2806. Account Balance After Rounded Purchase

Easy Math

Problem Description

The given problem scenario describes a situation where an individual has an initial bank account balance of $100. The individual intends to make a purchase with an amount specified by `purchaseAmount`. However, there is a unique rule at the store where the purchase is made; the actual amount paid, `roundedAmount`, must be a multiple of 10. To determine this amount, we find the nearest multiple of 10 to the `purchaseAmount`. If there is a tie between two multiples of 10 (meaning that the purchase amount is exactly halfway between them), we choose the larger multiple. The objective is to find out what the account balance will be after making a purchase according to these rules.

The task involves calculating the `roundedAmount` by finding the closest multiple of 10 to `purchaseAmount`. After the `roundedAmount` is determined, it is subtracted from the initial balance to obtain the final account balance, which is returned by the function.

Intuition

The solution aims to find the nearest multiple of 10 to the `purchaseAmount` such that the difference $|\texttt{roundedAmount} - \texttt{purchaseAmount}|$ is minimized. If there are two equally close multiples, we select the larger one.

To arrive at the solution, we can iterate backward from the initial balance of $100 in decrements of 10, which are the potential candidates for `roundedAmount`. As we do this, we calculate the absolute difference between each candidate and the `purchaseAmount`. The candidate with the smallest difference is the `roundedAmount` we want to find. If there is a tie, the loop iterating in reverse ensures we choose the larger multiple.

By keeping track of the smallest difference seen so far and the associated candidate (`roundedAmount`), we can decide which multiple of 10 to select. Once we determine the `roundedAmount`, the final balance is obtained simply by subtracting this value from the initial balance of $100. The resulting balance after the purchase is what the function returns.

Solution Approach

The implementation of the solution adopts a straightforward approach to solve the problem. The algorithm does not make use of any complex data structures or algorithms, such as dynamic programming or memoization. Instead, it relies on simple iteration and comparison.

Here's a step-by-step breakdown of the solution process:

1. Initialize two variables: `diff`, set to a high value (in this case, 100, as no absolute difference can exceed the initial balance), and `x`, which will represent our `roundedAmount`.
2. Iterate backward from the initial balance of $100 to 0 in decrements of 10. Each of these numbers represents a potential `roundedAmount`.
3. In each iteration, calculate the temporary difference `t` between the current multiple of 10 (`y`) and the `purchaseAmount`. This is done using the absolute difference function `abs(y - purchaseAmount)`.
4. Compare this temporary difference `t` with the smallest difference found so far (`diff`). If `t` is less than `diff`, it means we have found a closer multiple of 10 to `purchaseAmount`. Update `diff` to this new minimum difference and `x` to the current multiple of 10 (`y`).
5. After the loop ends, `x` holds the value of the nearest (largest in the case of a tie) multiple of 10 to `purchaseAmount`. The final account balance is calculated by subtracting `x` from the initial balance of $100.
6. Return the final account balance.

The algorithm uses a for-loop to execute the steps mentioned above. The tuple unpacking in `if (t := abs(y - purchaseAmount))` is a Python 3.8 feature called the "walrus operator" (`:=`), which assigns values to variables as part of a larger expression.

Here, `abs(y - purchaseAmount)` is simultaneously assigned to `t` and then compared against `diff`. This helps in writing more compact and readable code. No additional data structures are needed since the problem can be solved by simply tracking the difference and the corresponding multiple of 10.

Example Walkthrough

Let's assume `purchaseAmount` is $74. The goal is to round this amount to the nearest multiple of 10, which can be either 70 or 80, and then subtract it from the initial balance of 100.

We start with the initial `diff` set to a high value, which here is 100. The `roundedAmount` (`x`) is what we're looking to find.

1. We'll start checking from 100 downwards in steps of 10 (100, 90, 80, and so on) until we reach 0. These represent potential `roundedAmount` values.
2. When we reach $80 (y = 80), we calculate the difference: `t = abs(y - purchaseAmount) = abs(80 - 74) = 6`.

   Since `6` is less than our initial `diff` (100), we update `diff` to `6` and `x` to `80`.
3. Continuing the iteration, we check the next multiple of 10, which is $70: `t = abs(y - purchaseAmount) = abs(70 - 74) = 4`.

   Now `4` is less than our current `diff` (`6`), so we update `diff` to `4` and `x` to `70`.
4. We proceed with the loop, but since all subsequent multiples of 10 will increase the difference (moving further away from $74), there will be no updates to `diff` or `x`.
5. After completing the iterations, the nearest multiple of 10 is 70 (which is `x`), and the difference we've settled on is 4 (which is the final `diff`).
6. The final account balance can now be calculated by subtracting `x` from the initial balance: `finalBalance = initialBalance - x = 100 - 70 = 30`.

Therefore, after the purchase with the `purchaseAmount` of 74 that gets rounded to 70, the final account balance would be $30.

