Previous Next

456. 132 Pattern

June 27, 2017 | 50.2K views

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 $a_i < a_k < a_j$. 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] and
```

Solution

Approach 1: Brute Force

The simplest solution is to consider every triplet (i, j, k) and check if the corresponding numbers satisfy the 132 criteria. If any such triplet is found, we can return a True value. If no such triplet is found, we need to return a False value.

Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        for (int i = 0; i < nums.length - 2; i++) {
            for (int j = i + 1; j < nums.length - 1; j++) {
                for (int k = j + 1; k < nums.length; k++) {
                    if (nums[k] > nums[i] && nums[j] > nums[k])
                        return true;
                }
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. Three loops are used to consider every possible triplet. Here, n refers to the size of $nums$ array.
- Space complexity : $O(1)$. Constant extra space is used.

Approach 2: Better Brute Force

Algorithm

We can improve the last approach to some extent, if we make use of some observations. We can note that for a particular number $nums[j]$ chosen as 2nd element in the 132 pattern, if we don't consider $nums[k]$ (the 3rd element) for the time being, our job is to find out the first element, $nums[i](i)$ which is lesser than $nums[j]$.

Now, assume that we have somehow found a $nums[i], nums[j]$ pair. Our task now reduces to finding out a $nums[k](Kk > j > i)$, which falls in the range $(nums[i], nums[j])$. Now, to maximize the likelihood of a $nums[k]$ falling in this range, we need to increase this range as much as possible.

Since, we started off by fixing a $nums[j]$, the only option in our hand is to choose a minimum value of $nums[i]$ given a particular $nums[j]$. Once, this pair $nums[i], nums[j]$, has been found out, we simply need to traverse beyond the index j to find if a $nums[k]$ exists for this pair satisfying the 132 criteria.

Based on the above observations, while traversing over the $nums$ array choosing various values of $nums[j]$, we simultaneously keep a track of the minimum element found so far(excluding $nums[j]$). This minimum element always serves as the $nums[i]$ for the current $nums[j]$. Thus, we only need to traverse beyond the j^{th} index to check the $nums[k]$'s to determine if any of them satisfies the 132 criteria.

Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        int min_i = Integer.MAX_VALUE;
