2162. Minimum Cost to Set Cooking Time

Medium Math Enumeration Leetcode Link

Problem Description

In this problem, we are tasked with calculating the minimum fatigue cost required to set a microwave to cook for a target duration in seconds. The microwave interface takes up to four digits input, interpreted as minutes and seconds, with the first two digits representing minutes (00 to 99) and the last two representing seconds (00 to 99). To set the microwave, we prepend zeroes to make sure we always have a four-digit number if fewer digits are entered.

The microwave setting process involves moving a finger from the starting digit (given by `startAt`) to other digits and pushing them to set the time. Each move incurs a `moveCost` and each push incurs a `pushCost`. There might be multiple ways to set the desired cooking time, but we are interested in finding the way that results in the least total fatigue cost.

Finally, the target cooking time is defined in seconds (`targetSeconds`). Since the microwave interprets inputs as minutes and seconds, we must convert the target time accordingly. One minute is equivalent to 60 seconds, so the target time could sometimes be set more efficiently by adjusting the balance between minutes and seconds.

Intuition

The intuition behind the solution is to break down the problem into simpler, logical steps and use a helper function to calculate the cost of setting the microwave to a given time.

1. Convert the `targetSeconds` to minutes and seconds. Since the maximum number of seconds the microwave can display is 99, we first divide the `targetSeconds` by 60 to find the number of whole minutes, and then we use modulo to find the remaining seconds.
2. We create a helper function, `f(m, s)`, that calculates the cost of setting the microwave to `m` minutes and `s` seconds. It accounts for the cost of moving to and pushing the required digits. If the minutes or seconds exceed their respective bounds (e.g., more than 99), the function returns an infinite cost, indicating an invalid input.
3. The function first converts the minutes and seconds into an array of four digits, including leading zeros. It then ignores any leading zeros because they do not require any push.
4. It iterates through the array of digits, calculating the cost to move to and push each digit. If the current digit is the same as the previous one, we only consider the push cost; otherwise, we include both the move and push costs.
5. To find the minimum cost, we compare the cost of setting the exact minutes and seconds derived from `targetSeconds` and also consider the scenario where we decrement one minute and add 60 seconds (because sometimes it's cheaper to enter fewer minutes and more seconds, especially if it allows us to reduce the number of moves or pushes).
6. Finally, we return the minimum cost obtained from the two possibilities calculated using our helper function.

Using these steps, the solution efficiently examines the possible ways to enter the desired cooking time and selects the one with the lowest physical cost.

Solution Approach

The solution is implemented in Python and follows a step-by-step logical approach to minimize fatigue when setting the microwave timer. Let's walk through the implementation strategy:

- **Defining The Helper Function (`f`)**: The heart of this solution is the helper function `f(m, s)`, which calculates the cost for a proposed minute (`m`) and second (`s`) combination. It assumes that non-valid minute or second values (like more than 99 minutes or seconds) return an infinite cost denoted by `inf`. This prevents the use of invalid time formats.
- **Converting Time Into Digits**: Inside the helper function, we transform `m` and `s` into an array of individual digits, starting from the most significant to the least significant. This is done by integer division and modulo to split each one into tens and ones.
- **Cost Calculation Logic**: After getting the digit array, we iterate from the start (ignoring leading zeros) and calculate the cost as follows:
  - If the current digit is different from the previous one (`prev`), we have to move the finger, which incurs a `moveCost`.
  - Regardless of whether we moved or not, we always incur a `pushCost` when pressing a digit. The `prev` variable is then updated to the current digit for the next iteration.
- **Finding The Cost For Two Scenarios**: We call the helper function `f` twice with:
  - The exact minutes (`m`) and seconds (`s`) calculated from `targetSeconds`.
  - The adjusted values where we subtract one minute and add 60 seconds to `s` (only if the `m` is greater than 0 since we can't have negative minutes).
- **Algorithm Pattern**: The pattern employed by this solution is essentially a brute-force check of the possible valid combinations (there are only two), and simply choosing the one that returns the least cost. This is quick due to the low number of total possible and valid combinations.
- **Data Structures**: This solution relies on simple data structures: primarily an integer array to keep track of the digit-wise break down of the minute-second format required for setting the timer.

Example Scenario Execution: If `targetSeconds = 5500`, this translates to `91` minutes and `40` seconds, but since the microwave cannot display more than `99` seconds, we have to subtract one minute and add those 60 seconds making the time `90` minutes and `100` seconds. Then, we sequentially apply the helper function to both time formats and choose the least costly one.

```
// Insert calculated cost calculation steps here
```

From the implementation given; it is clear that the approach taken to solve the problem is both efficient and effective. It cleverly utilizes the simplicity of brute force in a scenario where analyzing all combinations is trivial, combined with a well-defined helper function to abstract out the cost-calculation logic.

Example Walkthrough

Let's go through an example to illustrate the solution approach. Suppose that `targetSeconds = 130`, `startAt = 1`, `moveCost = 2`, and `pushCost = 1`. We want to find the minimum fatigue cost to set the microwave to cook for `130` seconds.

