

Bit Manipulation

1. Number of 1 Bits

Pattern: Bit Manipulation

Problem Statement

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Note:

- In some languages (like Java), there is no unsigned integer type. In this case, the input will be given as a signed integer type.
 - The input must be treated as a **32-bit** binary representation, regardless of language-specific signedness.
-

Sample Input & Output

```
Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string has three '1' bits.
```

```
Input: n = 000000000000000000000000010000000
Output: 1
Explanation: Only one '1' bit is present at the 8th position from right.
```

Input: n = 11111111111111111111111111111101

Output: 31

Explanation: All bits are '1' except the second least significant bit.

LeetCode Editorial Solution + Inline Tests

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        # STEP 1: Initialize count to track number of 1 bits
        # - We'll inspect each bit via bitwise operations.
        count = 0

        # STEP 2: Loop while n is not zero
        # - Each iteration removes the lowest set bit.
        # - This avoids checking all 32 bits unnecessarily.
        while n:
            # STEP 3: Remove the lowest set bit using n & (n - 1)
            # - n & (n - 1) flips the least significant 1-bit to 0.
            # - Example: 1011 & 1010 = 1010
            n &= n - 1
            count += 1

        # STEP 4: Return total count of 1 bits
        # - Works for all 32-bit inputs, including negative in Python
        # - due to two's complement handling in bitwise ops.
        return count

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.hammingWeight(0b00000000000000000000000001011) == 3

    # Test 2: Edge case - single 1 bit
    assert sol.hammingWeight(0b00000000000000000000000010000000) == 1

    # Test 3: Tricky/negative - 31 ones (treated as 32-bit)
```

```
# Python uses arbitrary precision, but problem assumes 32-bit.
# We simulate 32-bit unsigned by masking if needed, but LeetCode
# handles input as if it's 32-bit. The bit trick works regardless.
assert sol.hammingWeight(0b111111111111111111111111111101) == 31

print(" All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `hammingWeight(0b1011)` (which is 11 in decimal):

1. **Initial state:**

- `n = 0b1011` (binary) → 11 (decimal)
- `count = 0`

2. **First loop iteration:**

- `n != 0` → enter loop
- Compute `n - 1 = 0b1010`
- `n = n & (n - 1) = 0b1011 & 0b1010 = 0b1010`
- `count = 1`

3. **Second loop iteration:**

- `n = 0b1010` 0
- `n - 1 = 0b1001`
- `n = 0b1010 & 0b1001 = 0b1000`
- `count = 2`

4. **Third loop iteration:**

- `n = 0b1000` `0`
- `n - 1 = 0b0111`
- `n = 0b1000 & 0b0111 = 0b0000`
- `count = 3`

5. **Loop ends:**

- `n = 0` \rightarrow exit loop
- Return `count = 3`

Final output: 3 — matches expected result.

Key insight: Each `n &= n - 1` removes exactly one 1 bit. So the number of iterations = number of 1s.

Complexity Analysis

- **Time Complexity:** $O(k)$ where `k` = number of 1 bits

In worst case (all 32 bits are 1), it's $O(32) = O(1)$.

But more precisely, it runs once per set bit — very efficient for sparse bits.

- **Space Complexity:** $O(1)$

Only uses a constant amount of extra space (`count` and temporary `n`).

No recursion, no auxiliary data structures.

2. Counting Bits

Pattern: Bit Manipulation + Dynamic Programming

Problem Statement

Given an integer `n`, return an array `ans` of length `n + 1` such that for each `i` ($0 \leq i \leq n$), `ans[i]` is the number of 1's in the binary representation of `i`.

