868. Binary Gap

Easy | Bit Manipulation

Problem Description

The problem requires calculating the longest distance between any two adjacent `1`s in the binary representation of a given positive integer `n`. In binary terms, adjacent `1`s are those with only `0`s or no `0` in between. The distance is measured by counting the number of bit positions between the two `1`s. If no two adjacent `1`s are found, the function should return `0`.

Intuition

To solve this problem, we need to inspect the binary representation of the number from the least significant bit (LSB) to the most significant bit (MSB). As we iterate through the binary bits, we keep track of the position of the most recent `1` encountered. Every time we find a new `1`, we compute the distance to the previous `1` (if there is one) and update our answer to be the maximum distance found so far.

The intuition behind the provided solution approach is as follows:

- We iterate over 32 bits because an integer in most systems is represented using 32 bits.
- At each bit position `i`, we check if the least significant bit is `1` by performing an "AND" operation with `1` (`n & 1`).
- If we encounter a `1`, we check if there is a previous `1` (indicated by `j != -1`). If there is, we calculate the distance between the current `1` (`i`) and the previous `1` (`j`), then update the result (`ans`) with the maximum distance found so far.
- We record the current bit position `i` as the new previous `1` position by setting `j = i`.
- After each iteration, we right-shift the number (`n >>= 1`) to check the next bit in the following iteration.
- We continue this process until all bits have been checked.
- The variable `ans` maintains the longest distance, which is returned at the end of the function.

Solution Approach

The solution employs a straightforward approach that leverages bitwise operations to iterate through each bit of the integer's binary representation. No additional data structures are necessary, and the algorithm follows these steps:

1. Initialize two variables: `ans` to store the longest distance found so far (initially set to zero), and `j` to keep track of the position of the last `1` encountered (initialized to -1 as no `1` has been encountered yet).
2. Loop 32 times, which corresponds to the maximum number of bits required to represent an integer in binary (as most modern architectures use 32 bits for standard integers).
3. For each bit position `i` from 0 to 31:
   - Check if the least significant bit of `n` is `1`. This is done using a bitwise AND operation (`n & 1`). If the result is not zero, the least significant bit is a `1`.
   - If `j` is not `-1` (which means we've found a previous `1`), calculate the distance from the current `1` to the previous `1` by subtracting `j` from `i` and update `ans` with the maximum distance found so far.
   - Set `j` to the current bit position `i`.
4. Right shift the bits of `n` (`n >>= 1`) to bring the next bit into the least significant bit position, preparing for the next iteration of the loop.
5. Repeat steps 3 and 4 until all 32 bit positions have been checked.
6. Once the loop is finished, the value stored in `ans` is the longest distance between two adjacent `1`s in the binary representation of `n`. Return `ans`.

In summary, the implementation uses an iteration over the bit positions and bitwise manipulation to identify and measure the distances between the `1`s. The pattern of maintaining a running maximum (`ans`) and tracking the last occurrence of interest (`j`) is common in problems where successive elements need to be compared or where distances need to be computed.

Example Walkthrough

Let's say we have the positive integer `n = 22`, whose binary representation is `10110`. We want to calculate the longest distance between any two adjacent `1`s in this binary. Now, let's walk through the solution step by step.

1. Initialize `ans` to `0` and `j` to `-1`.
2. Iterate from bit position `i = 0` to `31`. For `n = 22`, our binary is `0000...010110` (in 32-bit form).
3. At `i = 0`, `n & 1` is `0`. `n` is right-shifted: `n >>= 1`, so `n` becomes `11`.
4. At `i = 1`, `n & 1` is `1`, so we found the first `1`. Since `j` is `-1`, we just update `j = 1`.
5. At `i = 2`, `n & 1` is `1` again. Now, `j` is not `-1`, so we calculate the distance: `i - j` which is `2 - 1 = 1`. We update `ans` to `1` because `1` is larger than the current `ans` which is `0`. Now set `j = 2`.
6. At `i = 3`, `n & 1` is `0`. `n` is right-shifted.
7. At `i = 4`, `n & 1` is `0`. `n` is right-shifted.
8. At `i = 5`, `n & 1` is `1`. We calculate the distance from the previous `1`: `i - j` which is `5 - 2 = 3`. We update `ans` to `3` because `3` is larger than our current `ans` of `1`. Now set `j = 5`.
9. Continue this process until all bits are checked, but since `n` becomes `0` after several shifts, the rest of the iterations will not change `ans` or `j`.

After completing the loop, the value of `ans` is `3`. So the longest distance between two adjacent `1`s in the binary representation of `22` is `3`.