1. **Convert `targetSeconds` to minutes and seconds.**
   `130` seconds is equal to `2` minutes and `10` seconds since (130 = 2 \times 60 + 10). So our inputs to the helper function will be `m = 2` and `s = 10`.
2. **Use Helper Function `f(m, s)`.**
   Implement `f(2, 10)`:
   - Convert to array of digits: `m = 2` becomes `02` and `s = 10` becomes `10` for a complete array `[0, 2, 1, 0]`.
   - Start at `1`, move to `0` (start digit does not require push), move cost is `2`.
   - Push `0`, push cost is `1` (though first `0` is not pushed as it's leading).
   - Move to `2`, move cost is `2`.
   - Push `2`, push cost is `1`.
   - Remaining are `1` and `0`. Since `1` is new, move and then push (`2` move cost + `1` push cost).
   - Since we're already at `1`, we only push `0` which incurs only `1` push cost.
   - The total minimum cost for `f(2, 10)` is (2 + 1 + 2 + 1 + 2 + 1 + 1 = 10).
3. **Consider the second scenario where we subtract a minute and add 60 seconds.**
   Since `m = 2`, it's possible to subtract one minute to get `1` minute. Add 60 seconds to `10` to get `70` seconds. Now we input `f(1, 70)`:
   - Convert to array of digits: `m = 1` becomes `01` and `s = 70` becomes `70` for a complete array `[0, 1, 7, 0]`.
   - Start at `1`, move to `0` (start digit does not require push), move cost is `2`.
   - Push `0`, but since it's leading, we ignore this and move to `1`.
   - Push `1`, push cost is `1`.
   - Move to `7`, move cost is `2`.
   - Push `7`, push cost is `1`.
   - Push `0`, push cost is `1`.
   - The total minimum cost for `f(1, 70)` is (2 + 1 + 2 + 1 + 1 = 7).
4. **Select the minimum of the two.**
   Now we simply select the minimum of `f(2, 10)` and `f(1, 70)`, which in this case is `f(1, 70)` so, the minimum fatigue cost is `7`.

With this example, we see the solution finds the most efficient manner to set the required time on the microwave by converting the time and comparing the fatigue cost of two close time settings, thus ensuring the user expends the least amount of physical effort.

Python Solution

```python
class Solution:
    def minCostSetTime(self, start_at: int, move_cost: int, push_cost: int, target_seconds: int) -> int:
        # Define a function to calculate the cost based on minutes and seconds.
        def calculate_cost(minutes: int, seconds: int) -> int:
            # Check if minutes and seconds are within the obligatory range.
            if not 0 <= minutes < 100 or not 0 <= seconds < 100:
                return float('inf')  # Use 'inf' to denote an impossible situation.

            # List out the digits for minutes and seconds.
            digits = [minutes // 10, minutes % 10, seconds // 10, seconds % 10]

            # Skip leading zeros.
            index = 0
            while index < 4 and digits[index] == 0:
                index += 1

            # Initialize total cost and set the initial position for starting.
            total_cost = 0
            previous_digit = start_at

            # Calculate the total cost considering the move and push costs.
            for value in digits[index:]:
                if value != previous_digit:
                    total_cost += move_cost  # Add move cost if there is a change in digit.
                total_cost += push_cost    # Add push cost for every digit.
                previous_digit = value     # Update the previous digit.

            return total_cost

        # Convert total seconds to minutes and seconds.
        minutes, seconds = divmod(target_seconds, 60)

        # Calculate the cost of the two possible ways to enter the minutes and seconds.
        answer = min(
            calculate_cost(minutes, seconds),
            calculate_cost(minutes - 1, seconds + 60)
        )

        # Return the minimum cost of the two possibilities.
        return answer