Sample Input & Output

```
Input: n = 2
Output: [0, 1, 1]
Explanation:
0 → "0" → 0 ones
1 → "1" → 1 one
2 → "10" → 1 one
```

```
Input: n = 5
Output: [0, 1, 1, 2, 1, 2]
Explanation:
3 → "11" → 2 ones, 4 → "100" → 1 one, 5 → "101" → 2 ones
```

```
Input: n = 0
Output: [0]
Explanation: Only i = 0; binary "0" has zero 1s.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def countBits(self, n: int) -> List[int]:
        # STEP 1: Initialize result array with zeros
        # - Length is n + 1 to include 0 through n
        # - ans[0] = 0 because 0 has zero 1-bits
        ans = [0] * (n + 1)
```

```

# STEP 2: Main loop from 1 to n (inclusive)
#   - Use recurrence: ans[i] = ans[i >> 1] + (i & 1)
#   - i >> 1 = i // 2 (remove last bit)
#   - i & 1 = 1 if last bit is 1, else 0
#   - This leverages previously computed subproblems
for i in range(1, n + 1):
    ans[i] = ans[i >> 1] + (i & 1)

# STEP 3: No extra bookkeeping needed - DP fills array
#   - Each entry built from smaller index → safe

# STEP 4: Return full array
return ans

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    result1 = sol.countBits(5)
    print(f"Test 1 (n=5): {result1}") # Expected: [0,1,1,2,1,2]

    # Test 2: Edge case
    result2 = sol.countBits(0)
    print(f"Test 2 (n=0): {result2}") # Expected: [0]

    # Test 3: Tricky/negative (larger n)
    result3 = sol.countBits(8)
    print(f"Test 3 (n=8): {result3}") # Expected: [0,1,1,2,1,2,2,3,1]

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `countBits(5)` step by step:

1. Initialize `ans = [0, 0, 0, 0, 0, 0]`

- Array of length 6 (since $n=5 \rightarrow$ indices 0–5).
- `ans[0] = 0` is correct (binary “0” has no 1s).

2. Loop starts at `i = 1`

- `i >> 1 = 0, i & 1 = 1 \rightarrow ans[1] = ans[0] + 1 = 0 + 1 = 1`
- `ans = [0, 1, 0, 0, 0, 0]`

3. `i = 2`

- `2 >> 1 = 1, 2 & 1 = 0 \rightarrow ans[2] = ans[1] + 0 = 1 + 0 = 1`
- `ans = [0, 1, 1, 0, 0, 0]`

4. `i = 3`

- `3 >> 1 = 1, 3 & 1 = 1 \rightarrow ans[3] = ans[1] + 1 = 1 + 1 = 2`
- `ans = [0, 1, 1, 2, 0, 0]`

5. `i = 4`

- `4 >> 1 = 2, 4 & 1 = 0 \rightarrow ans[4] = ans[2] + 0 = 1 + 0 = 1`
- `ans = [0, 1, 1, 2, 1, 0]`

6. `i = 5`

- `5 >> 1 = 2, 5 & 1 = 1 \rightarrow ans[5] = ans[2] + 1 = 1 + 1 = 2`
- `ans = [0, 1, 1, 2, 1, 2]`

Final output: `[0, 1, 1, 2, 1, 2]` — matches expected.

Key Insight:

Every number `i` can be thought of as `(i // 2)` with one extra bit (the least significant bit).

So the count of 1s in `i` = count in `i//2` + (1 if LSB is 1, else 0).

This is a classic **DP with bit observation**.

Complexity Analysis

- **Time Complexity:** $O(n)$

Single loop from 1 to `n`. Each iteration does $O(1)$ bit operations (`>>` and `&`).

- **Space Complexity:** $O(1)$ auxiliary, $O(n)$ total

We must return an array of size $n+1$, so output space is $O(n)$.

No extra space beyond the result array — thus **$O(1)$ extra space**.

3. Reverse Bits

Pattern: Bit Manipulation

Problem Statement

Reverse bits of a given 32-bit unsigned integer.

Note:

- In some languages (e.g., Java), there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the internal binary representation remains the same whether signed or unsigned.
- In Java, the compiler represents signed integers using 2's complement notation.

Follow up: If this function is called many times, how would you optimize it?

Sample Input & Output

