456. 132 Pattern

Medium · Stack · Array · Binary Search · Ordered Set · Monotonic Stack — Leetcode Link

Problem Description

The problem requires us to check if there is a specific pattern, called the "132 pattern", within an array of integers, `nums`. This pattern is defined by finding three numbers in the sequence that can be indexed with `i`, `j`, and `k` such that `i < j < k`. These three numbers must satisfy the criteria that `nums[i] < nums[k] < nums[j]`.

In other words, amongst the three numbers forming this subsequence pattern, the first number should be the smallest and the last number should be in the middle—not as high as the second number but higher than the first. This pattern does not need to be consecutive but must maintain the order.

The objective is to return `true` if at least one occurrence of the "132 pattern" is found in the given array, and `false` otherwise.

Intuition

The solution exploits the property of a stack data structure to keep track of potential candidates for `nums[k]` and `nums[j]` from right to left.

1. Initialize a variable `vk` as negative infinity to represent a potential candidate for `nums[k]` which is the middle element in our "132 pattern". We initially set it to negative infinity because we're looking for a value that is greater than the smallest value found so far as we scan the array from right to left.

2. Then, create an empty list `stk` which will act as a stack to store potential candidates for `nums[j]`.

3. Reverse iterate over the array. For each element `x` (acting as `nums[i]`):

    - First, check if `x` is less than `vk`. If so, we have found a valid "132 pattern" because we have ensured earlier that any value in `vk` must be greater than the smallest value found and less than some value that came after it (acting as `nums[j]`). Therefore, we return `true`.

    - If not, then we pop elements from `stk` which are less than `x` and update `vk` with those values. The popping continues until we find an element greater than or equal to `x` or until `stk` is empty. This ensures that `vk` is always the greatest possible value just smaller than our current `nums[j]` candidate (the top of the stack) while being greater than the smallest value found so far.

    - Finally, push `x` onto the `stk` to consider it for the future `nums[j]`.

4. If the loop completes without finding the "132 pattern", return `false`.

This approach effectively finds the "132 pattern" by ensuring that for any given element, if it is a potential candidate for `nums[i]`, there is an already established candidate for `nums[k]` (`vk`) which is less than a potential candidate for `nums[j]` (top of the stack) and greater than the `nums[i]`. By scanning from right to left, we are efficiently maintaining candidates for `nums[j]` and `nums[k]` while looking for a valid `nums[i]`.

Solution Approach

The implementation of the solution utilizes a stack data structure for its ability to operate in a last-in-first-out manner. This approach allows us to keep track of potential `nums[j]` elements and efficiently update the candidate for `nums[k]`.

The algorithm follows these steps:

1. We start by initializing a variable `vk` to negative infinity (`-inf`). This variable will hold the potential candidate for the middle value `nums[k]` in the "132 pattern" as we traverse through the array.

2. An empty list `stk` is then created to function as our stack, which will store the potential candidates for the `nums[j]` values.

3. The array `nums` is iterated in reverse order using `for x in nums[::-1]:`. Here, `x` acts as a candidate for our `nums[i]`.

4. Inside the loop, we perform a check: `if x < vk: return True`. This is the condition that confirms the presence of the "132 pattern". Since `vk` is always chosen to be less than a previously encountered element (which would be `nums[j]`), finding an `x` that is even lesser confirms our sequence.

5. If the current element `x` doesn't validate the pattern, we handle the stack `stk`. With `while stk and stk[-1] < x:`, we pop elements from the stack that are less than `x`, updating `vk` to `stk.pop()`. Each popped value is a new candidate for `nums[k]` since it satisfies both being less than our `nums[j]` (which `x` could potentially be) and greater than the smallest element encountered (since it was added to the stack after that element).

6. After the above operation, regardless of whether we popped from the stack or not, we append the current element `x` to the stack with `stk.append(x)`. This step considers `x` for a future `nums[j]` candidate.

7. If the entire array is traversed without returning `True`, the function concludes with `return False`, indicating that no "132 pattern" was found.

The elegance in this solution lies in the efficiency with which it maintains the candidates for `nums[j]` and `nums[k]`, thus drastically reducing the time complexity compared to a naive triple-nested loop approach which would check all possible combinations of `i`, `j`, and `k`.

Thanks to the stack, we keep the order of elements in such a way that at any given moment, the top of the stack (`stk[-1]`) is our best guess for `nums[j]`, and `vk` is our best guess for `nums[k]`. This method ensures that we never miss a valid pattern and negate the need to check every possible subsequence explicitly.

Example Walkthrough

Let's consider the array `nums = [3, 1, 4, 2]` and walk through the solution approach to determine if a "132 pattern" exists.

1. Initialize `vk` to negative infinity since we haven't yet begun iterating over `nums`.
2. Create an empty stack `stk` to store potential candidates for `nums[j]`.

Now, we iterate over `nums` in reverse:

- Start with `x = 2 (nums[k])`. Push `x` onto `stk`, `vk` is unchanged. Stack `stk = [2]`.
- Next, `x = 4 (nums[j])`. Since `stk` is not empty and `2 < 4`, pop `2` from `stk` and update `vk` to `2`. Now `stk = []` and `vk = 2`. Push `4` onto `stk`, `stk` becomes `[4]`.
- Then, `x = 1 (nums[i])`. Check if `x < vk`. It is not since `1 < 2` is false, so we push `1` to `stk`, which becomes `[4, 1]`.
- Lastly, `x = 3 (nums[i])`. Check if `3 < vk (2)`. It is false. Pop `1` from `stk` as long as `1 < 3`. Update `vk` to `1`. Now we check again if `x < vk`. Since `3 < 4` is true and `vk(1) < x(3)` is also true, we have found our "132" pattern where `nums[i] = 1`, `nums[k] = 3`, and `nums[j] = 4`.

