Problem Description

The problem requires writing a function that calculates the integer square root of a given non-negative integer x. The integer square root of x is the largest integer y such that y*y is less than or equal to x. It's important to note that the function should return the floor of the square root, which means it should be rounded down to the nearest integer. For example, the integer square root of 8 is 2, because 2*2=4 is less than 8 and 3*3=9 is more than 8.

Additionally, the restrictions of the problem state that you cannot use any built-in exponent functions or operators that would directly calculate the square root. This means standard functions like pow in C++ or the exponent operator ** in Python are not allowed.

Intuition

The intuition behind the provided solution is based on using the binary search algorithm. Binary search is a technique used to find an element in a sorted array by repeatedly dividing the search interval in half. The principle of this algorithm can be applied to finding the square root since the square root of a number x will always be less than or equal to x. The algorithm starts by setting left to 0 and right to x, establishing the range within which the square root must lie. We then

enter a loop to continually narrow this range. In each iteration, we calculate a mid-point (mid) and check if mid*mid is less than or equal to x. If it is, we move the left bound up to mid since we know that the square root is at least mid. If mid*mid is greater than x, we set the right bound to mid - 1 because the square root must be smaller than mid. The loop continues until left and right converge to the same value, which will be the largest integer less than or equal to the square root of x.

By avoiding multiplication (which could cause overflow) and using integer division instead (the // operator in Python), the solution ensures that it works correctly for large integers. The use of the bitwise shift >> 1 is an efficient way to calculate mid by dividing the sum of left and right by 2 without using floating-point arithmetic, which could introduce errors in some environments due to rounding.

The solution approach applies the binary search algorithm to find the square root of a given non-negative integer x. Here's a

within which we'll be searching for the square root.

Solution Approach

step-by-step breakdown of how the algorithm is implemented: Initialization: Start by initializing two pointers, left at 0 and right at x. These pointers represent the bounds of the range

- Binary Search Loop: Enter a while loop that continues as long as left is less than right. This loop will progressively narrow down the range to zero in on the correct square root by adjusting left and right.
- Midpoint Calculation: In each iteration of the loop, calculate the midpoint mid using the expression (left + right + 1) >> 1. The use of bitwise shift >> efficiently divides the sum by 2.
- Square Root Check: Determine if mid is a valid square root by comparing mid * mid with x. However, we prevent potential overflow by comparing mid with x // mid instead, which achieves the same result using integer division.

Adjusting Bounds: If mid <= x // mid, it means that mid could be the square root, or the true square root could be bigger.

- Therefore, move left up to mid. If mid > x // mid, then mid is too large, so bring right down to mid 1. Convergence: The loop continues until left equals right, which means we have found the largest integer y such that y*y is
- less than or equal to x. At this point, the binary search has zeroed in on the exact floor value of the square root. **Return Value:** Exit the loop and return left, which now holds the floor of the square root of x.

The algorithm effectively uses a search pattern to find the exact point at which the square of a number shifts from being less

dramatically reduces the number of checks needed, making the algorithm efficient even for very large values of x.

than or equal to x, to being greater than it. Instead of checking every number up to x, which would be inefficient, binary search

Example Walkthrough Let's illustrate the solution approach by calculating the integer square root of the number 10. We want to find the largest integer y

Initialization: We initialize left to 0 and right to 10.

such that y*y is less than or equal to 10.

Binary Search Loop: We enter the while loop since left (0) is less than right (10). Midpoint Calculation: We calculate mid using (left + right + 1) >> 1, which is (0 + 10 + 1) >> 1 = 11 >> 1 = 5 (as 11 >>

Square Root Check: Next, we compare mid * mid to 10. Here, 5 * 5 equals 25, which is greater than 10. To avoid

1 means dividing 11 by 2 and taking the floor of the result).

- multiplication, we could compare mid to 10 // mid, where 5 is greater than 10 // 5 (which equals 2), confirming our result without risking overflow.
- **Loop Continuation:** We continue the loop with left still at 0 and right now at 4. **New Midpoint:** We calculate the new mid as (0 + 4 + 1) >> 1 = 5 >> 1 = 2.
- Second Square Root Check: We compare mid * mid with 10 again. Now 2 * 2 equals 4, which is less than 10. Comparing mid with 10 // mid, 2 is also less than 10 // 2 (5), so we move left up to mid.

Adjusting Bounds: Since mid (5) is greater than 10 // mid (2), we set right to mid - 1, which becomes 4.

- Adjusting Bounds Again: Now, left becomes 2, and right remains at 4.
- **Third Midpoint**: Recalculate the midpoint as $(2 + 4 + 1) \gg 1 = 7 \gg 1 = 3$. Third Square Root Check: We have 3 * 3 = 9, which is less than 10. Checking mid against 10 // mid gives 3 less than 10 //
- **Loop Ends**: Since mid (3) still produces a result that is less than x (10), we would repeat steps 3-5 until left equals right.

