1860. Incremental Memory Leak

Medium Simulation Leetcode Link

Problem Description

In this problem, we are dealing with two memory sticks, each represented by an integer value that indicates the amount of memory available in bits (`memory1` and `memory2`). There's a malfunctioning program that continuously consumes an increasing amount of memory every second.

The consumption pattern is such that at every second `i` (starting from 1), `i` bits of memory are needed. These bits are allocated to whichever memory stick has more available memory at that point in time. If both sticks have the same amount of memory available, then the first stick (`memory1`) is used. The process continues until there's a second where neither memory stick has enough available memory to allocate the required `i` bits, which results in the program crashing.

Our task is to determine at what time `crashTime` the program will crash and how much memory (`memory1_crash` and `memory2_crash`) will be available on the two sticks at that time.

Intuition

The algorithm is based on the straightforward simulation of the memory allocation process, sequentially, second by second. Starting at second `i = 1`, we check which stick has more memory. If they're equal, we default to the first stick as per the rules. Then we decrease the memory available on the chosen stick by `i` bits. This step is repeated, increasing `i` by 1 each time, simulating the passing seconds and increasing memory demand of the faulty program.

The process continues until the condition is reached where neither memory stick has enough memory available to meet the program's demand (`i` bits). At this point, we know the program has crashed, and we can return the current second as `crashTime`, along with the remaining memory bits on each stick (`memory1_crash` and `memory2_crash`).

The solution approach is efficient because it's a direct simulation and doesn't require any extra data structures or complex logic. This approach takes advantage of the problem's simplicity by working through the memory allocation step by step and stopping once the crashing condition is met.

Solution Approach

The solution uses a simple `while` loop as its core structure to implement the algorithm, and no additional data structures are necessary. Here's the step-by-step breakdown of the implementation according to the reference solution above:

1. Initialize an integer variable, `i`, to 1. This variable represents both the current second and the amount of memory required in the current second.
2. Enter a loop that continues until the value of `i` exceeds both `memory1` and `memory2`, the available memory in each memory stick. This condition is checked by the loop's conditional statement: `while i <= max(memory1, memory2)`.
3. Inside the loop, a conditional check is performed to determine which memory stick should have memory allocated to it. This is determined by comparing `memory1` and `memory2`. If `memory1` is greater than or equal to `memory2`, `memory1` is chosen:

   ```
   if memory1 >= memory2:
       memory1 -= i
   ```

   Otherwise, `memory2` is chosen:

   ```
   else:
       memory2 -= i
   ```

   Whichever stick is chosen, `i` bits are subtracted from its available memory.
4. After the memory has been allocated for the current second, increment `i` by 1 to simulate the next second: `i += 1`.
5. The loop exits when the program reaches a second in which neither memory stick has enough available memory for the `i` bits. At this point, the variable `i` represents the time at which the program crashes (because the `while` loop condition becomes false at the start of the second when the program is supposed to crash).
6. Finally, the function returns a list containing three elements: the crash time, and the remaining memory in `memory1` and `memory2`: `return [i, memory1, memory2]`.

This approach iterates through the seconds as long as the program hasn't crashed and adjusts the available memory on the memory sticks. It doesn't require complex data management or decision-making beyond simple arithmetic and comparison operations, thus it is efficient and easy to understand.

Example Walkthrough

Let's use a small example to illustrate the solution approach with `memory1` = 9 bits and `memory2` = 3 bits available.

- Initiate `i` to 1 representing the current second and the memory required.
- Enter the `while` loop since `i` is less than or equal to the max of `memory1` and `memory2`.

Second 1:

- Compare `memory1` (9) with `memory2` (3). `memory1` is greater, allocate `i` bits (1 bit) to `memory1`.
- After the allocation:
  - `memory1` = 9 - 1 = 8 bits
  - `memory2` = 3 bits
- Increment `i` by 1 (now `i` = 2).

Second 2:

- Compare `memory1` (8) with `memory2` (3). `memory1` is greater, allocate `i` bits (2 bits) to `memory1`.
- After the allocation:
  - `memory1` = 8 - 2 = 6 bits
  - `memory2` = 3 bits
- Increment `i` by 1 (now `i` = 3).

Second 3:

- Compare `memory1` (6) with `memory2` (3). `memory1` is greater, allocate `i` bits (3 bits) to `memory1`.
- After the allocation:
  - `memory1` = 6 - 3 = 3 bits
  - `memory2` = 3 bits
- Increment `i` by 1 (now `i` = 4).

Second 4:

- Compare `memory1` (3) with `memory2` (3). They are equal, so allocate `i` bits (4 bits) to `memory1` by default.
- Since `memory1` doesn't have enough memory to allocate 4 bits, the program cannot proceed to this second and the `while` loop exits.
- Now `memory1` is still 3 bits and `memory2` is still 3 bits.