```
Input: n = 00000010100101000001111010011100
Output:  964176192 (00111001011110000010100101000000)
Explanation: The input binary string represents 43261596.
               Reversing its bits gives 964176192.
```

```
Input: n = 11111111111111111111111111111101
Output:  3221225471 (10111111111111111111111111111111)
Explanation: Input is 4294967293 (signed: -3).
               Reversed bits yield 3221225471.
```


Input: n = 00000000000000000000000000000000
Output: 0
Explanation: All bits are zero; reversal remains zero.

LeetCode Editorial Solution + Inline Tests

```
class Solution:
    def reverseBits(self, n: int) -> int:
        # STEP 1: Initialize result to 0
        # - This will accumulate the reversed bits

        # STEP 2: Iterate 32 times (for 32-bit integer)
        # - In each iteration:
        #     a) Shift result left to make room for next bit
        #     b) Extract LSB of n using n & 1
        #     c) Add that bit to result
        #     d) Shift n right to process next bit

        # STEP 3: After 32 iterations, result holds reversed bits

        # STEP 4: Return result (already correct as unsigned 32-bit)
        result = 0
        for _ in range(32):
            result = (result << 1) | (n & 1)
            n >>= 1
        return result

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.reverseBits(43261596) == 964176192

    # Test 2: Edge case (all ones except last)
    assert sol.reverseBits(4294967293) == 3221225471

    # Test 3: Tricky/negative (zero input)
```

```
assert sol.reverseBits(0) == 0

print(" All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `reverseBits(6)` assuming **4-bit** for simplicity (real code uses 32):
Binary of 6 → 0110, reversed → 0110 → 0110 reversed is 0110? Wait—no!
Actually: 0110 (4-bit) → reversed → 0110 → becomes 0110? That's confusing.

Better: use **8-bit** example: $n = 6 \rightarrow 00000110 \rightarrow \text{reversed} \rightarrow 01100000 = 96$.

But our code uses **32 bits**, so let's walk through **first 4 iterations** of $n = 6$:

Initial:

```
- n = 6 → binary: ...00000110
- result = 0
```

Iteration 1:

```
- result << 1 → 0
- n & 1 → 6 & 1 = 0
- result = 0 | 0 = 0
- n >>= 1 → n = 3 (...00000011)
```

Iteration 2:

```
- result = 0 << 1 = 0
- n & 1 = 3 & 1 = 1
- result = 0 | 1 = 1
- n = 3 >> 1 = 1
```

Iteration 3:

```
- result = 1 << 1 = 2
- n & 1 = 1 & 1 = 1
- result = 2 | 1 = 3
- n = 0
```

Iteration 4:

```
- result = 3 << 1 = 6
- n & 1 = 0
```

```
- result = 6 | 0 = 6
- n = 0
```

... and so on for 32 iterations. After 32 steps, the original LSBs become MSBs in **result**.

For $n = 43261596$ (binary ends with $\dots 1100$), the first extracted bits $(0, 0, 1, 1, \dots)$ become the **most significant** bits in the output — hence reversal.

Final **result** after 32 shifts and ORs = bit-reversed integer.

Complexity Analysis

- **Time Complexity:** $O(1)$

We always loop exactly 32 times — constant time, independent of input value.

- **Space Complexity:** $O(1)$

Only use a few integer variables (**result**, loop counter). No extra space scales with input.

4. Single Number

Pattern: Arrays & Hashing \rightarrow Bit Manipulation (XOR)

Problem Statement

Given a **non-empty** array of integers **nums**, every element appears **twice** except for one. Find that single one.

You must implement a solution with **linear runtime complexity** and use **only constant extra space**.

Sample Input & Output

Input: [2,2,1]
Output: 1
Explanation: 2 appears twice, 1 appears once → answer is 1.

Input: [4,1,2,1,2]
Output: 4
Explanation: 1 and 2 appear twice; 4 is unique.

Input: [1]
Output: 1
Explanation: Only one element → it is the answer (edge case).