Initialise the search boundaries for binary search

mid = (left_boundary + right_boundary + 1) >> 1

return left; // Return the calculated square root

// Initialize left and right pointers for binary search.

// Perform binary search to find the square root of x.

// Calculate the mid-point, with a slight adjustment

// so move the right pointer to just before mid.

long long mid = left + ((right - left + 1) >> 1);

// so move the left pointer to mid.

// by adding 1 before shifting to ensure the mid always rounds up.

// Otherwise, the true square root must be smaller than mid,

// If mid squared is less than or equal to x, it is a valid potential square root,

Move the left boundary to the middle value

If the square of the middle value is less than or equal to x

3 (3), so we could consider moving left up to mid.

- next midpoint calculation after left = 3 and right = 4 would again be 3, and no further adjustments would be made.
- 13. Return Value: The loop will exit when left equals right, which will be the value 3 in this case. Since 3*3 is 9 and 4*4 is 16 (which is too big), 3 is indeed the largest integer whose square is less than or equal to 10. Therefore, the function returns 3 as the floor of the square root of 10.

At some point in the process, left and right will converge. Given our bounds adjustment pattern, they will meet at 3, since the

Python

The +1 ensures that we do not get stuck in an infinite loop with left_boundary and right_boundary equal

Perform binary search while left_boundary < right_boundary:</pre> # Use bitwise shifting to efficiently divide by 2; equivalent to mid = (left_boundary + right_boundary + 1) // 2

else:

def mySqrt(self, x: int) -> int:

if mid <= x // mid:</pre>

left_boundary, right_boundary = 0, x

left_boundary = mid

class Solution:

Solution Implementation

```
# Move the right boundary just before the middle value
               right_boundary = mid - 1
       # The left_boundary variable at the end of loop will hold the largest integer whose square is less than or equal to x
       return left_boundary
Java
class Solution {
   public int mySqrt(int x) {
       int left = 0;  // Initialize the left boundary of the search space
       int right = x;  // Initialize the right boundary of the search space
       while (left < right) { // Loop until the search space is narrowed down to one element</pre>
           int mid = (left + right + 1) >>> 1; // Compute the middle point, using unsigned right shift for safe division by 2
           if (mid \ll x / mid) \{ // If the square of mid is less than or equal to x
               left = mid;
                                  // Move the left boundary to mid, as mid is a potential solution
           } else {
               right = mid - 1; // Otherwise, discard mid and the right search space
       // The loop exits when left == right, which will be the largest integer less than or equal to the sqrt(x)
```

C++

public:

class Solution {

int mySqrt(int x) {

long long left = 0, right = x;

if (mid <= x / mid) {

right = mid - 1;

left = mid;

} else {

} else {

return left;

// Example of usage:

while (left < right) {</pre>

```
// Once left meets right, we've found the largest integer whose square is less than or equal to x_{ullet}
        return static_cast<int>(left);
};
TypeScript
/**
 * Calculates the square root of a given number using binary search.
 * @param {number} num The input number to calculate the square root for.
 * @return {number} The floor value of the square root of the input number.
function mySqrt(num: number): number {
    let left: number = 0; // Define the lower boundary of the search range
    let right: number = num; // Define the upper boundary of the search range
    // Continue looping until left pointer meets right pointer
    while (left < right) {</pre>
       // Using >>> 1 operates like Math.floor((left + right) / 2) but is faster
        const mid: number = (left + right + 1) >>> 1;
       // Check if the middle element squared is less than or equal to 'num'
       if (mid <= num / mid) {</pre>
```

```
class Solution:
   def mySqrt(self, x: int) -> int:
       # Initialise the search boundaries for binary search
        left_boundary, right_boundary = 0, x
```

console.log(mySqrt(10)); // Output should be 3

while left_boundary < right_boundary:</pre>

// When left meets right, we've found the integer part of the square root

mid = (left_boundary + right_boundary + 1) >> 1 # If the square of the middle value is less than or equal to x if mid <= x // mid:</pre> # Move the left boundary to the middle value left_boundary = mid else: # Move the right boundary just before the middle value right_boundary = mid - 1

left = mid; // If mid squared is within bounds, shift the left pointer to mid

right = mid - 1; // If mid squared is too large, shift the right pointer below mid

Time and Space Complexity

Perform binary search

The left_boundary variable at the end of loop will hold the largest integer whose square is less than or equal to x return left boundary

The provided code implements a binary search algorithm to find the integer square root of a number x.

Time Complexity The time complexity of this code is $0(\log n)$, where n is the value of the input x. This is because with each iteration of the while

loop, the search range is reduced by half, following the principle of binary search.

Use bitwise shifting to efficiently divide by 2; equivalent to mid = (left_boundary + right_boundary + 1) // 2

The +1 ensures that we do not get stuck in an infinite loop with left_boundary and right_boundary equal

Space Complexity

The space complexity of the code is 0(1). The algorithm only uses a constant amount of extra space to store variables like left, right, and mid, which do not depend on the size of the input.