Solution Implementation

Python

```python
class Solution:
    def binary_gap(self, n: int) -> int:
        # Initialize maximum distance (gap) and the position of the last found '1'
        max_gap, last_pos = 0, -1

        # Loop through each bit position
        for i in range(32):
            # Check if the least significant bit is '1'
            if n & 1:
                # If this is not the first '1' found, calculate the gap
                if last_pos != -1:
                    gap = i - last_pos
                    max_gap = max(max_gap, gap)
                # Update the position of the last found '1'
                last_pos = i
            # Right shift 'n' to check the next bit
            n >>= 1

        # Return the maximum distance (gap) between two consecutive '1's in the binary representation
        return max_gap
```

Java

```java
class Solution {
    public int binaryGap(int number) {
        int maxGap = 0; // Maximum distance found so far
        // Index to track the rightmost 1 bit we have seen
        int lastSeenIndex = -1; // -1 indicates that we haven't seen a 1 bit yet

        // Iterate through each bit; 'i' is the position of the current bit, shifting 'number' to the right each time
        for (int currentPosition = 0; number != 0; currentPosition++, number >>= 1) {
            // Check if the least significant bit is 1
            if ((number & 1) == 1) {
                // If we have seen an earlier 1 bit, update the max gap
                if (lastSeenIndex != -1) {
                    maxGap = Math.max(maxGap, currentPosition - lastSeenIndex);
                }
                // Update the index of the last seen 1 bit
                lastSeenIndex = currentPosition;
            }
        }
        return maxGap; // Return the maximum distance between two 1 bits in the binary representation
    }
}
```

C++

```cpp
#include <algorithm> // Include the algorithm library for the max function

class Solution {
public:
    int binaryGap(int N) {
        int maxDistance = 0;  // The maximum distance between two consecutive 1's
        for (int currentPos = 0, lastOnePos = -1;
             N > 0;
             ++currentPos, N >>= 1) // Shift N right to process the next bit
        {
            if (N & 1) { // Check if the rightmost bit of N is set (is 1)
                if (lastOnePos != -1) { // There was a previous 1
                    // Update the maximum distance between 1's
                    maxDistance = std::max(maxDistance, currentPos - lastOnePos);
                }
                // Update the position of the last 1 encountered
                lastOnePos = currentPos;
            }
        }
        return maxDistance; // Return the maximum distance found
    }
};
```

TypeScript

```typescript
function binaryGap(n: number): number {
    let maxGap = 0; // This will hold the maximum distance between two consecutive 1's.
    let lastIndex = -1; // This will keep the index of the last 1 bit found.

    // Iterate over the bits of 'n'.
    // 'i' will serve as the bit position counter.
    for (let i = 0; n !== 0; i++) {
        // If the least significant bit is a 1...
        if (n & 1) {
            // If it's not the first 1 bit we've found...
            if (lastIndex !== -1) {
                // ...update the maxGap with the larger value between the current maxGap and the distance from the last 1 bit.
                maxGap = Math.max(maxGap, i - lastIndex);
            }
            // Update lastIndex to the current bit's position.
            lastIndex = i;
        }
        // Right shift 'n' by 1 bit to process the next bit during the next loop iteration.
        n >>= 1;
    }
    return maxGap; // Return the maximum gap found.
}
```


```python
class Solution:
    def binary_gap(self, n: int) -> int:
        # Initialize maximum distance (gap) and the position of the last found '1'
        max_gap, last_pos = 0, -1

        # Loop through each bit position
        for i in range(32):
            # Check if the least significant bit is '1'
            if n & 1:
                # If this is not the first '1' found, calculate the gap
                if last_pos != -1:
                    gap = i - last_pos
                    max_gap = max(max_gap, gap)
                # Update the position of the last found '1'
                last_pos = i
            # Right shift 'n' to check the next bit
            n >>= 1

        # Return the maximum distance (gap) between two consecutive '1's in the binary representation
        return max_gap
```


Time and Space Complexity

The given code snippet is designed to find the maximum distance between any two consecutive "1" bits in the binary representation of a given integer `n`.

Time Complexity:

The time complexity of the code is determined by the number of iterations in the for-loop which is fixed at 32 (since it's considering a 32-bit integer). Therefore, the loop runs a constant number of times, independent of the input size. As a result, the time complexity of the code is `O(1)`.

Inside the loop, all operations performed (checking if the least significant bit is 1, updating the maximum distance `ans`, shifting of `n`, and updating `j`) are done in constant time.

Given that the number of operations is fixed and limited by the size of the integer (32 bits in this case), it does not scale with `n`, thus confirming that the time complexity is constant.

Space Complexity:

The space complexity of the code is the amount of memory it uses in addition to the input. The code uses a constant amount of extra space for the variables `ans`, `j`, and `i`. These variables are independent of the input size since their size does not grow with `n`.

Hence, the space complexity of the code is `O(1)` as well, because it allocates a fixed amount of space that does not vary with the size of the input `n`.