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        # STEP 1: Initialize result to 0
        # - XOR identity:  $a \oplus 0 = a$ 
        # - We'll accumulate XOR of all numbers

        result = 0

        # STEP 2: Main loop over all numbers
        # - XOR has key properties:
        #    $a \oplus a = 0$  (duplicates cancel)
        #    $a \oplus b \oplus a = b$  (order doesn't matter)
        # - After processing all, only unique remains

        for num in nums:
            result ^= num

        # STEP 3: No extra bookkeeping needed
        # - XOR inherently handles cancellation

        # STEP 4: Return result
```

```

        # - Works even for single-element input

        return result

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.singleNumber([2, 2, 1]) == 1

    # Test 2: Edge case (single element)
    assert sol.singleNumber([1]) == 1

    # Test 3: Tricky/negative (multiple pairs + unique)
    assert sol.singleNumber([4, 1, 2, 1, 2]) == 4

    print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `singleNumber([4, 1, 2, 1, 2])` step by step.

Initial state:

- `nums = [4, 1, 2, 1, 2]`
- `result = 0`

Step 1: Process `num = 4`

- Execute: `result ^= 4` → $0 \oplus 4 = 4$
- State: `result = 4`

Step 2: Process `num = 1`

- Execute: `result ^= 1` → $4 \oplus 1 = 5$ (binary: $100 \oplus 001 = 101$)
- State: `result = 5`

Step 3: Process `num = 2`

- Execute: `result ^= 2` $\rightarrow 5 \oplus 2 = 7$ ($101 \oplus 010 = 111$)
- State: `result = 7`

Step 4: Process `num = 1`

- Execute: `result ^= 1` $\rightarrow 7 \oplus 1 = 6$ ($111 \oplus 001 = 110$)
- State: `result = 6`

Step 5: Process `num = 2`

- Execute: `result ^= 2` $\rightarrow 6 \oplus 2 = 4$ ($110 \oplus 010 = 100$)
- State: `result = 4`

Final return: 4

Why it works:

- The two 1s cancel: $1 \oplus 1 = 0$
- The two 2s cancel: $2 \oplus 2 = 0$
- Left with $4 \oplus 0 \oplus 0 = 4$

Complexity Analysis

- **Time Complexity:** $O(n)$

One pass through the array $\rightarrow n$ iterations, each with $O(1)$ XOR operation.

- **Space Complexity:** $O(1)$

Only one integer variable (`result`) used \rightarrow constant extra space.

5. Missing Number

Pattern: Arrays & Hashing (Math / XOR / Sum Trick)

Problem Statement

Given an array `nums` containing `n` distinct numbers in the range $[0, n]$, return the only number in the range that is missing from the array.

Sample Input & Output

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers. The full set is [0,1,2,3]; 2 is missing.

Input: nums = [0,1]

Output: 2

Explanation: n = 2; full set [0,1,2]; 2 is missing.

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation: Only 8 is missing from [0..9].

LeetCode Editorial Solution + Inline Tests

```
from typing import List
```

```
class Solution:
```

```
    def missingNumber(self, nums: List[int]) -> int:
```

```
        # STEP 1: Initialize expected sum using formula  $n*(n+1)/2$ 
```

```
        # - Since numbers are 0 to n (inclusive), total count = n+1
```

```
        # - But input has only n elements → missing one
```

```
        n = len(nums)
```

```
        expected_sum = n * (n + 1) // 2
```

```
        # STEP 2: Compute actual sum of given array
```

```
        # - Sum all elements present
```

```
        actual_sum = sum(nums)
```

```
        # STEP 3: Missing number = expected - actual
```

```
        # - This works because all numbers are distinct and in range
```

```
        return expected_sum - actual_sum
```

```
# ----- INLINE TESTS -----
```

```
if __name__ == "__main__":
```

```
sol = Solution()

# Test 1: Normal case
assert sol.missingNumber([3, 0, 1]) == 2

# Test 2: Edge case - missing last number
assert sol.missingNumber([0, 1]) == 2

# Test 3: Tricky/negative - large shuffled input
assert sol.missingNumber([9,6,4,2,3,5,7,0,1]) == 8

print(" All tests passed!")
```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `missingNumber([3, 0, 1])` step by step:

