132 Pattern Detection - Detailed Documentation

Table of Contents

1. **Problem Statement**

Given an array of n integers nums, a 132 pattern is a subsequence of three integers nums[i], nums[j], and nums[k] such that:

- i < j < k
- nums[i] < nums[k] < nums[j]

Return True if there is a 132 pattern in nums, otherwise, return False.

Example 1

- Input: nums = [1, 2, 3, 4]
- Output: False
- Explanation: There is no 132 pattern.

Example 2

- Input: nums = [3, 1, 4, 2]
- Output: True
- Explanation: The pattern [1, 4, 2] follows 132 order.

Example 3

- Input: nums = [-1, 3, 2, 0]
- Output: True
- Explanation: Multiple valid patterns: [-1, 3, 2], [-1, 3, 0], [-1, 2, 0].

Constraints

- n == nums.length
- 1 <= n <= 2 * 10^5
- -10^9 <= nums[i] <= 10^9

2. Intuition

The goal is to identify three numbers in the array following the **132 pattern** constraint efficiently.

- A **brute-force approach** checking all triplets (i, j, k) is **too slow** ($O(n^3)$ complexity).
- A **more optimal approach** is to track potential nums[k] using a **monotonic stack** and iterate in **reverse order**.

3. Key Observations

i. We need three numbers where:
   a. nums[i] < nums[k] < nums[j]
   b. i < j < k

ii. If we iterate **backward**, we can maintain the largest valid nums[k] efficiently using a **monotonic decreasing stack**.

iii. The nums[k] value must be **less than** nums[j] **but greater than** nums[i], which can be efficiently tracked using a **stack**.

4. Approach

We use a monotonic decreasing stack to track elements from right to left:

1. Initialize variables:
   - stack = [] (monotonic decreasing stack)
   - third = -∞ (potential nums[k])
2. Iterate from right to left:
   - If nums[i] < third, return True (we found a valid pattern).
   - While stack is non-empty and stack[-1] < nums[i], update third = stack.pop() (ensuring it remains the largest possible nums[k]).
   - Push nums[i] onto the stack.
3. Return False if no pattern is found.

5. Edge Cases

Case Type	**Example**	**Expected Output**	**Reason**
Minimum Input	[1]	False	Only one element, cannot form a pattern.
All Increasing	[1, 2, 3, 4, 5]	False	No valid nums[k] satisfying nums[i] < nums[k] < nums[j].
All Decreasing	[5, 4, 3, 2, 1]	False	No valid 132 pattern possible.
Negative Values	[-1, 3, 2, 0]	True	Multiple valid patterns exist.
Duplicates	[3, 3, 3, 3, 3]	False	All elements are the same, no pattern possible.
Large Input	[10^5, ..., 1]	False	Fully decreasing, so no 132 pattern.

6. **Complexity Analysis**

Time Complexity

- **O(n)** – Each element is pushed and popped from the stack **at most once**.

Space Complexity

- **O(n)** – In the worst case, all elements are stored in the stack.

7. **Alternative Approaches**

Brute Force ($O(n^3)$)

- Iterate through all triplets (i, j, k).
- Compare each nums[i], nums[j], nums[k]` for the 132 pattern.
- Time Complexity: $O(n^3)$ (too slow for large inputs).

Using Two Arrays ($O(n^2)$)

- Precompute the minimum nums[i] up to index j.
- Use a nested loop to find nums[k] satisfying the pattern.
- Time Complexity: $O(n^2)$, still inefficient.

8. **Test Cases**

```
def test_find132pattern():
    sol = Solution()
    assert sol.find132pattern([1, 2, 3, 4]) == False
    assert sol.find132pattern([3, 1, 4, 2]) == True
    assert sol.find132pattern([-1, 3, 2, 0]) == True
    assert sol.find132pattern([3, 3, 3, 3]) == False
    assert sol.find132pattern([1, 0, 1, -4, -3]) == False
    assert sol.find132pattern([9, 11, 8, 9, 10, 7, 8]) == True
    assert sol.find132pattern([10, 20, 30, 40]) == False
    assert sol.find132pattern([3, 5, 0, 3, 4]) == True
print("All test cases passed!")
test_find132pattern()
```

9. Final Thoughts

- The **monotonic stack approach** efficiently detects the **132 pattern** in **O(n) time**.
- The solution works well even for large inputs ($n \approx 2 \times 10^5$).
- Alternative approaches like **brute force** and **nested loops** are too slow.
- This is an excellent example of using **stacks** to solve **sequence-based** problems efficiently.