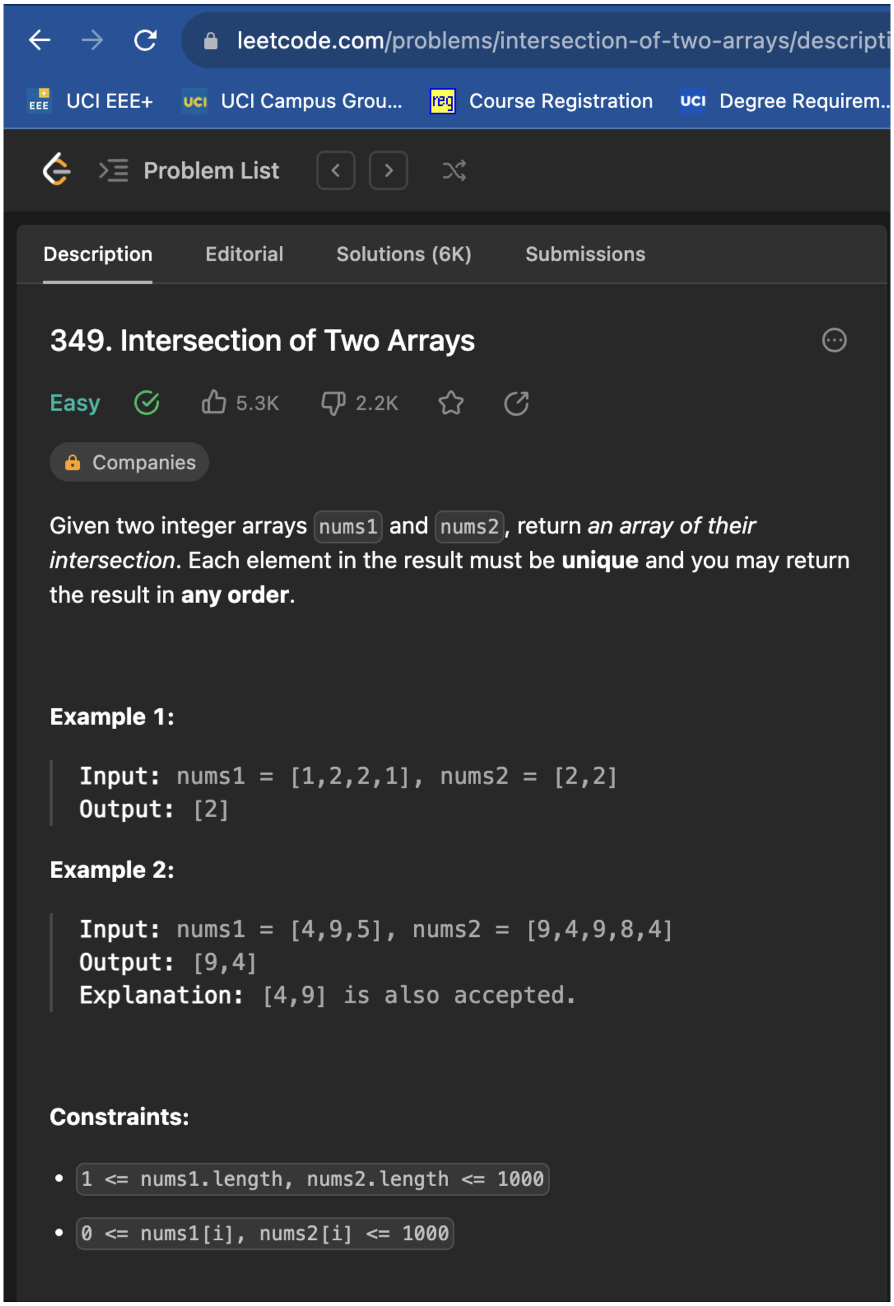
MCS – 253P ADVANCED PROGRAMMING AND PROBLEM SOLVING

HOMEWORK –3

Aswin Sampath
saswin@uci.edu

(53844684)