1. **Line:** `n = len(nums)`
 - `nums = [3, 0, 1] → length = 3`
 - `n = 3`
2. **Line:** `expected_sum = n * (n + 1) // 2`
 - Compute sum of 0 to 3: $3 * 4 // 2 = 6$
 - `expected_sum = 6`
3. **Line:** `actual_sum = sum(nums)`
 - Sum of `[3, 0, 1]` = $3 + 0 + 1 = 4$
 - `actual_sum = 4`
4. **Line:** `return expected_sum - actual_sum`
 - $6 - 4 = 2$
 - Returns 2

Final Output: 2

Why it works: The full sequence `[0,1,2,3]` sums to 6. The input sums to 4. The difference is the missing number.

Complexity Analysis

- **Time Complexity:** $O(n)$

We iterate once through the array to compute `sum(nums)`. All other operations are $O(1)$.

- **Space Complexity:** $O(1)$

Only a few integer variables are used — no extra space proportional to input size.

6. Find the Duplicate Number

Pattern: Floyd's Tortoise and Hare (Cycle Detection)

Problem Statement

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`. Return this repeated number.

You must solve the problem **without modifying** the array `nums` and using only **constant extra space**.

Sample Input & Output

```
Input: nums = [1,3,4,2,2]
Output: 2
Explanation: The number 2 appears twice.
```

```
Input: nums = [3,1,3,4,2]
Output: 3
Explanation: The number 3 appears twice.
```

```
Input: nums = [1,1]
Output: 1
Explanation: Only two elements; both are 1 - edge case with minimal size.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers
        # - Treat array as a linked list where index -> value = next pointer
        # - Because values are in [1, n], they point to valid indices
        slow = fast = nums[0]

        # STEP 2: Phase 1 - Detect cycle using Floyd's algorithm
        # - Move slow by 1 step, fast by 2 steps
        # - They will meet inside the cycle due to pigeonhole principle
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        # STEP 3: Phase 2 - Find entrance to cycle (duplicate number)
        # - Reset one pointer to start; move both at same speed
```

```

# - Meeting point is the duplicate (cycle entrance)
slow = nums[0]
while slow != fast:
    slow = nums[slow]
    fast = nums[fast]

# STEP 4: Return result
# - Guaranteed to find duplicate due to problem constraints
return slow

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.findDuplicate([1,3,4,2,2]) == 2

    # Test 2: Edge case - minimal input
    assert sol.findDuplicate([1,1]) == 1

    # Test 3: Tricky/negative - duplicate not at end
    assert sol.findDuplicate([3,1,3,4,2]) == 3

    print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `nums = [1,3,4,2,2]` step by step.

Initial state:

- `nums = [1, 3, 4, 2, 2]` (indices: $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow \dots$) - Think of it as a linked list:
 $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 4 \rightarrow \dots \rightarrow$ cycle starts at **2**

Phase 1: Detect cycle

- `slow = nums[0] = 1`
- `fast = nums[0] = 1`

Iteration 1:

- `slow = nums[1] = 3`
- `fast = nums[nums[1]] = nums[3] = 2`
- Now: `slow=3, fast=2`

Iteration 2:

- `slow = nums[3] = 2`
- `fast = nums[nums[2]] = nums[4] = 2`
- Now: `slow=2, fast=2` → **they meet!** Break loop.

Phase 2: Find cycle entrance (duplicate)

- Reset `slow = nums[0] = 1`
- Keep `fast = 2` (meeting point)

Iteration 1:

- `slow = nums[1] = 3`
- `fast = nums[2] = 4`
- Not equal → continue

Iteration 2:

- `slow = nums[3] = 2`
- `fast = nums[4] = 2`
- Now `slow == fast == 2` → **found duplicate!**

Return 2.

Final output: 2

Key insight: The duplicate creates a cycle because two indices point to the same value (e.g., both index 3 and 4 point to value 2 → two “incoming edges” to node 2).

Complexity Analysis

- **Time Complexity:** $O(n)$

We traverse the array at most twice: once to detect the cycle, once to find the entrance. Each step moves pointers forward, so linear in n .

- **Space Complexity:** $O(1)$

Only two integer pointers (**slow**, **fast**) are used — no extra arrays, hash maps, or recursion stack. Meets constant space requirement.

6. Find the Duplicate Number

Pattern: Floyd's Tortoise and Hare (Cycle Detection)

Problem Statement

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range $[1, n]$ inclusive.

There is only **one repeated number** in `nums`. Return this repeated number.

You must solve the problem **without modifying** the array `nums` and using only **constant extra space**.

Sample Input & Output

