

Problem 3.6: What is overfitting in machine learning, and how can it be prevented?

Overfitting in machine learning occurs when a model learns the training data too well, including its noise and outliers. As a result, the model performs excellently on the training data but poorly on new, unseen data because it fails to generalize.

Why Overfitting Happens:

- **Complex Models:** Models with too many parameters relative to the number of training samples can fit the training data perfectly but fail to generalize.
- **Insufficient Training Data:** With too little data, the model might capture noise as patterns.
- **Noisy Data:** Data with lots of noise can lead the model to learn irrelevant details.

How to Identify Overfitting:

- **High Training Accuracy but Low Test Accuracy:** The model performs well on training data but poorly on validation or test data.
- **Complexity of the Model:** The model has too many parameters compared to the amount of training data.

Techniques to Prevent Overfitting:

1. **Cross-Validation:**
    - Use techniques like k-fold cross-validation to ensure the model performs well on different subsets of the data.
    - This helps in evaluating the model's performance on multiple splits, reducing the risk of overfitting.
2. **Regularization:**
    - Apply regularization techniques like L1 (Lasso) and L2 (Ridge) to penalize large coefficients, encouraging simpler models.
    - Regularization adds a penalty term to the loss function, discouraging overly complex models.
3. **Simplifying the Model:**
    - Reduce the complexity of the model by decreasing the number of parameters or layers (in the case of neural networks).
    - Prune unnecessary features or use feature selection techniques to retain only the most important features.
4. **Data Augmentation:**
    - Increase the size of the training dataset by augmenting the data (e.g., rotating, flipping, or scaling images in image recognition tasks).
    - This helps the model learn better and generalize well to new data.
5. **Early Stopping:**
    - Monitor the model’s performance on a validation set during training and stop training when the performance starts to deteriorate.
    - This prevents the model from continuing to learn the noise in the training data.

6. **Ensembling:**
    - Combine predictions from multiple models using techniques like bagging, boosting, or stacking.
    - Ensembling helps in reducing the variance and improving the model's generalization ability.

Math

Problem 4.1: What is a p-value in statistical hypothesis testing, and how is it interpreted?

Solution:

A p-value is a measure used in statistical hypothesis testing to determine the significance of the results. It helps you decide whether to reject the null hypothesis, which is the default assumption that there is no effect or no difference.

Understanding the p-value:

- **Definition:** The p-value is the probability of obtaining test results at least as extreme as the observed results, assuming that the null hypothesis is true.
- **Range:** The p-value ranges from 0 to 1. A lower p-value indicates that the observed data is less likely to occur under the null hypothesis.

Interpreting the p-value:

1. **Threshold for Significance (Alpha Level):**
    - Before conducting a test, you choose a significance level (alpha), commonly set at 0.05.
    - This alpha level represents the threshold for deciding whether the p-value is sufficiently low to reject the null hypothesis.
2. **Decision Rule:**
    - If the p-value is less than or equal to the alpha level (e.g., 0.05), you reject the null hypothesis.
    - If the p-value is greater than the alpha level, you fail to reject the null hypothesis.

Example: Suppose you're testing a new drug and want to determine if it is more effective than an existing drug. The null hypothesis (H0) states that there is no difference in effectiveness between the two drugs.

1. **Conduct the Test:**
    - Collect data and perform a statistical test (e.g., a t-test) to compare the effectiveness of the two drugs.
2. **Calculate the p-value:**
    - Let's say the p-value calculated from the test is 0.03.
3. **Compare with Alpha Level:**
    - If your chosen alpha level is 0.05, then $0.03 < 0.05$.
4. **Make a Decision:**
    - Since the p-value is less than the alpha level, you reject the null hypothesis and conclude that there is a statistically significant difference in effectiveness between the new drug and the existing drug.

Key Points to Remember:

- **Significance Level (Alpha):** The chosen threshold (e.g., 0.05) against which the p-value is compared.
- **Statistical Significance:** A low p-value (below the alpha level) indicates strong evidence against the null hypothesis, suggesting that the observed effect is unlikely to be due to chance.
- **Not Proof of Effect:** A small p-value does not prove that the null hypothesis is false or that the observed effect is practically significant; it only suggests that the data is inconsistent with the null hypothesis.

Problem 4.2: What is the Central Limit Theorem (CLT) in statistics, and why is it important?

Solution:

The Central Limit Theorem (CLT) is a fundamental principle in statistics that describes the distribution of the sample mean of a large number of independent, identically distributed random variables. It is crucial because it allows statisticians to make inferences about population parameters even when the population distribution is unknown.

Understanding the Central Limit Theorem:

1. **Statement of the CLT:**
    - The CLT states that, given a sufficiently large sample size, the distribution of the sample mean will approach a normal (Gaussian) distribution, regardless of the original distribution of the population.
    - Mathematically, if $X_1, X_2, \ldots, X_n$ are independent and identically distributed random variables with mean μ and variance σ^2, then the sample mean $\bar{X}$ of these variables will be approximately normally distributed with mean μ and variance $(\sigma^2)/n$ as the sample size (n) becomes large.
2. **Key Concepts:**
    - **Sample Mean ($\bar{X}$):** The average of the sample observations.
    - **Normal Distribution:** A symmetric, bell-shaped distribution characterized by its mean and standard deviation.
    - **Sufficiently Large Sample Size:** Typically, a sample size of 30 or more is considered large enough for the CLT to hold.

Importance of the CLT:

- **Foundation for Inference:** The CLT is the basis for many statistical methods, including confidence intervals and hypothesis tests, because it allows us to use the normal distribution to approximate the sampling distribution of the mean.
- **Simplifies Analysis:** It simplifies the analysis of complex data by ensuring that the sample mean follows a normal distribution, even if the data itself does not.
- **Robustness:** The CLT applies to a wide variety of distributions, making it a powerful and versatile tool in statistics.

Implications of the CLT:

1. **Normal Approximation:** For large sample sizes, the sample mean will be approximately normally distributed. This means we can use properties of the normal distribution, such as standard z-scores, to make inferences about the population mean.
2. **Error Reduction:** As the sample size increases, the standard error of the mean decreases, leading to more precise estimates of the population mean.
3. **Confidence Intervals:** The CLT allows us to construct confidence intervals for the population mean using the sample mean and standard error.

4. **Hypothesis Testing:** It enables us to perform hypothesis tests on the population mean, comparing the sample mean to a hypothesized value using the normal distribution.

Problem 4.3: What is the difference between descriptive vs inferential statistics?

Solution:

Descriptive Statistics:

- Summarize and describe the main features of a dataset.
- Mean (average), median (middle value), mode (most frequent value), standard deviation (measure of spread).
- Use Case: If you have test scores for a class, you might use the average score to describe how the class performed overall.

Inferential Statistics:

- Make predictions or inferences about a population based on a sample of data.
- Confidence intervals, hypothesis tests.
- Use Case: If you want to know the average height of all students in a school, you might measure the height of a sample of students and use inferential statistics to estimate the average height for the entire school.

Key Difference:

- Descriptive statistics describe the data you have.
- Inferential statistics use that data to make predictions or generalizations about a larger group.