2139. Minimum Moves to Reach Target Score

Medium Greedy Math Leetcode Link

Problem Description

In this game, you begin with the number `1` and your goal is to reach a given `target` number. You can perform two types of moves: increment the current number by `1` or double it. The increment operation can be used as many times as needed, but the double operation can only be used up to a specified limit, `maxDoubles`. Your objective is to determine the smallest number of moves required to go from `1` to the `target` number, considering the limit on the number of double operations.

Intuition

Consider working backwards from the `target` number down to `1`. If `target` is greater than `1` and we still have the option to double (i.e., `maxDoubles` is not `0`), we need to decide between incrementing or doubling. To make efficient use of our moves, we should prefer doubling whenever possible because it can dramatically reduce the `target` number, especially when `target` is even. If `target` is odd, we have no choice but to decrement it by `1` to make it even, at which point doubling becomes an option again.

If `target` is even and we have the capability to double (meaning `maxDoubles > 0`), we should apply the doubling move—which is just a simple division by `2` of the target in the case of our reverse strategy. On the other hand, if `target` is already `1`, we are done, and no moves are needed. The same applies if we run out of double operations (`maxDoubles = 0`); we will just keep decrementing the `target` by `1` until we reach the value of `1`.

We continue this process iteratively, incrementing our move count each time we make an adjust to the `target`. Once we've exhausted our doubling options, or the `target` reaches `1`, we simply add the remaining distance from `target` to `1` to our move count. This remaining distance will be precisely `target - 1`, because, at that stage, we can only increment by `1` to reach our goal.

The provided solution implements this approach iteratively, avoiding the overhead of recursion, and ensures we use the minimum moves by always choosing to double rather than increment when both moves are possible.

Solution Approach

The solution can be understood as a greedy algorithm. A greedy algorithm is an approach for solving problems by making a sequence of choices, each of which simply looks like the best at the moment, without considering future consequences. In this problem, at each step, we attempt to make a move that brings us closer to `1` in the quickest way possible, given our constraint on the number of double operations available (denoted by `maxDoubles`).

Here's the step-by-step breakdown of the algorithm referenced in the Solution provided:

1. Initialize a counter for the number of moves (`ans`) to `0`.
2. Start a while loop that continues as long as there are double operations left (`maxDoubles > 0`) and the `target` is greater than `1`.
3. Increment the move counter (`ans`) by `1` at each step of the loop—that's because we're making a move, either doubling or incrementing.
4. Within the loop, check if `target` is odd (`target % 2 == 1`). If so, subtract `1` to make it even. This is a necessity since doubling is only efficient on even numbers. Note that this subtraction is effectively reversing an increment operation.
5. If the `target` is even, halve the `target` (`target >>= 1` is equivalent to `target = target / 2`). This represents the reverse of a doubling operation. Also, decrement the count of available double operations (`maxDoubles`) since we've just used one.
6. After exiting the loop, if no double operations are left or the `target` has reached `1`, add the difference between the `target` and `1` to the move counter (`ans`). This is because now we can only use increment operations to reach `1`.
7. Finally, return the total number of moves (`ans`). This value represents the minimum moves needed to reach `target` starting from `1` under the given constraints.

The space complexity of the algorithm is O(1), since it only requires a fixed amount of additional space (for the variables `ans`, `target`, and `maxDoubles`). The time complexity is O(min(log(target), maxDoubles)), which comes from the fact that doubling reduces the `target` by a factor of 2, and thus it would take at most log2(target) doublings to reach `1` (in a scenario with unlimited doublings), and we will make at most `maxDoubles` doubling moves.

This approach guarantees the minimum number of moves since:

- Doubling when possible minimizes the number of operations by taking the largest possible reduction at each step.
- Using increment operations only when necessary ensures that we are not wasting any double operations.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the target number `10` and a `maxDoubles` limit of `2`.

1. We start with `target = 10` and `maxDoubles = 2`. Our initial move counter (`ans`) is `0`.
2. Since `maxDoubles > 0` and `target > 1`, we enter the loop.
3. `target` is even, so we can apply a double move. We halve `target` (10 becomes 5) and increment `ans` by `1` (so `ans` is now `1`). We also reduce `maxDoubles` by `1` (now it's `1`).
4. `target` is now `5`, which is odd. We can't double an odd number, so we subtract `1` from the `target` (5 becomes 4) and increment `ans` by `1` (`ans` is now `2`).
5. With `target` back to an even number (4), and `maxDoubles > 0`, we perform a double move. We halve `target` (4 becomes 2) and increment `ans` by `1` (`ans` is now `3`), and decrement `maxDoubles` by `1` (`maxDoubles` is now `0`).
6. `target` is `2`, which is even, but we're out of double moves. So now we have to simply subtract `1` to continue. The `target` becomes `1`, and we increment `ans` by `1` (so `ans` is now `4`).
7. We have now reached `1`, so we don't need to enter the loop again. There is no need to add the difference between `target` and `1` to `ans`, because `target` is already `1`.
8. The loop is finished, and the `ans` value is `4`. This is the least number of moves needed to get from `1` to `10` with a maximum of `2` doubles.

