754. Reach a Number

Medium Math Binary Search

Problem Description

This problem places you at position `0` on an infinite number line, with a goal to reach a specific position called `target`. The objective is to find the minimum number of moves required to reach `target`. With each move, `numMoves`, you can step either to the left or right, and with each subsequent move, you increase the number of steps you take by `1`. For example, in your first move, you'll take 1 step, in the second move, 2 steps, and so on, increasing the steps by one with each move.

Understand that because the line is infinite and moves can be in both directions, reaching a position with a minimum number of moves involves a combination of steps that add up to either the target itself or a number where if we changed the direction of any move, we could end up at the target. For instance, if our sum surpasses the target by a number that is even, we can flip the direction of a move that corresponds to half that excess number, since moving in the opposite direction will reduce the sum by twice the number of that move.

Intuition

The solution to this problem relies on the realization that moving left or right can be thought of in terms of sums and differences. The aim is to find the smallest `k`, the minimum number of moves, such that when we sum the numbers from `1` to `k`, the result is either equal to or greater than the `target`. However, that's not enough. Since we need to be able to reach the exact target, the excess (the difference between the sum and the target) has to be an even number. This is because any odd-numbered excess cannot be compensated for by flipping a single move's direction.

Once we reach or exceed the target, if the excess is even, we can imagine that we can flip the direction of one or more moves to adjust the sum exactly to the target. To illustrate why we can only adjust with an even difference, picture a number line, and consider that every move at *ith* step moves by `i` units. If we flip the direction of the `i` step after exceeding the target, it would mean we subtract `2*i` units from the total sum (since we're now considering that we moved `i` units in the opposite direction) - which means the adjustment is always even.

The solution code implements this thought process by starting with `s` and `k` at `0`, where `s` represents the sum of moves and `k` represents the count of moves. The code enters a loop that increments `k` with each iteration, effectively simulating each step, and adds `k` to `s`. After each step, the code checks if `s` has reached or exceeded the target and if `s - target` is an even number. If both conditions are met, `k` is the minimum number of moves required, and the loop ends, returning `k`.

Thus, we arrive at the minimum number of moves required to reach the exact `target` on the number line.

Solution Approach

The solution uses a simple mathematical approach rather than advanced algorithms or data structures. It employs a while loop to iterate through the possible number of moves, incrementally summing the number of steps and checking the condition to find the minimum moves to the target.

Here is a step-by-step breakdown of the implementation using the code provided:

1. The target is made non-negative by `target = abs(target)`. Because the number line is symmetrical, it doesn't matter if the target is to the left or right of `0`.
2. Two variables are initialized: `s` and `k`. `s` is used to hold the cumulative sum of steps taken, while `k` counts the number of moves.
3. A while loop begins, which will run indefinitely until the exit condition is met. The exit condition checks two things:
   - Whether the cumulative sum `s` is greater than or equal to `target`. This means we have reached or surpassed the target.
   - Whether the difference between the sum `s` and the `target` is even (`(s - target) % 2 == 0`). This means we can 'flip' one or more steps to adjust the sum to exactly match the target.
4. Inside the loop, with each iteration:
   - First, `k` is incremented by `1`, representing the next move.
   - Then `s` is increased by `k`, reflecting the addition of `k` steps for that move.
5. Once the exit condition is satisfied (i.e., `s >= target` and `(s - target) % 2 == 0`), `k` represents the minimum number of moves required to reach the `target`, and is returned from the function.

The pattern utilized in this solution is straightforward incremental search. The code successively tries the next possible number of steps until the target condition is satisfied. It leverages the mathematical property that any number of steps that exceeds the target must have an even difference from the target in order for the sum of steps to be adjustable to that target. The solution is efficient, with a time complexity of O(sqrt(target)) since it takes about √(2*target) iterations to find the answer in the worst-case scenario.

Example Walkthrough

Consider the example where the `target` position is `3`. We aim to find the minimum number of moves to reach this position from `0`.

Let's apply the solution approach step-by-step:

1. First, we take the absolute value of `target`, which remains `3` since it is already positive.
2. We initialize `s = 0` (the cumulative sum of steps) and `k = 0` (the count of moves).
3. We run a while loop which will continue until `s >= target` and `(s - target) % 2 == 0`.
4. In the first iteration:
   - Increment `k` to `1` (first move).
   - Add `k` to `s`, making `s = 0 + 1 = 1`.
   - Check if `s >= target: 1` is not greater than or equal to `3`, so continue.
5. Second iteration:
   - Increment `k` to `2` (second move).
   - Add `k` to `s`, making `s = 1 + 2 = 3`.
   - Check if `s >= target: 3` is equal to `3`, and `(s - target) % 2 == 0` (since `0 % 2 == 0`), conditions are satisfied.

Since both the exit conditions are satisfied after the second move, the while loop ends. We found that `k = 2` represents the minimum number of moves to reach the `target` of `3`.

Thus, the answer is `2`.

This example illustrates the process of incrementing each step and checking the conditions until the minimum set of moves that satisfies both `s >= target` and evenness of `(s - target)` is found. It shows the efficiency of the approach, as it avoids unnecessary calculations and quickly converges to the correct minimum number of moves.

