2769. Find the Maximum Achievable Number

Easy Math

Problem Description

In this problem, we are provided with two integers, `num` and `t`. We have to understand the concept of an "achievable" integer `x`. An integer `x` is considered achievable if we can make it equal to `num` by using a certain operation at most `t` times. This operation involves changing the value of `x` by `1` (either increasing or decreasing it) and simultaneously changing `num` by `1` in the opposite direction (if `x` is decreased, `num` is increased, and vice versa). Our task is to find the maximum possible achievable integer `x`.

To solve this, we need to think about how the operation affects the relationship between `x` and `num`. Each operation changes the difference between `x` and `num` by 2 units. Because we have the freedom to do this `t` times, we must evaluate how far we can push `x` away from `num` initially so that after at most `t` operations they can still become equal.

Intuition

To derive the intuition behind the solution, it's crucial to look at the effect of the operation on the difference between `x` and `num`. Since each operation can be performed at most `t` times, we want to maximize `x` before we start performing operations.

If we increase `x` by `1` and decrease `num` by `1`, the difference between them increases by `2`. Conversely, if we decrease `x` by `1` and increase `num` by `1`, the difference decreases by `2`. Since we're looking for the maximum achievable `x`, we'll want to increase `x` as much as possible before hitting the limit of `t` operations. Since `t` is the maximum number of times we can perform this operation, the farthest we can stretch `x` from `num` initially is by `t` operations, each of which will increase the difference by `2`. Therefore, the maximum achievable `x` can simply be calculated by taking `num` and adding `t*2` to it.

Thus, even without running through the operations, we can directly find the maximum achievable `x` by this straightforward mathematical relationship.

Solution Approach

The given problem is simple in nature and doesn't require complex algorithms or data structures. It's essentially a mathematical problem that we solve through an understanding of integers and their relationships after a series of operations.

Given that we can either increase or decrease `x` by 1 and simultaneously do the opposite to `num` in the same operation, we realize that each operation will either widen or narrow the gap between `x` and `num` by 2. To reach the highest achievable `x`, we need to maximize `x` from the onset.

As per the Reference Solution Approach, we see that this operation of increasing `x` by 1 and decreasing `num` by 1 (or vice versa) can be done up to `t` times. Therefore, to maximize `x`, we would theoretically perform all `t` operations in the direction that increases `x` (and decreases `num`).

The solution code implements this straightforward insight:

```
class Solution:
    def theMaximumAchievableX(self, num: int, t: int) -> int:
        # By performing the operation t times, each time increasing x by 1,
        # the maximum achievable x is calculated.
        return num + t * 2
```

Here, we take the original `num` and add `t * 2` to account for `t` operations, each increasing the initial value of `x` by 2. There's no iteration, conditional logic, or use of additional space for data structures, making the entire solution a single mathematical expression.

In conclusion, the solution relies on the understanding that each operation can happen `t` times, changing the value of `x` by a total of `t * 2`. We immediately return `num + t * 2` as the result, representing the maximum achievable `x`. This simplicity is what makes the problem solvable in constant time and space complexity, i.e., O(1).

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have `num = 10` and `t = 3`. Our goal is to find the maximum possible achievable integer `x`.

Recall that the operation we can perform changes `x` by `1` and simultaneously changes `num` by `1` in the opposite direction. Let's see how we can use this operation to our advantage:

1. We start with `num = 10` and we choose an initial value for `x`. As we want to find the maximum achievable `x`, it's intuitive to start by increasing `x`.
2. For the first operation, if we increase `x` by `1` (making `x` initially 11), we have to decrease `num` by `1` (making `num` now 9).
3. After the first operation, the difference between `x` and `num` is `11 - 9 = 2`. We can perform the operation two more times since `t=3`.
4. For the second operation, we increase `x` by `1` to 12 and decrease `num` by `1` to 8. The difference is now `12 - 8 = 4`.
5. Finally, for the third operation, we increase `x` by `1` to 13 and decrease `num` by `1` to 7. The difference is now `13 - 7 = 6`.

Since we have now performed the operation `t` times (which is 3 times in our case), we have reached our limit, and the resulting value of `x` is the maximum achievable. Every time we performed an operation, we increased the difference by `2`, and since we've done that `t` times, the maximum increase in `x` compared to the original `num` is `t * 2`.

In our example, the maximum achievable `x` is therefore `10 + (3 * 2) = 16`.

This simple example illustrates that we can calculate the maximum achievable `x`, without having to actually increment and decrement `x` and `num` step by step, by simply taking the original `num` and adding `t * 2` to it, as shown in the solution code. Thus, our answer for this example is `x = 16`.

Solution Implementation

Python

```
class Solution:
    def theMaximumAchievableX(self, num: int, t: int) -> int:
        # This method calculates the maximum value achievable for 'x'
        # by adding the given number 'num' and twice the value of 't'.

        # Calculate the result by adding 'num' with double the value of 't'
        result = num + t * 2

        # Return the computed result
        return result
```

Java

```
class Solution {

    // Method to calculate the maximum achievable value of X.
    // Parameters:
    //   int num: The base number from which we start.
    //   int t: The number of times we perform the operation to increase 'num'.
    // Returns:
    //   The maximum achievable value of X after performing the operation 't' times.
    public int theMaximumAchievableX(int num, int t) {
        // The operation consists of doubling the increment each time (hence, t * 2)
        // Add the total increment to the original 'num' to get the maximum achievable X.
        return num + t * 2;
    }
}
```

C++

```
class Solution {
public:
    // Function to calculate the maximum achievable value of x
    int theMaximumAchievableX(int num, int t) {
        // The formula to compute the maximum achievable value
        int maximizedX = num + t * 2;

        // Return the calculated value
        return maximizedX;
    }
};
```

TypeScript

```
// This function calculates the maximum achievable value of x given a number 'num' and a multiplier 't'.
// The value of 'x' is found by adding 'num' to twice the value of 't'.
// @param {number} num - The base number to which the result of 't * 2' will be added.
// @param {number} t - The multiplier that is used to find the additional value to add to 'num'.
// @returns {number} - The maximum achieved value of 'x'.

function theMaximumAchievableX(num: number, t: number): number {
    // Calculate the result by adding num to twice the value of t and return it.
    return num + t * 2;
}
```

```
class Solution:
    def theMaximumAchievableX(self, num: int, t: int) -> int:
        # This method calculates the maximum value achievable for 'x'
        # by adding the given number 'num' and twice the value of 't'.

        # Calculate the result by adding 'num' with double the value of 't'
        result = num + t * 2

        # Return the computed result
        return result
```

Time and Space Complexity

The time complexity of the given code is `O(1)`. This is because the operation `num + t * 2` is performed in constant time, regardless of the size of the input values for `num` and `t`.

Similarly, the space complexity of the code is also `O(1)`. The function does not allocate any additional space that grows with the input size. It only uses a fixed amount of space to store the input parameters and the result of the computation.