132 Pattern

Given a sequence of n integers a_1, a_2, ..., a_n, a 132 pattern is a subsequence a_i, a_j, a_k such that **i** j k and a_i **k j**. Design an algorithm that takes a list of n numbers as input and checks whether there is a 132 pattern in the list.

Note: n will be less than 15,000.

Example 1:

```
Input: [1, 2, 3, 4]

Output: False

Explanation: There is no 132 pattern in the sequence.
```

Example 2:

```
Input: [3, 1, 4, 2]

Output: True

Explanation: There is a 132 pattern in the sequence: [1, 4, 2].
```

Example 3:

```
Input: [-1, 3, 2, 0]

Output: True

Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] a
nd [-1, 2, 0].
```

Solution 1

We want to search for a subsequence (`s1,s2,s3`)

INTUITION: The problem would be simpler if we want to find sequence with `s1 > s2 > s3`, we just need to find `s1`, followed by `s2` and `s3`. Now if we want to find a 132 sequence, we need to switch up the order of searching. we want to first find `s2`, followed by `s3`, then `s1`.

IDEA: We can start from either side but I think starting from the end allow us to finish in a single pass. The idea is to start from end and search for a candidate for `s2` and `s3`. A number becomes a candidate for `s3` if there is any number on the left of `s2` that is bigger than it.

DETECTION: Keep track of the largest candidate of `s3` and once we encounter any number smaller than `s3`, we know we found a valid sequence since `s1 < s3` implies `s1 < s2`.

IMPLEMENTATION:

1. Have a `stack`, each time we store a new number, we first `pop` out all numbers that are smaller than that number. The numbers that are `popped` out becomes candidate for `s3`.
2. We keep track of the `maximum` of such `s3` (which is always the most recently `popped` number from the `stack`).
3. Once we encounter any number smaller than `s3`, we know we found a valid sequence since `s1 < s3` implies `s1 < s2`.

RUNTIME: Each item is `pushed` and `popped` once at most, the time complexity is therefore O(n).

EXAMPLE:
`i = 6`, **nums** = [9, 11, 8, 9, 10, 7, `9`], **S1 candidate** = `9`, **S3 candidate** = `None`, **Stack** = `Empty`
`i = 5`, **nums** = [9, 11, 8, 9, 10, `7`, 9], **S1 candidate** = `7`, **S3 candidate** = `None`, **Stack** = `[9]`
`i = 4`, **nums** = [9, 11, 8, 9, `10`, 7, 9], **S1 candidate** = `10`, **S3 candidate** = `None`, **Stack** = `[9,7]`
`i = 3`, **nums** = [9, 11, 8, `9`, 10, 7, 9], **S1 candidate** = `9`, **S3 candidate** = `9`, **Stack** = `[10]`
`i = 2`, **nums** = [9, 11, `8`, 9, 10, 7, 9], **S1 candidate** = `8`, **S3 candidate** = `9`, **Stack** = `[10,9]` **We have `8<9`, sequence found!**

EDIT: Thanks @Pumpkin78 and @dalwise for pointing out that the maximum candidate for s3 is always the recently popped number from the stack, because if we encounter any entry smaller than the current candidate, the function would already have returned.

```
    bool find132pattern(vector<int>& nums) {
        int s3 = INT_MIN;
        stack<int> st;
        for( int i = nums.size()-1; i >= 0; i -- ){
            if( nums[i] < s3 ) return true;
            else while( !st.empty() && nums[i] > st.top() ){
              s3 = st.top(); st.pop();
            }
            st.push(nums[i]);
        }
        return false;
    }
```

written by jade86 original link here

Solution 2

The idea is that we can use a stack to keep track of previous min-max intervals.

Here is the principle to maintain the stack:

For each number `num` in the array

If stack is empty:

- push a new Pair of `num` into stack

If stack is not empty:

- if `num` < `stack.peek().min`, push a new Pair of `num` into stack
- if `num` >= `stack.peek().min`, we first pop() out the peek element, denoted as `last`
    - if `num` < `last.max`, we are done, return `true`;
    - if `num` >= `last.max`, we merge `num` into `last`, which means `last.max` = `num`.
      Once we update `last`, if stack is empty, we just push back `last`.
      However, the crucial part is:
      If stack is not empty, the updated `last` might:
        - Entirely covered stack.peek(), i.e. `last.min` < `stack.peek().min` (which is always true) && `last.max` >= `stack.peek().max`, in which case we keep popping out stack.peek().
        - Form a 1-3-2 pattern, we are done ,return `true`

So at any time in the stack, **non-overlapping** `Pairs` are formed in descending order by their min value, which means the min value of peek element in the stack is always the min value globally.

```
   class Pair{
        int min, max;
        public Pair(int min, int max){
            this.min = min;
            this.max = max;
        }
    }
    public boolean find132pattern(int[] nums) {
        Stack<Pair> stack = new Stack();
        for(int n: nums){
            if(stack.isEmpty() || n <stack.peek().min ) stack.push(new Pair(n,n))
;
            else if(n > stack.peek().min){
                Pair last = stack.pop();
                if(n < last.max) return true;
                else {
                    last.max = n;
                    while(!stack.isEmpty() && n >= stack.peek().max) stack.pop();
                    // At this time, n < stack.peek().max (if stack not empty)
                    if(!stack.isEmpty() && stack.peek().min < n) return true;
                    stack.push(last);
                }

            }
        }
        return false;
    }
```

written by leogogogo original link here

Solution 3

Idea: Find peak and bottom

For every [bottom, peak], find if there is one number `bottom<number<peak`.

```
public class Solution {
    public boolean find132pattern(int[] nums) {
        if(nums.length<3) return false;
        Integer low = null, high = null;
        int start = 0, end = 0;
        while(start<nums.length-1){
            while(start<nums.length-1 && nums[start]>=nums[start+1]) start++;
            // start is lowest now
            int m = start+1;
            while(m<nums.length-1 && nums[m]<=nums[m+1]) m++;
            // m is highest now
            int j = m+1;
            while(j<nums.length){
                if(nums[j]>nums[start] && nums[j]<nums[m]) return true;
                j++;
            }
            start = m+1;
        }
        return false;
    }
}
```

written by YuTingLiu original link here