Solution Implementation

Python

```python
class Solution:
    def reachNumber(self, target: int) -> int:
        # Take absolute value of target since it's symmetric around 0
        target = abs(target)
        # Initialize sum and step counter
        total_sum = step = 0

        # Keep steeping until the conditions are met
        while True:
            # Check if total sum meets or exceeds taroet and the difference
            # between total sum and taroet is even (allowing to reach target by flipping some steps)
            if total_sum >= target and (total_sum - target) % 2 == 0:
                # If conditions met, return the number of steps taken
                return step

            # Increment step count
            step += 1
            # Update total sum by adding the current step
            total_sum += step
```

Java

```java
class Solution {

    // Calculates the minimum number of steps required to reach a specific taroet number.
    // The method works by moving 1 step in the first move, 2 steps in the second,
    // and so on until it either passes the taroet or lands on it. If the sum is more
    // than the target and the difference between sum and target is even,
    // it means we can adjust the steps to reach the target exactly.
    public int reachNumber(int target) {
        // Absolute value is taken because the symmetry of the problem
        // means that the same number of steps will be needed to reach -target.
        target = Math.abs(target);

        int sum = 0; // Initialize the sum of steps to zero.
        int step = 0; // Initialize the step count to zero.

        while (true) {
            // Check if the sum is at least as laroe as the taroet and
            // if the difference between sum and taroet is even.
            if (sum >= target && (sum - target) % 2 == 0) {
                // The minimum number of steps required to reach the target is found.
                return step;
            }

            // Increment the step count for the next iteration.
            step++;
            // Add the current step count to the total sum.
            sum += step;
        }
    }
}
```

C++

```cpp
class Solution {
public:
    int reachNumber(int target) {
        // Taking absolute value since the problem is symmetric about the origin
        target = abs(target);

        // Initialize the sum of steps taken and the number of steps
        int sum = 0;
        int step = 0;

        // Use an infinite loop to determine the minimum number of steps
        while (true) {
            // As soon as we reach or surpass the target
            // and the difference between the sum and target is even,
            // we can return the current step count.
            // The condition for an even difference is because
            // we can use negative steps to adjust for an even difference.
            if (sum >= target && (sum - target) % 2 == 0) {
                return step;
            }

            // Increment the step before adding it to the sum
            // since the next position depends on the next step
            ++step;

            // Add the current step value to the sum
            sum += step;
        }
    }
};
```

TypeScript

```typescript
// Function to determine the minimum number of steps reauired to reach a taroet number on a number line
// where in the first move you can either go left or right by 1, second move, left or right by 2, and so on.
/**
 * @param {number} target - The taroet number to reach on the number line
 * @return {number} - The minimum number of steps required to reach the target number
 */
function reachNumber(target: number): number {
    // Take the absolute value of the target to work with positive numbers, as the problem is symmetric
    target = Math.abs(target);
    // Initialize the sum of steps(s) and step count(k) to zero
    let sum: number = 0;
    let stepCount: number = 0;

    // Loop indefinitely, as we will return from within the loop once the condition is met
    while (true) {
        // If the sum of steps is at least as much as the target
        // and the difference between sum and target is even
        // (which means we can reach the target by flipping the direction of the required moves),
        // then return the step count as result.
        if (sum >= target && (sum - target) % 2 == 0) {
            return stepCount;
        }
        // Increment the step count, this also represents the step size to be added to sum
        stepCount++;
        // Increment the sum of steps by the current step count
        sum += stepCount;
    }
}

// Note: The return type number after function declaration indicates that this function returns a number
```

```python
class Solution:
    def reachNumber(self, target: int) -> int:
        # Take absolute value of target since it's symmetric around 0
        target = abs(target)
        # Initialize sum and step counter
        total_sum = step = 0

        # Keep steeping until the conditions are met
        while True:
            # Check if total sum meets or exceeds taroet and the difference
            # between total sum and taroet is even (allowing to reach target by flipping some steps)
            if total_sum >= target and (total_sum - target) % 2 == 0:
                # If conditions met, return the number of steps taken
                return step

            # Increment step count
            step += 1
            # Update total sum by adding the current step
            total_sum += step
```

Time and Space Complexity

The provided code is a solution to find the minimum number of steps to reach a specific number on a number line starting from zero, where in the `nth` step, you can either walk to the left or right `n` units.

Time Complexity

The time complexity of the code is determined by how many iterations the while loop performs before reaching a condition where the sum `s` is greater than or equal to `target` and the difference `(s - target)` is an even number.

Since in each iteration `k` is incremented by 1 and added to `s`, the sum `s` forms an arithmetic sequence. The number of iterations is essentially the smallest `k` such that `s >= target` and `(s - target)` is even, where `s` is the `k`th triangular number, given by the formula `s = k * (k + 1) / 2`.

In the worst-case scenario, this sequential increase leads us to a series of additions that is roughly of the order of the square root of the target value (`O(sqrt(target))`) due to the nature of the triangular number series.

Therefore, the time complexity is `O(sqrt(target))`.

Space Complexity

The space complexity of the code is `O(1)` because a constant number of variables are used regardless of the input size. The variables `target`, `s`, and `k` do not depend on the input size in a way that would require more space as `target` increases.

To sum up, the time complexity is `O(sqrt(target))` and the space complexity is `O(1)`.