Q1) Intersection of Two Arrays



349. Intersection of Two Arrays

Given two integer arrays `nums1` and `nums2`, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

Example 1:

```
Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2]
```

Example 2:

```
Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
Output: [9,4]
Explanation: [4,9] is also accepted.
```

Constraints:

- `1 <= nums1.length, nums2.length <= 1000`
- `0 <= nums1[i], nums2[i] <= 1000`

Understanding the Problem:

The problem is about finding the intersection of two integer arrays, nums1 and nums2. The intersection should consist of unique elements from both arrays, and the order of elements in the result doesn't matter.

Identifying Edge Cases:

- Empty input arrays.
- The possibility of having duplicate values within each input array.
- The constraints on the lengths and values of the input arrays.

Effective Test Cases:

- Input: nums1 = [1,2,2,1], nums2 = [2,2]. Expected Output: [2].
- Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]. Expected Output: [4,9] or [9,4] (the order doesn't matter).
- Input: nums1 = [1,2,3], nums2 = [4,5,6]. Expected Output: [] (no common elements).
- Input: nums1 = [], nums2 = [1,2,3]. Expected Output: [] (one of the input arrays is empty).

Algorithmic Solution:

The code uses an unordered map to store the frequencies of elements in nums1, and then iterates through nums2 to check if each element is present in the map. The result is stored in a set to ensure uniqueness and is then converted back to a vector.

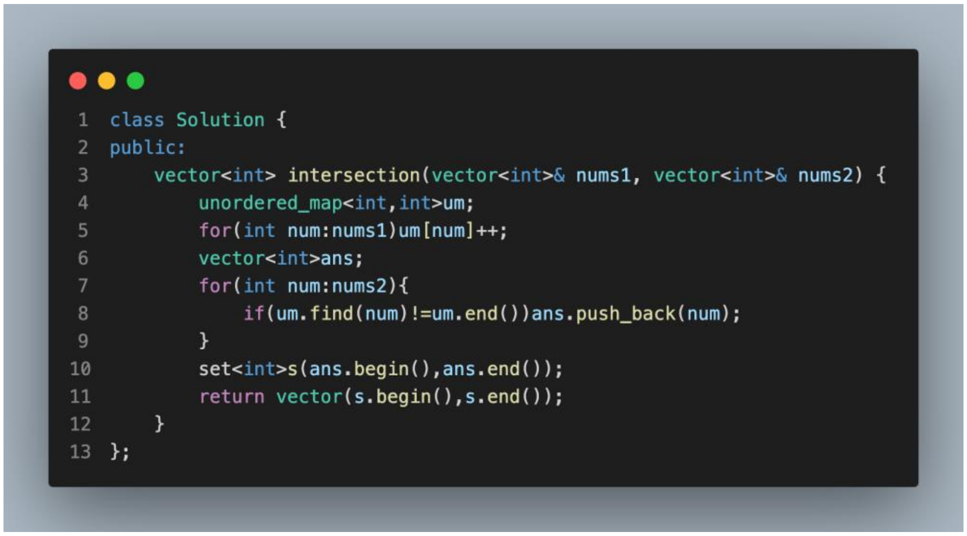
Time and Space Complexity Analysis:

Time Complexity: The code iterates through both nums1 and nums2, so the time complexity is O(m + n), where m is the length of nums1, and n is the length of nums2.