        for (int j = 0; j < nums.length - 1; j++) {
            min_i = Math.min(min_i, nums[j]);
            for (int k = j + 1; k < nums.length; k++) {
                if (nums[k] < nums[j] && min_i < nums[k])
                    return true;
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two loops are used to find the $nums[j], nums[k]$ pairs. Here, n refers to the size of $nums$ array.
- Space complexity : $O(1)$. Constant extra space is used.

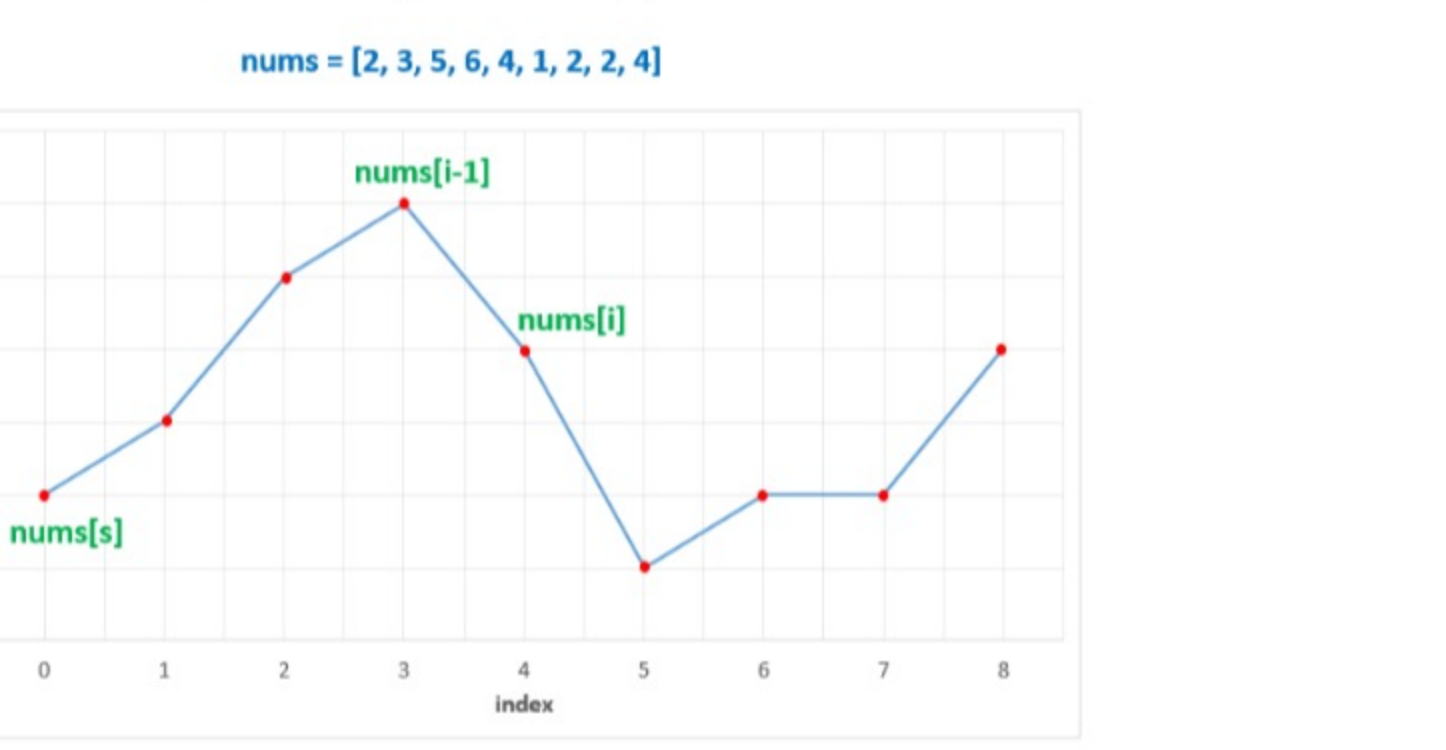
Approach 3: Searching Intervals

Algorithm

As discussed in the last approach, once we've fixed a $nums[i], nums[j]$ pair, we just need to determine a $nums[k]$ which falls in the range $(nums[i], nums[j])$. Further, to maximize the likelihood of any arbitrary $nums[k]$ falling in this range, we need to try to keep this range as much as possible. But, in the last approach, we tried to work only on $nums[i]$. But, it'll be a better choice, if we can somehow work out on $nums[j]$ as well.

To do so, we can look at the given $nums$ array in the form of a graph, as shown below:

nums = [2, 3, 5, 6, 4, 1, 2, 2, 4]



From the above graph, which consists of rising and falling slopes, we know, the best qualifiers to act as the $nums[i], nums[j]$ pair, as discussed above, to maximize the range $nums[i], nums[j]$, at any instant, while traversing the $nums$ array, will be the points at the endpoints of a local rising slope. Thus, once we've found such points, we can traverse over the $nums$ array to find a $nums[k]$ satisfying the given 132 criteria.

To find these points at the ends of a local rising slope, we can traverse over the given $nums$ array. While traversing, we can keep a track of the minimum point found after the last peak($nums[s]$).

Now, whenever we encounter a falling slope, say, at index i, we know, that $nums[i-1]$ was the endpoint of the last rising slope found. Thus, we can scan over the k indices(k>i), to find a 132 pattern.

But, instead of traversing over $nums$ to find a k satisfying the 132 pattern for every such rising slope, we can store this range $(nums[s], nums[i-1])$(acting as $(nums[i], nums[j])$) in, say an $intervals$ array.

While traversing over the $nums$ array to check the rising/falling slopes, whenever we find any rising slope, we can keep adding the endpoint pairs to this $intervals$ array. At the same time, we can also check if the current element falls in any of the ranges found so far. If so, this element satisfies the 132 criteria for that range.