```

Java Solution

```java
class Solution {
    // Entry method to calculate the minimum cost to set time on a digital clock
    public int minCostSetTime(int startAt, int moveCost, int pushCost, int targetSeconds) {
        // Calculate the initial minutes and seconds based on targetSeconds
        int minutes = targetSeconds / 60;
        int seconds = targetSeconds % 60;

        // Use the calculated minutes and seconds to find the cost, considering two scenarios:
        // 1. Direct input
        // 2. Subtracting a minute to possibly reduce the move cost and then adding 60 seconds
        return Math.min(
            calculateCost(minutes, seconds, startAt, moveCost, pushCost),
            calculateCost(minutes - 1, seconds + 60, startAt, moveCost, pushCost)
        );
    }

    // Helper method to calculate the cost of setting the time using number keypad
    private int calculateCost(int minutes, int seconds, int prevDigit, int moveCost, int pushCost) {
        // If minutes or seconds are out of range, return the maximum value as this is not a valid time
        if (minutes < 0 || minutes > 99 || seconds < 0 || seconds > 99) {
            return Integer.MAX_VALUE;
        }

        // Create an array representing the four digits of the time in MMSS format
        int[] digits = new int[] {minutes / 10, minutes % 10, seconds / 10, seconds % 10};
        int index = 0;

        // Skip leading zeros in the time representation
        while (index < 4 && digits[index] == 0) {
            index++;
        }

        // Initialize the total cost to 0
        int totalCost = 0;
        // Calculate the cost digit by digit
        for (; index < 4; ++index) {
            // If the current digit is different from the previous, add the move cost
            if (digits[index] != prevDigit) {
                totalCost += moveCost;
            }
            // Add the push cost for the current digit
            totalCost += pushCost;
            // Update the previous digit to current for next iteration
            prevDigit = digits[index];
        }
        return totalCost;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum cost to set the microwave to a target time.
    int minCostSetTime(int startAt, int moveCost, int pushCost, int targetSeconds) {
        // Maximum minutes on the microwave is 99, so calculate minutes and seconds accordingly.
        int minutes = targetSeconds / 60;
        int seconds = targetSeconds % 60;

        // Calculate costs for two possible ways to enter the time.
        // We can either input the exact minutes and seconds, or input one minute less
        // and add 60 seconds (in case it yields a lower cost due to fewer button moves).
        int costExactTime = calculateCost(minutes, seconds, startAt, moveCost, pushCost);
        int costOneMinuteLess = calculateCost(minutes - 1, seconds + 60, startAt, moveCost, pushCost);

        // Return the minimum of the two calculated costs.
        return min(costExactTime, costOneMinuteLess);
    }

private:
    // Function to calculate the cost of setting the given time on the microwave.
    int calculateCost(int minutes, int seconds, int prevDigit, int moveCost, int pushCost) {
        // If time is out of valid range, return maximum possible cost.
        if (minutes < 0 || minutes > 99 || seconds < 0 || seconds > 99) return INT_MAX;

        // Create a vector to store each digit of the minutes and seconds.
        vector<int> digits = {minutes / 10, minutes % 10, seconds / 10, seconds % 10};

        // Skip leading zeroes to start pushing buttons from the first non-zero digit.
        int i = 0;
        while (i < 4 && digits[i] == 0) {
            ++i;
        }

        int totalCost = 0; // Total cost starts at 0.
        // Calculate the total cost starting from the first non-zero digit.
        for (; i < 4; ++i) {
            // If the current digit is different from the previous one, add move cost.
            if (digits[i] != prevDigit) totalCost += moveCost;
            // Add push cost for every digit entered.
            totalCost += pushCost;
            // Set the current digit as the previous one for the next iteration.
            prevDigit = digits[i];
        }
        return totalCost; // Return the total cost of the operation.
    }
};

```

Typescript Solution

```typescript
// Function to find the minimum cost to set the microwave to a target time.
function minCostSetTime(startAt: number, moveCost: number, pushCost: number, targetSeconds: number): number {
    // Calculate minutes and seconds from target seconds.
    let minutes = Math.floor(targetSeconds / 60);
    let seconds = targetSeconds % 60;

    // Calculate costs for inputting the exact time or one minute less.
    let costExactTime: number = calculateCost(minutes, seconds, startAt, moveCost, pushCost);
    let costOneMinuteLess: number = calculateCost(minutes - 1, seconds + 60, startAt, moveCost, pushCost);

    // Return the minimum of the two calculated costs.
    return Math.min(costExactTime, costOneMinuteLess);
}

// Function to calculate the cost of setting a given time on the microwave.
function calculateCost(minutes: number, seconds: number, prevDigit: number, moveCost: number, pushCost: number): number {
    // Ensure time is in the valid range.
    if (minutes < 0 || minutes > 99 || seconds < 0 || seconds > 99) return Number.MAX_SAFE_INTEGER;

    // Store each digit of the minutes and seconds in an array.
    let digits: number[] = [Math.floor(minutes / 10), minutes % 10, Math.floor(seconds / 10), seconds % 10];

    // Find the first non-zero digit, skipping leading zeroes.
    let i: number = 0;
    while (i < digits.length && digits[i] === 0) {
        i++;
    }

    let totalCost: number = 0;
    // Calculate total cost from the first non-zero digit.
    for (; i < digits.length; i++) {
        // Add move cost if the current digit is different from the previous one.
        if (digits[i] !== prevDigit) totalCost += moveCost;
        // Add push cost for every digit entered.
        totalCost += pushCost;
        // Update prevDigit to the current one for the next iteration.
        prevDigit = digits[i];
    }

    // Return the calculated total cost.
    return totalCost;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by the operations that it performs:

- The function `f` is called exactly two times, regardless of the input size, so we can consider it a constant-time operation in the context of analyzing the overall function. Within `f`, most operations are simple arithmetic operations, conditions, and a loop which, in the worst case, iterates at most 4 times, as it's dealing with time in minutes and seconds.
- The cost computation within the loop runs a constant number of times, as mentioned, at most 4, since the digits for minutes and seconds each can be in the range [0, 99]. The condition and the arithmetic inside the loop are constant-time operations.

Hence, the time complexity of the `f` function is O(1).

Since there are no other loops or recursive calls that depend on the size of the input in the main function and `f` is called only twice, the time complexity of the `minCostSetTime` function is also O(1).

Space Complexity

- The `f` function uses a fixed amount of additional space: an array `arr` of size 4 and a few integer variables to hold parameters and intermediate results.
- There is no dynamic data structure or additional memory usage that scales with the input size.

Therefore, the space complexity of the function is O(1).