The loop has concluded indicating that at the end of the third second, going into the fourth second, we do not have enough memory to allocate `i` bits to either of the memory sticks. So, the program would crash right before the fourth second starts.

The function would return `[4, 3, 3]`, representing the crash time (`crashTime` = 4 seconds), and the remaining memory in `memory1` (3 bits) and `memory2` (3 bits) at the time of the crash.

Python Solution

```python
from typing import List

class Solution:
    def mem_leak(self, memory1: int, memory2: int) -> List[int]:
        # Initialize the counter i, representing the memory units needed each time.
        i = 1

        # Continue the process while there is enough memory in either of the memory banks.
        while i <= max(memory1, memory2):
            # If memory1 has more or equal memory than memory2, subtract 'i' units from memory1.
            if memory1 >= memory2:
                memory1 -= i
            # Otherwise, subtract 'i' units from memory2.
            else:
                memory2 -= i

            # Increment 'i' for the next iteration as each time the requirement increases by 1 unit.
            i += 1

        # Return a list containing the current value of i, and the remaining memory in memory1 and memory2.
        return [i, memory1, memory2]

# Example usage:
# sol = Solution()
# result = sol.mem_leak(2, 2)
# print(result)  # Should output the time of crash and remaining memory in memory1 and memory2.

```

Java Solution

```java
class Solution {
    public int[] memLeak(int memory1, int memory2) {
        int second = 1; // Initialize a time counter starting at 1

        // The loop runs as long as either memory1 or memory2 is enough for the current time counter.
        // The condition inside the loop checks which memory to reduce based on which one is larger.
        while (second <= Math.max(memory1, memory2)) {
            if (memory1 >= memory2) {
                // if memory1 is larger or equal to memory2
                // memory1 is reduced by the current value of the time counter.
                memory1 -= second;
            } else {
                // if memory2 is larger than memory1
                // memory2 is reduced by the current value of the time counter.
                memory2 -= second;
            }
            second++; // Increment the time counter after each iteration
        }

        // Return result as an array containing the value of the time counter
        // when the loop stops, and the remaining memory in both memory slots.
        return new int[] {second, memory1, memory2};
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // This function simulates a memory leak scenario between two memory banks.
    vector<int> memLeak(int memory1, int memory2) {
        // Initialize a counter 'i', starting at 1, representing time seconds.
        int i = 1;

        // Continue the loop until 'i' is greater than both memory banks.
        // Note: Loop uses 1-based indexing, as 'i' represents seconds.
        while (i <= max(memory1, memory2)) {
            // If memory1 is not less than memory2,
            // then memory1 will leak 'i' amount of memory.
            if (memory1 >= memory2) {
                memory1 -= i;
            } else {
                // Otherwise, memory2 will leak 'i' amount of memory.
                memory2 -= i;
            }

            // Increment the counter 'i' by 1 for the next second.
            ++i;
        }

        // Return a vector containing the time (i), and the remaining
        // memory in memory bank 1 and memory bank 2 respectively.
        return {i, memory1, memory2};
    }
};

```

Typescript Solution

```typescript
/**
 * Simulates a memory leak where memory blocks are allocated iteratively
 * until neither of the two memory sources can satisfy the required memory block size.
 *
 * @param {number} memory1 - The size of the first memory source.
 * @param {number} memory2 - The size of the second memory source.
 * @returns {number[]} - An array with the first element being the first
 *                       iteration number that cannot be processed, and the
 *                       remaining two elements being the sizes of memory1 and memory2
 *                       after memory leak simulation.
 */
function memLeak(memory1: number, memory2: number): number[] {
    // Initialize iteration counter
    let iteration = 1;

    // Continue allocation until the required memory exceeds both memory sources
    // We can stop when iteration number exceeds the max of both memories as
    // this will be the first iteration that fails due to insufficient memory.
    while (iteration <= memory1 || iteration <= memory2) {
        // If memory1 is greater or equal, allocate from memory1; otherwise, allocate from memory2
        if (memory1 >= memory2) {
            memory1 -= iteration;
        } else {
            memory2 -= iteration;
        }

        // Move to the next iteration
        ++iteration;
    }

    // Return the result as [iteration number, size of memory1, size of memory2]
    return [iteration, memory1, memory2];
}

```

Time and Space Complexity

The time complexity of the given code depends on how quickly the while loop reaches a point where `memory1` and `memory2` are both less than `i`. Since `i` starts at 1 and increments by 1 on each iteration, and the maximum value of `i` that doesn't crash (exceeds the remaining memory) is at most `max(memory1, memory2)`, in the worst-case scenario, the loop can execute O(sqrt(max(memory1, memory2))) times. This is because the sum of the first `n` natural numbers is given by the formula `n*(n+1)/2`, and we're looking for the point where this sum exceeds `memory1` or `memory2`.

The space complexity of the method is O(1), which is constant, because the amount of extra memory used by the algorithm does not depend on the input size and is limited to a fixed number of integer variables (`i`, `memory1`, and `memory2`).