If no such element is found till the end, we need to return a False value.

Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        List < int[] > intervals = new ArrayList < > ();
        int i = 1, s = 0;
        while (i < nums.length) {
            if (nums[i] <= nums[i - 1]) {
                if (s < i - 1)
                    intervals.add(new int[] {nums[s], nums[i - 1]});
                s = i;
            }
            for (int[] a: intervals)
                if (nums[i] > a[0] && nums[i] < a[1])
                    return true;
            i++;
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. We traverse over the $nums$ array of size n once to find the slopes. But for every element, we also need to traverse over the $intervals$ to check if any element falls in any range found so far. This array can contain atmost $(n/2)$ pairs, in the case of an alternate increasing-decreasing sequence(worst case e.g. `[5 6 4 7 3 8 2 9]`).
- Space complexity : $O(n)$. $intervals$ array can contain atmost $n/2$ pairs, in the worst case(alternate increasing-decreasing sequence).

Approach 4: Stack

Algorithm

In Approach 2, we found out $nums[i]$ corresponding to a particular $nums[j]$ directly without having to consider every pair possible in $nums$ to find this $nums[i], nums[j]$ pair. If we do some preprocessing, we can make the process of finding a $nums[k]$ corresponding to this $nums[i], nums[j]$ pair also easy.

The preprocessing required is to just find the best $nums[i]$ value corresponding to every $nums[j]$ value. This is done in the same manner as in the second approach i.e. we find the minimum element found till the j^{th} element which acts as the $nums[i]$ for the current $nums[j]$. We maintain thes values in a min array. Thus, $min[j]$ now refers to the best $nums[i]$ value for a particular $nums[j]$.

Now, we traverse back from the end of the $nums$ array to find the $nums[k]$'s. Suppose, we keep a track of the $nums[k]$ values which can potentially satisfy the 132 criteria for the current $nums[j]$. We know, one of the conditions to be satisfied by such a $nums[k]$ is that it must be greater than $nums[i]$. Or in other words, we can also say that it must be greater than $min[j]$ for a particular $nums[j]$ chosen.

Once it is ensured that the elements left for competing for the $nums[k]$ are all greater than $min[j]$(or $nums[i]$), our only task is to ensure that it should be lesser than $nums[j]$. Now, the best element from among the competitors, for satisfying this condition will be the minimum one from out of these elements.

If this element, $nums[min]$ satisfies $nums[min] < nums[j]$, we've found a 132 pattern. If not, no other element will satisfy this criteria, since they are all greater than or equal to $nums[min]$ and thus greater than or equal to $nums[j]$ as well.

To keep a track of these potential $nums[k]$ values for a particular $nums[i], nums[j]$ considered currently, we maintain a $stack$ on which these potential $nums[k]$'s satisfying the 132 criteria lie in a descending order(minimum element on the top). We need not sort these elements on the $stack$, but they'll be sorted automatically as we'll discuss along with the process.

After creating a min array, we start traversing the $nums[j]$ array in a backward manner. Let's say, we are currently at the j^{th} element and let's also assume that the $stack$ is sorted right now. Now, firstly, we check if $nums[j] > min[j]$. If not, we continue with the $(j-1)^{th}$ element and the $stack$ remains sorted. If not, we keep on popping the elements from the top of the $stack$ till we find an element, $stack[top]$ such that, $stack[top] > min[j]$(or $stack[top] > nums[i]$).

Once the popping is done, we're sure that all the elements pending on the $stack$ are greater than $nums[i]$ and are thus, the potential candidates for $nums[k]$ satisfying the 132 criteria. We can also note that the elements which have been popped from the $stack$, all satisfy $stack[top] \leq min[j]$.

Since, in the min array, $min[p] \leq min[q]$, for every $p > q$, these popped elements also satisfy $stack[top] \leq min[k]$, for all $0 \leq k < j$. Thus, they are not the potential $nums[k]$ candidates for even the preceding elements. Even after doing the popping, the $stack$ remains sorted.