In this particular case, the `ans` (which is `4`) represents the smallest number of moves required to go from `1` to `10` when you can double no more than `2` times. With these moves, we've utilized both available doubles efficiently and only incremented when necessary.

Python Solution

```python
class Solution:
    def min_moves(self, target: int, max_doubles: int) -> int:
        # Initialize the number of moves to 0
        moves = 0

        # Continue until no more doubles are allowed,
        # or the target is reduced to 1
        while max_doubles and target > 1:
            # Increment the move count for every operation
            moves += 1

            # If the target is odd, subtract 1 (increment operation)
            if target % 2 == 1:
                target -= 1
            else:
                # If the target is even, use a double operation
                max_doubles -= 1   # Use one of the allowed doubles
                target >>= 1       # Halve the target by right shifting

        # After using all doubles, add the remaining distance to 1
        # (all remaining moves are increments)
        moves += target - 1

        # Return the total number of moves
        return moves

# The method name `min_moves` is used to initiate an action to find the minimum moves.
# The variable `moves` is more readable and standardized according to Python naming conventions.
# Comments are provided to explain what each segment of the code is doing.

```

Java Solution

```java
class Solution {
    public int minMoves(int target, int maxDoubles) {
        // This variable stores the number of moves required to reduce the target to 1.
        int moves = 0;

        // Continue the loop until we have no more doubling operations available
        // or until the target becomes 1.
        while (maxDoubles > 0 && target > 1) {
            // Increment the move count with each iteration of the loop.
            moves++;

            // If the target is odd, we subtract one to make it even (operation type 1).
            if (target % 2 == 1) {
                target--;
            } else {
                // If the target is even and we still have doubling operations left,
                // we use one and halve the target (operation type 2).
                maxDoubles--;
                target /= 2; // equivalent to target >>= 1, but clearer with respect to halving.
            }
        }

        // If there are no double operations left, add the remaining (target - 1)
        // to the moves as we can only decrement by 1 in each move from then on.
        moves += target - 1;

        // The total number of moves is returned.
        return moves;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to compute the minimum number of moves required to reach 'target' starting from 1.
    // `maxDoubles` defines the maximum number of times the doubling operation can be performed.
    int minMoves(int target, int maxDoubles) {
        int numMoves = 0;  // Initialize a counter for the number of moves.

        // Continue reducing 'target' while doubles are still available and 'target' is greater than 1.
        while (maxDoubles > 0 && target > 1) {
            numMoves++;  // Increment the moves counter.

            // If 'target' is odd, perform an increment operation to make it even.
            if (target % 2 == 1) {
                target--;  // Decrement 'target' to make it even, which counts as a move.
            } else {
                maxDoubles--;  // Use a double operation and decrement the remaining doubles.
                target /= 2;   // Halve the target since it's even.
            }
        }

        // After finishing all the available doubles or reaching `target` <= 1,
        // perform (target - 1) increment operations to reach exactly 'target'.
        numMoves += target - 1;

        // Return the total number of moves calculated.
        return numMoves;
    }
};

```

Typescript Solution

```typescript
function minMoves(target: number, maxDoubles: number): number {
    let moves = 0; // Initialize the count of moves

    // As long as there are remaining doubles and target is greater than 1, keep iterating
    while (maxDoubles > 0 && target > 1) {
        moves++; // Increment the move counter

        if (target % 2 === 1) {
            // If the target is odd, decrement it to make it even
            target--;
        } else {
            // If the target is even, utilize a double and halve the target
            maxDoubles--;
            target /= 2;
        }
    }

    // Once no more doubles are allowed, increment the move count directly to reach 1
    moves += target - 1;

    return moves; // Return the total number of moves required
}

```

Time and Space Complexity

The time complexity of the given code is `O(log(target))`. This is because the while loop can run at most `log2(target)` times if `maxDoubles` is sufficiently large. Each operation inside the loop involves either a simple subtraction by 1 (in case `target` is odd) or a division by 2 using bitwise right shift (in case `target` is even). The final operation outside the loop is a single subtraction, which is done in constant time.

The space complexity of the code is `O(1)` since no additional space is used that scales with the input size. All operations are performed using a fixed amount of variables (`ans`, `target`, and `maxDoubles`).