<https://leetcode.com/problems/increasing-triplet-subsequence/>

Given an integer array nums, return true *if there exists a triple of indices* (i, j, k) *such that* i < j < k *and* nums[i] < nums[j] < nums[k]. If no such indices exists, return false.

**Example 1:**

Input: nums = [1,2,3,4,5]

Output: true

Explanation: Any triplet where i < j < k is valid.

**Example 2:**

Input: nums = [5,4,3,2,1]

Output: false

Explanation: No triplet exists.

**Example 3:**

Input: nums = [2,1,5,0,4,6]

Output: true

Explanation: The triplet (3, 4, 5) is valid because nums[3] == 0 < nums[4] == 4 < nums[5] == 6.

**Constraints:**

* 1 <= nums.length <= 5 \* 105
* -231 <= nums[i] <= 231 - 1

**Follow up:** Could you implement a solution that runs inO(n)time complexity andO(1)space complexity?

**Attempt 1: 2023-04-08**

**Solution 1: DP (10 min, exactly same as L300. Longest Increasing Subsequence, but TLE)**

class Solution {

public boolean increasingTriplet(int[] nums) {

int maxLen = 0;

int len = nums.length;

int[] dp = new int[len];

Arrays.fill(dp, 1);

for(int i = 0; i < len; i++) {

for(int j = 0; j < i; j++) {

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

dp[i] = Math.max(dp[i], dp[j] + 1);

maxLen = Math.max(maxLen, dp[i]);

}

}

}

return maxLen >= 3;

}

}

Time Complexity: O(2^N), where N is the size of nums. At each index, we have choice to either take or not take the element and we explore both ways. So, we 2 \* 2 \* 2...N times = O(2^N)

Space Complexity: O(N), max recursive stack depth.

**Solution 2: Binary Search (10 min, exactly same as L300. Longest Increasing Subsequence)**

class Solution {

public boolean increasingTriplet(int[] nums) {

List<Integer> list = new ArrayList<Integer>();

for(int cur : nums) {

if(list.size() == 0 || list.get(list.size() - 1) < cur) {

list.add(cur);

} else {

int index = binarySearch(list, cur);

list.set(index, cur);

}

}

return list.size() >= 3;

}

private int binarySearch(List<Integer> list, int target) {

int start = 0;

int end = list.size() - 1;

while(start <= end) {

int mid = start + (end - start) / 2;

if(list.get(mid) >= target) {

end = mid - 1;

} else {

start = mid + 1;

}

}

return start;

}

}

Time Complexity: O(N \* logN), where N <= 2500 is the number of elements in array nums.

Space Complexity: O(N), we can achieve O(1) in space by overwriting values of sub into original nums array.

**Solution 3: One Pass (30 min)**

class Solution {

public boolean increasingTriplet(int[] nums) {

int first = Integer.MAX\_VALUE;

int second = Integer.MAX\_VALUE;

for(int num : nums) {

if(num <= first) {

first = num;

} else if(num <= second) {

second = num;

} else {

return true;

}

}

return false;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/increasing-triplet-subsequence/solutions/79004/concise-java-solution-with-comments/>

<https://leetcode.com/problems/increasing-triplet-subsequence/solutions/79004/concise-java-solution-with-comments/comments/247643>

public boolean increasingTriplet(int[] nums) {

// start with two largest values, as soon as we find a number bigger than both, while both have been updated, return true.

int small = Integer.MAX\_VALUE, big = Integer.MAX\_VALUE;

for (int n : nums) {

if (n <= small) { small = n; } // update small if n is smaller than both

else if (n <= big) { big = n; } // update big only if greater than small but smaller than big

else return true; // return if you find a number bigger than both

}

return false;

}

This was a reply to comment, but I'm posting it here if it helps anyone else

Intuitively what this solution does is keeps tracks of lower the bounds for the first and second element of the subsequence. Instead of small and big I will call it first and second initially we have first = INF and second = INF. Also I will simplify your test case to [1,0,2,0,-1,3]