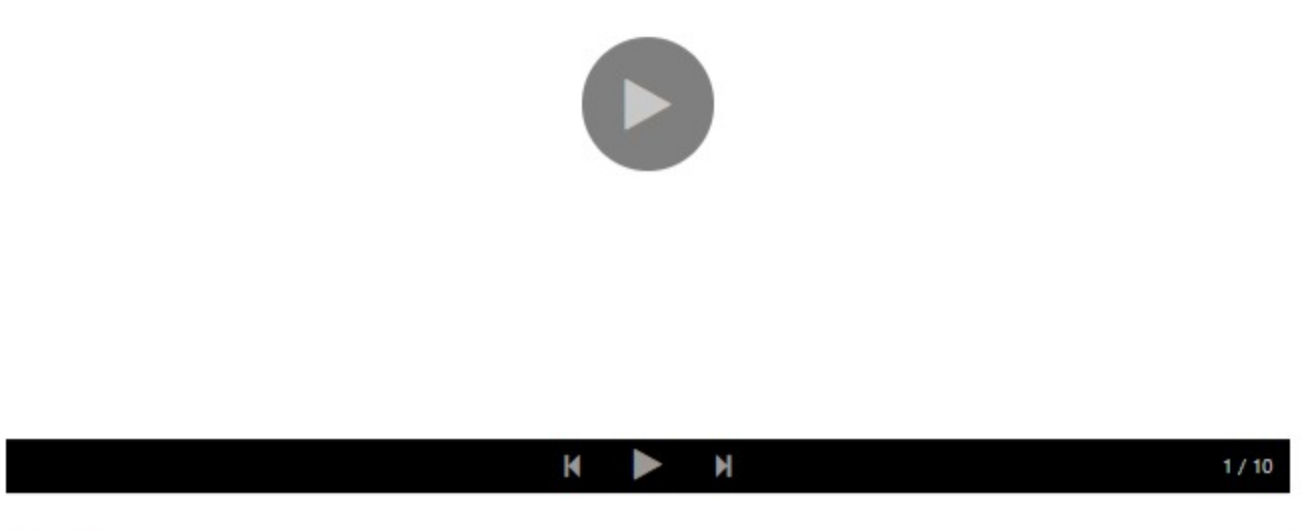
After the popping is done, we've got the minimum element from amongst all the potential $nums[k]$'s on the top of the $stack$(as per the assumption). We can check if it is greater than $nums[j]$ to satisfy the 132 criteria(we've already checked $stack[top] > nums[i]$). If this element satisfies the 132 criteria, we can return a True value. If not, we know that for the current j, $nums[j] > min[j]$. Thus, the element $nums[j]$ could be a potential $nums[k]$ value, for the preceding $nums[i]'s$.

Thus, we push it over the $stack$. We can note that, we need to push this element $nums[j]$ on the $stack$ only when it didn't satisfy $stack[top]$. Thus, $nums[j] \leq stack[top]$. Thus, even after pushing this element on the $stack$, the $stack$ remains sorted. Thus, we've seen by induction, that the $stack$ always remains sorted.

Also, note that in case $nums[j] \leq min[j]$, we don't push $nums[j]$ onto the $stack$. This is because this $nums[j]$ isn't greater than even the minimum element lying towards its left and thus can't act as $nums[k]$ in the future.

If no element is found satisfying the 132 criteria till reaching the first element, we return a False value.

The following animation better illustrates the process.



Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        if (nums.length < 3)
            return false;
        Stack < Integer > stack = new Stack < > ();
        int[] min = new int[nums.length];
        min[0] = nums[0];
        for (int i = 1; i < nums.length; i++)
            min[i] = Math.min(min[i - 1], nums[i]);
        for (int j = nums.length - 1; j >= 0; j--) {
            if (nums[j] > min[j]) {
                while (!stack.isEmpty() && stack.peek() <= min[j])
                    stack.pop();
                if (!stack.isEmpty() && stack.peek() < nums[j])
                    return true;
                stack.push(nums[j]);
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We travesre over the $nums$ array of size n once to fill the min array. After this, we traverse over $nums$ to find the $nums[k]$. During this process, we also push and pop the elements on the $stack$. But, we can note that atmost n elements can be pushed and popped off the $stack$ in total. Thus, the second traversal requires only $O(n)$ time.