```
Input: nums = [1,3,4,2,2]
```

```
Output: 2
```

```
Explanation: The number 2 appears twice.
```

```
Input: nums = [3,1,3,4,2]
```

```
Output: 3
```

```
Explanation: The number 3 appears twice.
```

Input: nums = [1,1]

Output: 1

Explanation: Only two elements; both are 1 - edge case with minimal size.

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        # STEP 1: Initialize pointers
        #   - Treat array as a linked list where index -> value = next pointer
        #   - Because values are in [1, n], they point to valid indices
        slow = fast = nums[0]

        # STEP 2: Phase 1 - Detect cycle using Floyd's algorithm
        #   - Move slow by 1 step, fast by 2 steps
        #   - They will meet inside the cycle due to pigeonhole principle
        while True:
            slow = nums[slow]
            fast = nums[nums[fast]]
            if slow == fast:
                break

        # STEP 3: Phase 2 - Find entrance to cycle (duplicate number)
        #   - Reset one pointer to start; move both at same speed
        #   - Meeting point is the duplicate (cycle entrance)
        slow = nums[0]
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]

        # STEP 4: Return result
        #   - Guaranteed to find duplicate due to problem constraints
        return slow

# ----- INLINE TESTS -----
if __name__ == "__main__":
```

```

sol = Solution()

# Test 1: Normal case
assert sol.findDuplicate([1,3,4,2,2]) == 2

# Test 2: Edge case - minimal input
assert sol.findDuplicate([1,1]) == 1

# Test 3: Tricky/negative - duplicate not at end
assert sol.findDuplicate([3,1,3,4,2]) == 3

print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `nums = [1,3,4,2,2]` step by step.

Initial state:

- `nums = [1, 3, 4, 2, 2]` (indices: $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow \dots$) - Think of it as a linked list:
 $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 4 \rightarrow \dots \rightarrow$ cycle starts at **2**

Phase 1: Detect cycle

- `slow = nums[0] = 1`
- `fast = nums[0] = 1`

Iteration 1:

- `slow = nums[1] = 3`
- `fast = nums[nums[1]] = nums[3] = 2`
- Now: `slow=3, fast=2`

Iteration 2:

- `slow = nums[3] = 2`
- `fast = nums[nums[2]] = nums[4] = 2`
- Now: `slow=2, fast=2` → **they meet!** Break loop.

Phase 2: Find cycle entrance (duplicate)

- Reset `slow = nums[0] = 1`
- Keep `fast = 2` (meeting point)

Iteration 1:

- `slow = nums[1] = 3`
- `fast = nums[2] = 4`
- Not equal \rightarrow continue

Iteration 2:

- `slow = nums[3] = 2`
- `fast = nums[4] = 2`
- Now `slow == fast == 2` \rightarrow **found duplicate!**

Return 2.

Final output: 2

Key insight: The duplicate creates a cycle because two indices point to the same value (e.g., both index 3 and 4 point to value 2 \rightarrow two “incoming edges” to node 2).

Complexity Analysis

- **Time Complexity:** $O(n)$

We traverse the array at most twice: once to detect the cycle, once to find the entrance. Each step moves pointers forward, so linear in `n`.

- **Space Complexity:** $O(1)$

Only two integer pointers (`slow`, `fast`) are used — no extra arrays, hash maps, or recursion stack. Meets constant space requirement.

7. Add Binary

Pattern: String Manipulation + Simulation

Problem Statement

Given two binary strings **a** and **b**, return their sum as a binary string.

The input strings contain only '0' and '1' characters.

The result must not have leading zeros, except for the number "0" itself.

Sample Input & Output

Input: a = "11", b = "1"

Output: "100"

Explanation: $3 + 1 = 4 \rightarrow$ "100" in binary.

Input: a = "1010", b = "1011"

Output: "10101"

Explanation: $10 + 11 = 21 \rightarrow$ "10101" in binary.

Input: a = "0", b = "0"

Output: "0"

Explanation: Edge case - both inputs are zero.

LeetCode Editorial Solution + Inline Tests

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        # STEP 1: Initialize structures
        #   - Use two pointers starting from the end of each string
        #   - Track carry for overflow from bit addition
        i, j = len(a) - 1, len(b) - 1
        carry = 0
        result = []

        # STEP 2: Main loop / recursion
        #   - Process bits from least significant to most
```

```

# - Continue until both strings and carry are exhausted
while i >= 0 or j >= 0 or carry:
    total = carry

    # Add bit from string a if available
    if i >= 0:
        total += int(a[i])
        i -= 1

    # Add bit from string b if available
    if j >= 0:
        total += int(b[j])
        j -= 1

    # STEP 3: Update state / bookkeeping
    # - Current bit is total % 2
    # - New carry is total // 2
    result.append(str(total % 2))
    carry = total // 2

# STEP 4: Return result
# - Reverse because we built it backwards
return ''.join(reversed(result))

# ----- INLINE TESTS -----
if __name__ == "__main__":
    sol = Solution()

    # Test 1: Normal case
    assert sol.addBinary("11", "1") == "100"

    # Test 2: Edge case
    assert sol.addBinary("0", "0") == "0"

    # Test 3: Tricky/negative
    assert sol.addBinary("1010", "1011") == "10101"

    print(" All tests passed!")