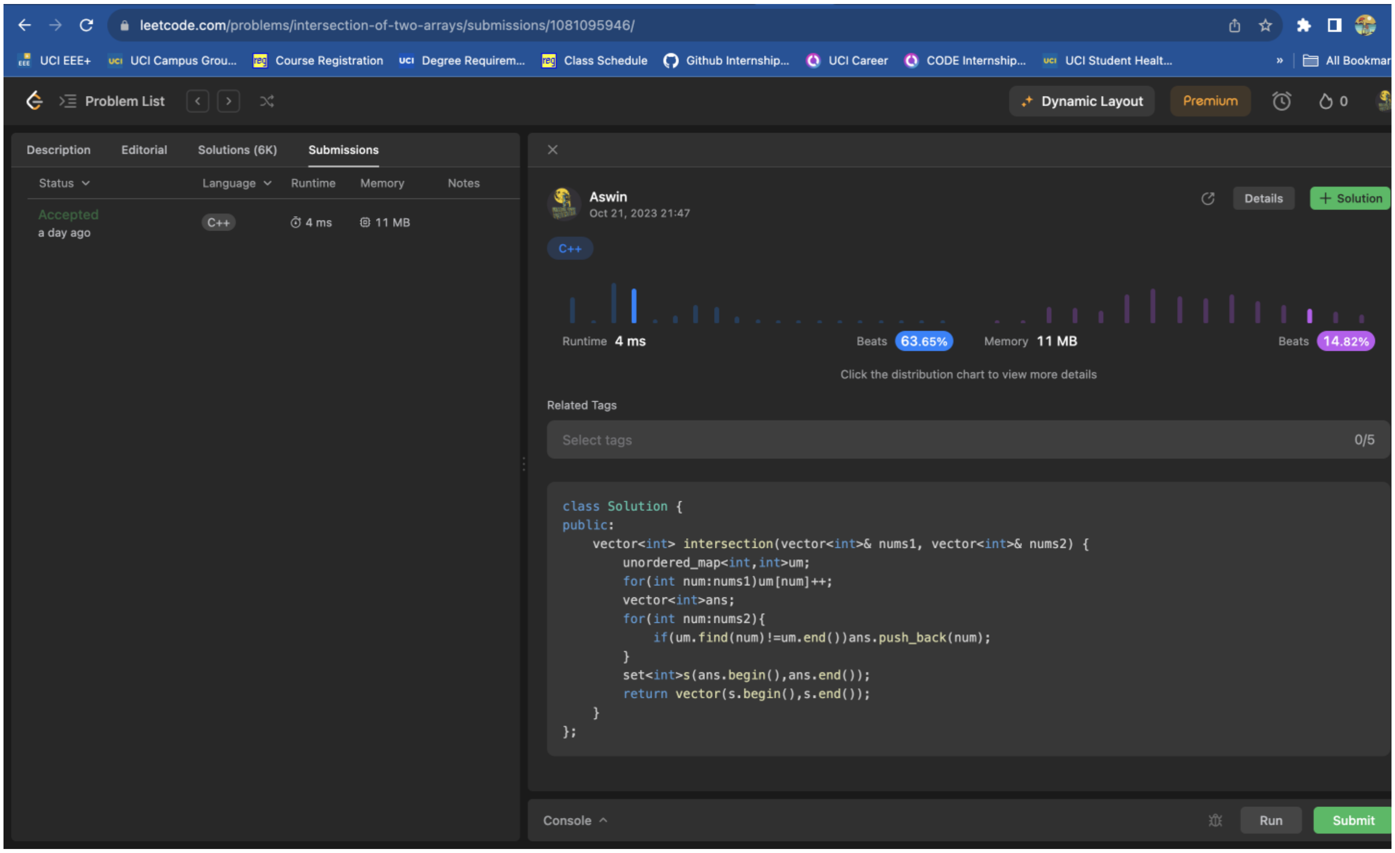
Space Complexity: The code uses an unordered map to store the frequencies of elements in `nums1, which requires O(m) space. The set for storing the result can also take up to O(m) space. So, the space complexity is O(m).

This code provides a correct solution to the problem, finding the intersection of two arrays and returning the result in any order.