- Space complexity : $O(n)$. The $stack$ can grow upto a maximum depth of n. Furhter, min array of size n is used.

Approach 5: Binary Search

Algorithm

In the last approach, we've made use of a separate $stack$ to push and pop the $nums[k]$'s. But, we can also note that when we reach the index j while scanning backwards for finding $nums[k]$, the $stack$ can contain atmost $n - j - 1$ elements. Here, n refers to the number of elements in $nums$ array.

We can also note that this is the same number of elements which lie beyond the j^{th} index in $nums$ array. We also know that these elements lying beyond the j^{th} index won't be needed in the future ever again. Thus, we can make use of this space in $nums$ array instead of using a separate $stack$. The rest of the process can be carried on in the same manner as discussed in the last approach.

We can try to go for another optimization here. Since, we've got an array for storing the potential $nums[k]$ values now, we need not do the popping process for a $min[j]$ to find an element just larger than $min[j]$ from amongst these potential values.

Instead, we can make use of Binary Search to directly find an element, which is just larger than $min[j]$ in the required interval, if it exists. If such an element is found, we can compare it with $nums[j]$ to check the 132 criteria. Otherwise, we continue the process as in the last approach.

Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        if (nums.length < 3)
            return false;
        int[] min = new int[nums.length];
        min[0] = nums[0];
        for (int i = 1; i < nums.length; i++)
            min[i] = Math.min(min[i - 1], nums[i]);
        for (int j = nums.length - 1, k = nums.length; j >= 0; j--) {
            if (nums[j] > min[j]) {
                k = Arrays.binarySearch(nums, k, nums.length, min[j] + 1);
                if (k < 0)
                    k = -1 - k;
                if (k < nums.length && nums[k] < nums[j])
                    return true;
                nums[--k] = nums[j];
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \log n)$. Filling min array requires $O(n)$ time. The second traversal is done over the whole $nums$ array of length n. For every current $nums[j]$ we need to do the Binary Search, which requires $O(\log n)$. In the worst case, this Binary Search will be done for all the n elements, and the required element won't be found in any case, leading to a complexity of $O(n \log n)$.
- Space complexity : $O(n)$. min array of size n is used.

Approach 6: Using Array as a Stack

Algorithm

In the last approach, we've seen that in the worst case, the required element won't be found for all the n elements and thus Binary Search is done at every step increasing the time complexity.

To remove this problem, we can follow the same steps as in Approach 4 i.e. We can remove those elements(update the index k) which aren't greater than $nums[i](min[j])$. Thus, in case no element is larger than $min[j]$ the index k reaches the last element.

Now, at every step, only $nums[j]$ will be added and removed from consideration in the next step, improving the time complexity in the worst case. The rest of the method remains the same as in Approach 4.

This approach is inspired by @fun4leetcode

Java

```java
public class Solution {
    public boolean find132pattern(int[] nums) {
        if (nums.length < 3)
            return false;
        int[] min = new int[nums.length];
        min[0] = nums[0];
        for (int i = 1; i < nums.length; i++)
            min[i] = Math.min(min[i - 1], nums[i]);
        for (int j = nums.length - 1, k = nums.length; j >= 0; j--) {
            if (nums[j] > min[j]) {
                while (k < nums.length && nums[k] <= min[j])
                    k++;
                if (k < nums.length && nums[k] < nums[j])
                    return true;
                nums[--k] = nums[j];
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We travesre over the $nums$ array of size n once to fill the min array. After this, we traverse over $nums$ to find the $nums[k]$. Atmost n elements can be put in and out of the $nums$ array in total. Thus, the second traversal requires only $O(n)$ time.
- Space complexity : $O(n)$. min array of size n is used.

Previous Next