Iteration Onefirst = 1 second = INFIteration Twofirst = 0 second = INFIteration Threefirst = 0 second = 2Iteration Four (Nothing Changes)first = 0 second = 2Iteration Five (Confusing Part)first = -1 second = 2Iteration Sixreturn true; Since 3 > 2 && 3 > -1Setting first = -1 is important, yet doesn't change the answer in this case since second = 2 implies that their existed a value that was previously smaller than 2. Now if you find any value greater that 2 we know their exist in an increasing triplet sub sequence. But notice if we had a test case like this [1,0,2,0,-1,0,1] we now could see the importance of the updated lower bound for first = -1, so we we can have a correct lower bound for second = 0. And also note this answer ask for existence, and not to construct the triplet, as this solution wouldn't be able to in its current form

In this problem we just need to determine whether the subsequence exists. so after assigning the value to 'min' and 'secondMin', even though there might be a smaller value afterward and the variable 'min' gets updated, it does not affect the increasing subsequence, overall as long as there is an integer that is larger than 'secondMin'

**My way to approach such a problem. How to think about it? Explanation of my think flow.**

**Refer to**

<https://leetcode.com/problems/increasing-triplet-subsequence/solutions/79053/my-way-to-approach-such-a-problem-how-to-think-about-it-explanation-of-my-think-flow/>

I initially solved this problem by "thinking hard", so I came up with a convoluted solution (though greatly simplified when coding): <https://leetcode.com/discuss/105584/space-time-elegant-short-clean-solution-detailed-explanation>

Today, I revisited this problem. This time, I don't think about how to solve it, instead I want to think about "how to think about it".

Ok, so I read the description again, then I realize, it is asking about some sort of "increasing subsequence" with size 3.

Then I think about all the relevant algorithm I know, for example, the famous "Longest Increasing Subsequence" (LIS) problem.

Then I instantly got a solution: Find the LIS of the input, and if it is greater than 3, return true;Looks like a working solution, what's its complexity then:

There is a O(nlogk) solution to LIS (if you don't know it, just search this problem in Leetcode and see the discussions), where n is the array length and k is the length of LIS. Here, k is no larger than 2, so it is O(nlog2) = O(n). Very well, a O(n) solution is so easily obtained here:

class Solution {

public:

bool increasingTriplet(vector<int>& nums) {

vector<int> dp;

for (auto n : nums)

{

auto iter = lower\_bound(begin(dp), end(dp), n);

if (iter == end(dp))

{

dp.push\_back(n);

if (dp.size() == 3)

return true;

continue;

}

if (\*iter > n)

\*iter = n;

}

return false;

}

};

The only difference between LIS and this problem is the check "if (dp.size() == 3)"; For comparison, this is the code to return the LIS of the input nums: You can copy-paste it to the LIS problem and pass it actually.

vector<int> dp;

for (auto n : nums)

{

auto iter = lower\_bound(begin(dp), end(dp), n);

if (iter == end(dp))

{

dp.push\_back(n);

continue;

}

if (\*iter > n)

\*iter = n;

}

return dp.size();

Apparently, as you may have already noticed, the "dp" here contains at most 2 elements, so one instant simplification here is to replace "lower\_bound" call to a simple "if comparison else comparison". Then a much more simplified version is obtained:

class Solution {

public:

bool increasingTriplet(vector<int>& nums) {

int a = INT\_MAX, b = INT\_MAX;

for (auto n : nums)

if (n <= a)

a = n;

else if (n <= b)

b = n;

else

return true;

return false;

}

};

You may have seen 100 ways to explain why this "if .. else" works in other discussions. Here, it is so easy to understand: it is just a simple version of Binary Search for 2 elements -- the replacement of lower\_bound in above solution.

Following this think flow, I managed to come up with this elegant solution without any "hard thinking".