```

How to use: Copy-paste this block into .py or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `addBinary("11", "1")` step by step:

1. **Initialize:**

- `i = 1` (last index of "11"), `j = 0` (last index of "1")
- `carry = 0`, `result = []`

2. **First loop iteration** (`i=1, j=0, carry=0`):

- `total = 0`
- Add `a[1] = '1' → total = 1, i = 0`
- Add `b[0] = '1' → total = 2, j = -1`
- Append `2 % 2 = 0 → result = ['0']`
- `carry = 2 // 2 = 1`

3. **Second loop iteration** (`i=0, j=-1, carry=1`):

- `total = 1`
- Add `a[0] = '1' → total = 2, i = -1`
- `j < 0`, skip
- Append `2 % 2 = 0 → result = ['0', '0']`
- `carry = 1`

4. **Third loop iteration** (`i=-1, j=-1, carry=1`):

- `total = 1`
- No more bits in `a` or `b`
- Append `1 % 2 = 1 → result = ['0', '0', '1']`

- `carry = 0`

5. **Loop ends** (all indices < 0 and `carry = 0`)

6. **Reverse result**: `['1', '0', '0']` \rightarrow `"100"`

Final output: `"100"`

Complexity Analysis

- **Time Complexity**: $O(\max(m, n))$

We iterate once over the longer of the two input strings ($m = \text{len}(a)$, $n = \text{len}(b)$). Each operation inside the loop is $O(1)$.

- **Space Complexity**: $O(\max(m, n))$

The `result` list stores at most $\max(m, n) + 1$ characters (for final carry). Reversal and join use linear space.