Solution Implementation

Python

```python
class Solution:
    def accountBalanceAfterPurchase(self, purchase_amount: int) -> int:
        # Initialize the closest difference to the maximum possible value (100)
        # and the closest rounded amount to 0
        closest_difference = 100
        closest_rounded_amount = 0

        # Iterate backward from 100 to 0 with a step of -10
        for rounded_amount in range(100, -1, -10):
            # Calculate the absolute difference between the purchase amount and
            # the current rounded amount
            current_difference = abs(rounded_amount - purchase_amount)

            # Determine if the current difference is smaller than the closest
            # difference we have found so far
            if current_difference < closest_difference:
                # If so, update the closest difference and the corresponding
                # rounded amount
                closest_difference = current_difference
                closest_rounded_amount = rounded_amount

        # Return the adjusted account balance after the purchase, which is
        # 100 subtracted by the closest rounded amount
        return 100 - closest_rounded_amount
```

Java

```java
class Solution {

    // This method calculates the account balance after a purchase with an initial balance of 100.
    // It finds the closest decrement of 10 from the purchase amount and subtracts it from 100.
    public int accountBalanceAfterPurchase(int purchaseAmount) {
        // Initialize the minimum difference found to 100 (which can be the max difference as per the logic below)
        int minDifference = 100;
        // This will hold the closest matching decrement value
        int closestMatch = 0;

        // Loop through decrements of 10 starting from 100 to 0
        for (int currentDecrement = 100; currentDecrement >= 0; currentDecrement -= 10) {
            // Calculate the absolute difference between the purchase amount and the current decrement
            int currentDifference = Math.abs(currentDecrement - purchaseAmount);
            // If the current difference is smaller than any previously recorded difference
            if (currentDifference < minDifference) {
                // Update the minimum difference
                minDifference = currentDifference;
                // Update the closest matching decrement which we might subtract from the balance
                closestMatch = currentDecrement;
            }
        }
        // Return the balance after subtracting the closest matching decrement
        return 100 - closestMatch;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate account balance after a purchase is made.
    // The function finds the nearest multiple of 10 to the purchase amount
    // and subtracts it from the starting balance, which is assumed to be 100.
    int accountBalanceAfterPurchase(int purchaseAmount) {
        int minDifference = 100;    // Initialize the minimum difference to the highest value possible (100).
        int closestMultiple = 0;    // This will hold the closest multiple of 10 to purchaseAmount.

        // Iterate over possible multiples of 10, from 100 down to 0, decrementing by 10.
        for (int currentMultiple = 100; currentMultiple >= 0; currentMultiple -= 10) {
            // Calculate the absolute difference between currentMultiple and purchaseAmount.
            int currentDifference = abs(currentMultiple - purchaseAmount);

            // If the current difference is less than the minimum difference found so far,
            // update minDifference and closestMultiple.
            if (currentDifference < minDifference) {
                minDifference = currentDifference;
                closestMultiple = currentMultiple;
            }
        }

        // The new balance is the original balance (100) minus the closest multiple of 10 to purchaseAmount.
        return 100 - closestMultiple;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the new account balance after a purchase is made.
 * This function assumes the account starts with a balance of 100,
 * and subtracts the nearest multiple of 10 to the purchase amount from it.
 *
 * @param {number} purchaseAmount - The amount of the purchase made.
 * @return {number} The account balance after the purchase.
 */
function accountBalanceAfterPurchase(purchaseAmount: number): number {
    // Initialize the closest difference to the purchase amount and its multiple of 10.
    let closestDifference: number = 100;
    let closestMultiple: number = 0;

    // Iterate through multiples of 10 from 100 down to 0.
    for (let multiple = 100; multiple >= 0; multiple -= 10) {
        // Calculate the absolute difference from the current multiple to the purchase amount.
        const currentDifference: number = Math.abs(multiple - purchaseAmount);

        // If the current difference is smaller than the closest one, update the closest values.
        if (currentDifference < closestDifference) {
            closestDifference = currentDifference;
            closestMultiple = multiple;
        }
    }

    // Return the difference between the initial balance and the closest multiple of 10 found.
    return 100 - closestMultiple;
}
```

```python
class Solution:
    def accountBalanceAfterPurchase(self, purchase_amount: int) -> int:
        # Initialize the closest difference to the maximum possible value (100)
        # and the closest rounded amount to 0
        closest_difference = 100
        closest_rounded_amount = 0

        # Iterate backward from 100 to 0 with a step of -10
        for rounded_amount in range(100, -1, -10):
            # Calculate the absolute difference between the purchase amount and
            # the current rounded amount
            current_difference = abs(rounded_amount - purchase_amount)

            # Determine if the current difference is smaller than the closest
            # difference we have found so far
            if current_difference < closest_difference:
                # If so, update the closest difference and the corresponding
                # rounded amount
                closest_difference = current_difference
                closest_rounded_amount = rounded_amount

        # Return the adjusted account balance after the purchase, which is
        # 100 subtracted by the closest rounded amount
        return 100 - closest_rounded_amount
```

Time and Space Complexity

The time complexity of the given code snippet is $O(1)$ because the code contains a single for-loop that always iterates a constant number of times (10 iterations exactly, from 100 down to 0 in steps of 10).

The space complexity of the code is also $O(1)$ because the code only uses a fixed amount of additional space regardless of the input size. The variables `diff`, `x`, and `t` are the only extra variables that are used, and they do not depend on the size of the input.