<https://leetcode.com/problems/132-pattern/>

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 and 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 in the sequence.

**Example 2:**

Input: nums = [3,1,4,2]

Output: true

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

**Example 3:**

Input: nums = [-1,3,2,0]

Output: true

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

**Constraints:**

* n == nums.length
* 1 <= n <= 2 \* 105
* -109 <= nums[i] <= 109

**Attempt 1: 2023-03-27**

**Solution 1: Monotonic Decreasing Stack (120 min)**

**Style 1: Scan from left to right (too hard to come up with)**

**Refer to**

<https://leetcode.com/problems/132-pattern/solutions/906789/short-java-o-n-solution-with-detailed-explanation-sample-test-case-stack-100/comments/1139841>

class Solution {

public boolean find132pattern(int[] nums) {

Deque<int[]> stack = new ArrayDeque<>();

for(int min = nums[0], i = 1; i < nums.length; i++) {

while(!stack.isEmpty() && nums[i] >= stack.peek()[0]) {

stack.pop();

}

if(!stack.isEmpty() && nums[i] > stack.peek()[1]) {

return true;

}

stack.push(new int[]{nums[i], min = Math.min(nums[i], min)});

}

return false;

}

}

<https://leetcode.com/problems/132-pattern/solutions/94077/java-on-solution-using-stack-in-detail-explanation/>

class Solution {

class Pair{

int min, max;

public Pair(int min, int max){

this.min = min;

this.max = max;

}

}

// Test with {6,12,3,4,6,11,20}

public boolean find132pattern(int[] nums) {

Stack<Pair> stack = new Stack();

for(int n : nums) {

// Keep push smaller min Pair onto stack to make sure its descending order

// based on min value, which means the peek Pair's min value will always

// globally smallest one, this will help to make sure we have largest range

// (stored as Pair.min/Pair.max) to find possible third number when construct

// 132 Pattern

// 保持stack 中的每一段都是以其第一个值单调递减的

if(stack.isEmpty() || n < stack.peek().min) {

stack.push(new Pair(n, n));

// 此时 n 大于 stack top的最小值

} else if(n > stack.peek().min) {

Pair last = stack.pop();

// If find 132 Pattern, directly return true

// 如果 n 同时小于stack top的最大值，那么返回True

if(n < last.max) {

return true;

// If not find, detail check relation between current number n and previous

// Pair stored on stack peek

} else {

// Update(merge) previous stack peek Pair's smaller max value with current number n

// 否则的话，把stack top的最大值更新为n， 注意此时stack top的最小值，是全部stack元素的最小值

last.max = n;

// This while loop target is to find the possible existing Pair stored

// on stack which satisfy 132 Pattern condition as

// [stack.peek().min < n < stack.peek().max], to approach this case,

// we recursively pop out all Pairs that satisfy n >= stack.peek().max

// 如果 n 比stack栈顶元素的最大值还要大，那么说明当前的range包含于pop出来的元素的range，所以pop栈顶

while(!stack.isEmpty() && n >= stack.peek().max) {

stack.pop();

}

// At this time, if find one Pair n < stack.peek().max (if stack not empty)

// it means 132 Pattern condition [stack.peek().min < n < stack.peek().max]

// exist, return true

// 此时的情况是如果栈顶最小值比n小，同时栈顶的最大值比n大

if(!stack.isEmpty() && stack.peek().min < n) {

return true;

}

// If still not find 132 Pattern, push back the updated(merge previous stack

// peek Pair's smaller max value with current number n)

stack.push(last);

}

}

}

return false;

}

}

**Style 2: Scan from right to left**

class Solution {

public boolean find132pattern(int[] nums) {

Stack<Integer> stack = new Stack<Integer>();

int second = Integer.MIN\_VALUE;

for(int i = nums.length - 1; i >= 0; i--) {

if(nums[i] < second) {

return true;

}

while(!stack.isEmpty() && stack.peek() < nums[i]) {

second = stack.pop();

}

stack.push(nums[i]);

}

return false;

}

}

**Refer to**

<https://leetcode.com/problems/132-pattern/solutions/906789/short-java-o-n-solution-with-detailed-explanation-sample-test-case-stack-100/>

**EXPLANATION**:

*Here, if we fix the peak, i.e.* ***3*** *in the* ***132 pattern****, then we can determine if any numbers on its left and right satisfy the given pattern. We will do this with the help of a stack. Our stack implementation will take care of the* ***32 pattern*** *and then we will iterate over the array to find if any number satisfies the* ***1 pattern****. See the algorithm below for better understanding*.

**Algorithm:**

1. Create a stack and initialize a variable **second** with INT\_MIN value.
2. Start traversing from the end of array.
3. Check if the current number is lesser than **second**. If it is, then it means our **132 pattern** is satisfied as the stack is taking care of the **32 pattern** and the current number satisfies the entire pattern. So return **true**.
4. If the above is not true, update the peak value in the stack. Keep popping from the stack until stack is empty OR the top value is greater than the current value.
5. Push the current number into the stack.
6. If the loop terminates, it means that the pattern was not found in the array. So, return **false**.

***Take the sample input as [3, 6, 4, 1, 2] and try out this algorithm using a pen & paper. You will be able to visualize the method then.***

Basically, when scan from right to left till num = 4, the **top** of stack is containing the **highest** number so far, i.e. **4**, and **second** is containing the **second highest** number after the highest number, i.e. **2**. So, this satisfies the **32 pattern**. Now, we will just keep updating **second** and **stack top** when we encounter a number which is greater than the highest number.

class Solution {

public boolean find132pattern(int[] nums) {

Stack<Integer> stack = new Stack();

int second = Integer.MIN\_VALUE;

for(int i = nums.length - 1; i >= 0; i--) {

if(nums[i] < second) {

return true;

}

while(!stack.isEmpty() && nums[i] > stack.peek()) {

second = stack.pop();

}

stack.push(nums[i]);

}

return false;

}

}

**Solution 2: Brute Force (10 min, TLE)**

class Solution {

public boolean find132pattern(int[] nums) {

for(int i = 0; i < nums.length; i++) {

for(int j = i + 1; j < nums.length; j++) {

for(int k = j + 1; k < nums.length; k++) {

if(nums[i] < nums[k] && nums[k] < nums[j]) {

return true;

}

}

}

}

return false;

}

}

**Refer to**

<https://leetcode.com/problems/132-pattern/solutions/94089/java-solutions-from-o-n-3-to-o-n-for-132-pattern-updated-with-one-pass-slution/>

**I. Naive O(n^3) solution**

The naive O(n^3) solution is a no-brainer --- simply check every (i, j, k) combination to see if there is any 132 pattern.

public boolean find132pattern(int[] nums) {

for (int i = 0; i < nums.length; i++) {

for (int j = i + 1; j < nums.length; j++) {

for (int k = j + 1; k < nums.length; k++) {

if (nums[i] < nums[k] && nums[k] < nums[j]) return true;

}

}

}

return false;

}

And of course it will get rejected due to TLE. So let's see how we can do better.