Explanation of Midpoint Calculation in Binary Search

In the binary search algorithm, calculating the midpoint using left + (right - left) // 2 is important to avoid potential overflow issues, especially when dealing with large arrays or large index values. This practice ensures compatibility with languages that do not handle integer overflow gracefully.

Example for Practical Understanding

Suppose we are working within a hypothetical range where the maximum integer value (max_int) is 5, meaning we cannot store integers greater than range 5.

1. **Incorrect Calculation ((left + right) // 2)**:

   - If left = 4 and right = 4, using (left + right) // 2 results in:

```
(4 + 4) // 2 = 8 // 2 = 4
```

   - Here, 8 // 2 equals 4, which appears correct initially. However, in scenarios where left and right are at their maximum (max_int), adding them (4 + 4) could lead to an integer value (8) that exceeds the maximum allowed value (max_int), causing an overflow.
   - If the sum is 8 and we can store a maximum of 5, the sequence will be 1, 2, 3, 4, 5, and then it will repeat: 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; and so on. This implies that although the sum is 8, it effectively becomes 3 due to the storage limit. Moreover, it can even become -3, which will affect our calculations.

Actual Value of sum	1	2	3	4	5	6	7	8	9	10
What value it will be treated	1	2	3	4	5	1	2	3	4	5

2. **Correct Calculation (`left + (right - left) // 2`)**:

   - Using left + (right - left) // 2 ensures:

```
left + (4 - 4) // 2 = 4 + 0 // 2 = 4 + 0 = 4
```

   - This calculation correctly finds the midpoint as 4, avoiding any potential overflow because (4 - 4) is 0, and 4 + 0 is 4, which is within the valid range (max_int).

Conclusion

By using left + (right - left) // 2, we mitigate the risk of integer overflow scenarios, ensuring the binary search algorithm remains robust and reliable. This approach is particularly important in environments where integer limits are stringent, maintaining consistency and reliability in search operations.