Binary Search Midpoint Calculation: Complete Guide

The Three Approaches

1. `int mid = left + (right - left) / 2;` (Java - Overflow Safe)

When to use: This is the **preferred approach** for most competitive programming and production code in languages with fixed integer sizes.

Why it's different:

- **Overflow Prevention**: The key advantage is preventing integer overflow
- When `left` and `right` are both large integers (close to `Integer.MAX_VALUE`), adding them directly could exceed the maximum integer value
- By calculating `(right - left) / 2` first, we work with the difference, which is always smaller than either original value
- Then we add this smaller value to `left`, ensuring we never exceed the integer limits

Mathematical breakdown:

```
Instead of: (left + right) / 2
We compute: left + (right - left) / 2
Which equals: left + right/2 - left/2 = left/2 + right/2 = (left + right) / 2
```

Example where overflow matters:

```java
int left = 1_500_000_000;  // Close to Integer.MAX_VALUE (2.1 billion)
int right = 1_800_000_000;
// left + right = 3.3 billion > Integer.MAX_VALUE → OVERFLOW!
// But (right - left) = 300 million → Safe!
```

2. `int mid = (int) Math.floor((left + right) / 2);` (Java - Explicit Floor)

When to use: This approach is **rarely necessary** in practice for binary search with integers.

Why it's different:

- **Explicit Floor Operation**: `Math.floor()` explicitly rounds down to the nearest integer

- **Double Precision**: `Math.floor()` works with `double` values, so the division happens in floating-point arithmetic first
- **Type Casting**: Requires explicit casting back to `int`

When this might be relevant:

- When working with floating-point binary search (searching in continuous ranges)
- When you want to be extremely explicit about the rounding behavior
- In languages where integer division doesn't automatically floor negative results

Potential issues:

- **Performance overhead**: Converting to double, calling Math.floor(), then casting back
- **Still vulnerable to overflow**: `left + right` can still overflow before the division
- **Unnecessary complexity**: For positive integers, regular integer division already floors

3. `let mid = Math.floor((left + right) / 2)` (JavaScript)

When to use: This is the **standard approach in JavaScript** for binary search.

Why it's different in JavaScript:

- **No Integer Overflow**: JavaScript uses 64-bit floating-point numbers for all numeric values
- **Safe Range**: Can safely represent integers up to `Number.MAX_SAFE_INTEGER` (2^53 - 1)
- **Division Behavior**: Regular division (`/`) in JavaScript produces floating-point results, so `Math.floor()` is needed to get integer indices

JavaScript-specific considerations:

```javascript
// Without Math.floor(), you might get:
let left = 5, right = 8;
let mid = (left + right) / 2;  // mid = 6.5 (not a valid array index!)

// With Math.floor():
let mid = Math.floor((left + right) / 2);  // mid = 6 (valid index)
```

Practical Recommendations

For Java/C++/C# (Fixed Integer Size Languages):

```java
// ✅ RECOMMENDED: Overflow-safe approach
int mid = left + (right - left) / 2;

// ❌ AVOID: Potential overflow
int mid = (left + right) / 2;

// ❌ UNNECESSARY: Overcomplicated for integer binary search
int mid = (int) Math.floor((left + right) / 2);
```

For JavaScript:

```javascript
// ✅ RECOMMENDED: Standard JavaScript approach
let mid = Math.floor((left + right) / 2);

// ⚠️ ALTERNATIVE: Bitwise right shift (also floors for positive numbers)
let mid = (left + right) >> 1;

// ❌ WRONG: Produces floating-point results
let mid = (left + right) / 2;
```

For Python:

```python
# ✅ RECOMMENDED: Integer division automatically floors
mid = (left + right) // 2

# ✅ ALSO FINE: Python handles arbitrarily large integers
mid = left + (right - left) // 2
```

Edge Cases to Consider

Negative Numbers:

The overflow-safe approach works correctly with negative numbers:

```java
```

```
int left = -10, right = -5;
int mid = left + (right - left) / 2; // = -10 + 5/2 = -10 + 2 = -8 ✓
```

Large Numbers:

```
java

// This could overflow in Java:
int left = 1_000_000_000;
int right = 2_000_000_000;
int mid = (left + right) / 2; // Potential overflow!

// This is safe:
int mid = left + (right - left) / 2; // Always safe
```

Summary Table

Approach	Language	Overflow Safe?	Performance	Recommended?
`left + (right - left) / 2`	Java/C++	✅ Yes	Fast	✅ Preferred
`(int) Math.floor((left + right) / 2)`	Java	❌ No	Slower	❌ Avoid
`Math.floor((left + right) / 2)`	JavaScript	✅ Yes*	Fast	✅ Standard

*JavaScript doesn't have integer overflow in the traditional sense due to its number representation.

Key Takeaway

The choice of midpoint calculation depends on your language and the potential range of your data:

- **Java/C++**: Always use the overflow-safe version unless you're certain your values are small
- **JavaScript**: Use `Math.floor((left + right) / 2)` as the standard approach
- **Python**: Use integer division `//` which automatically floors the result

The overflow-safe approach in Java isn't just about being cautious—it's about writing robust code that works correctly even with edge case inputs that you might encounter in competitive programming or real-world applications.