Because we successfully found a "132" pattern, we can return `true`.

This example demonstrates how the stack and the `vk` variable are updated throughout the process while confirming the pattern as soon as the conditions are met without having to finish iterating through the entire array. This makes the approach efficient and effective at identifying the presence of a "132 pattern".

Python Solution

```python
class Solution:
    def find132pattern(self, nums: List[int]) -> bool:
        # Initialize the third value in the pattern to negative infinity
        third_value = float('-inf')
        # Initialize an empty stack to store elements of the nums list
        stack = []
        # Traverse the nums list in reverse order
        for number in reversed(nums):
            # If the current number is less than the third_value, a 132 pattern is found
            if number < third_value:
                return True
            # While there's an element in the stack and it's less than the current number
            while stack and stack[-1] < number:
                # Update the third_value to the last element in the stack
                third_value = stack.pop()
            # Push the current number onto the stack
            stack.append(number)
        # Return False if no 132 pattern is found
        return False

```

Java Solution

```java
class Solution {
    public boolean find132pattern(int[] nums) {
        // Initialize vk with the smallest possible integer value using bitwise shift
        int potentialMidVal = Integer.MIN_VALUE;
        Deque<Integer> stack = new ArrayDeque<>(); // Create a stack to store the elements

        // Iterate from the end to the start of the array
        for (int i = nums.length - 1; i >= 0; --i) {
            // If the current element is less than the potential middle value of the 132 pattern,
            // this means we have found a 132 pattern.
            if (nums[i] < potentialMidVal) {
                return true;
            }
            // While the stack is not empty and the top element is less than the current number,
            // update the potential middle value to be the new top of the stack
            while (!stack.isEmpty() && stack.peek() < nums[i]) {
                potentialMidVal = stack.pop();
            }
            // Push the current number onto the stack
            stack.push(nums[i]);
        }
        // If the loop completes without finding a 132 pattern, return false
        return false;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <stack>
#include <climits>

class Solution {
public:
    // This function checks if there is a 132 pattern in the input vector "nums".
    // A 132 pattern is a subsequence of three integers where the first is smaller than the third and both are smaller than the secor
    bool find132pattern(vector<int>& nums) {
        // Initialize the variable to hold the value of the third element in the 132 pattern, initialized to the minimum integer valu
        int thirdValue = INT_MIN;

        // Use a stack to help find potential candidates for the second element in the 132 pattern.
        stack<int> candidates;

        // Iterate through the input array backwards.
        for (int i = nums.size() - 1; i >= 0; --i) {
            // Check if we have achieved the 132 pattern
            if (nums[i] < thirdValue) {
                // we found a valid 132 pattern
                return true;
            }
            // While we have candidates and the current number is greater than the candidate at the top of the stack
            while (!candidates.empty() && candidates.top() < nums[i]) {
                // The candidate could potentially be the third value in the pattern, so we update the thirdValue.
                thirdValue = candidates.top();
                candidates.pop(); // Remove the candidate as it has been used
            }
            // Push the current number onto the stack to be a candidate for the second position in the 132 pattern.
            candidates.push(nums[i]);
        }

        // If we reach this point, no 132 pattern has been found.
        return false;
    }
};

```

Typescript Solution

```typescript
function find132pattern(nums: number[]): boolean {
    // Initialize the variable to represent the 'k' element of the 132 pattern and set it to negative infinity.
    let patternK = -Infinity;

    // Initialize an empty stack to use as an auxiliary data structure.
    const stack: number[] = [];

    // Iterate through the given array from the end towards the beginning.
    for (let i = nums.length - 1; i >= 0; --i) {
        // If the current element is smaller than the 'k' element of the pattern, a 132 pattern is found.
        if (nums[i] < patternK) {
            return true;
        }

        // While there are elements on the stack and the top of the stack is smaller than the current element,
        // update the 'k' element with the values popped from the stack.
        while (stack.length && stack[stack.length - 1] < nums[i]) {
            patternK = stack.pop()!;
        }

        // Push the current element onto the stack.
        stack.push(nums[i]);
    }

    // If no 132 pattern is found, return false.
    return false;
}

```

Time and Space Complexity

Time Complexity

The given algorithm iterates through the input array `nums` once in reverse order. The outer loop has a worst-case scenario of `O(n)` where `n` is the length of `nums`. For each element, the algorithm performs operations involving a stack `stk`. In the worst case, for each element of the array, the algorithm might push and pop on the order of `O(n)` times collectively over the entire run of the algorithm due to the nature of maintaining a stack for the pattern. However, an important property of the stack operations is that each element is pushed and popped at most once, leading to amortized `O(1)` time per element. Therefore, the total time complexity of the algorithm is `O(n)`.

Space Complexity

The space complexity of the algorithm is determined by the additional space used by the stack `stk`. In the worst case, the stack could grow to have all elements of `nums` if the array is strictly decreasing, leading to a space complexity of `O(n)`. Otherwise, the stack will contain fewer elements. So the worst-case space complexity is `O(n)`.