Peak Index in Mountain Array - LeetCode 852

The problem finds the index of the "peak" element in a mountain array - an array that increases to a single peak, then decreases. **Guaranteed to have exactly one peak, array length ≥ 3.**

Example: `[0,3,8,9,5,2]` → Peak is `9` at **index 3**

Mountain Array Properties 🔑

Peak element characteristics:

- **Left side (before peak):** Strictly **increasing** → `arr[i] > arr[i-1]`
- **Right side (after peak):** Strictly **decreasing** → `arr[i] > arr[i+1]`
- **Peak itself:** `arr[peak] > arr[peak-1]` **AND** `arr[peak] > arr[peak+1]`

Important constraints:

- Peak **cannot** be at index `0` or `n-1`
- Array length ≥ 3, so safe to access `mid-1` and `mid+1`

Why Binary Search? 💡

Mountain array has sorted structure (increasing → decreasing), perfect for binary search

Time complexity: O(log n) instead of O(n) linear scan

Step-by-Step Algorithm

1. Initialize Search Space

```
start = 1, end = n-2  // Skip edges (never peak)
```

Why? Eliminates boundary checks for `mid-1` / `mid+1`

2. Binary Search Loop (`while start <= end`)

```
mid = start + (end - start) / 2  // Avoid overflow
```

3. Check if mid is Peak 🔍

```
if (arr[mid] > arr[mid-1] && arr[mid] > arr[mid+1])
    return mid;  // Found peak!
```

4. Decide Search Direction 🚀

Case A: Increasing slope (`arr[mid-1] < arr[mid]`)

```
start = mid + 1;  // Peak must be RIGHT
```

Case B: Decreasing slope (`arr[mid-1] >= arr[mid]`)

```
end = mid - 1;  // Peak must be LEFT
```

Dry Run: [0,3,8,9,5,2]

```
n=6, start=1, end=4 (n-2)

Iteration 1:
mid = 1 + (4-1)/2 = 2 (arr[2]=8)
Check: 8 > 3? ✓  8 > 9? ×  → Not peak
arr[1]=3 < arr[2]=8? ✓ → Increasing → start=3

Iteration 2:
start=3, end=4
mid = 3 + (4-3)/2 = 3 (arr[3]=9)
Check: 9 > 8? ✓  9 > 5? ✓ → **PEAK FOUND! Return 3**
```

Complete Code

```
int peakIndexInMountainArray(vector<int>& arr) {
    int start = 1, end = arr.size() - 2;  // Optimized bounds

    while (start <= end) {
        int mid = start + (end - start) / 2;  // Safe mid

        // Peak check
        if (arr[mid] > arr[mid-1] && arr[mid] > arr[mid+1])
            return mid;

        // Increasing slope → search right
        else if (arr[mid-1] < arr[mid])
            start = mid + 1;

        // Decreasing slope → search left
        else
            end = mid - 1;
    }
    return -1;  // Never reaches (guaranteed peak exists)
}
```