Code:



```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int>um;
        for(int num:nums1)um[num]++;
        vector<int>ans;
        for(int num:nums2){
            if(um.find(num)!=um.end())ans.push_back(num);
        }
        set<int>s(ans.begin(),ans.end());
        return vector(s.begin(),s.end());
    }
};
```



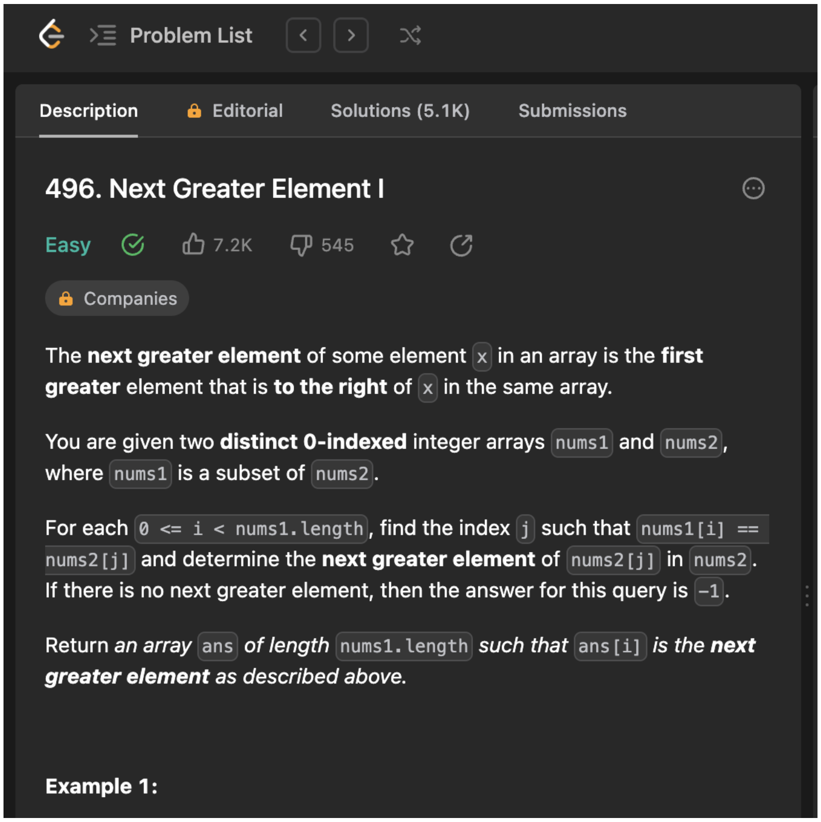
Accepted a day ago | C++ | 4 ms | 11 MB

Aswin
Oct 21, 2023 21:47
C++

Runtime 4 ms Beats 63.65%
Memory 11 MB Beats 14.82%

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int>um;
        for(int num:nums1)um[num]++;
        vector<int>ans;
        for(int num:nums2){
            if(um.find(num)!=um.end())ans.push_back(num);
        }
        set<int>s(ans.begin(),ans.end());
        return vector(s.begin(),s.end());
    }
};
```

Question 2) Next Greater Element I



496. Next Greater Element I

The **next greater element** of some element `x` in an array is the **first greater** element that is **to the right** of `x` in the same array.

You are given two **distinct 0-indexed** integer arrays `nums1` and `nums2`, where `nums1` is a subset of `nums2`.

For each `0 <= i < nums1.length`, find the index `j` such that `nums1[i] == nums2[j]` and determine the **next greater element** of `nums2[j]` in `nums2`. If there is no next greater element, then the answer for this query is `-1`.

Return *an array* `ans` *of length* `nums1.length` *such that* `ans[i]` *is the* ***next greater element*** *as described above.*

Example 1:

Understanding the Problem:

The problem involves two distinct integer arrays, nums1 and nums2, where nums1 is a subset of nums2. For each element in nums1, you need to find the index in nums2 and determine the next greater element in nums2 that appears to the right of that element. If there is no next greater element, the answer should be -1.

Identifying Edge Cases:

- Empty input arrays.
- Elements in nums1 not present in nums2.
- Duplicate values within the arrays.
- The possibility of no next greater element for an element in nums2.

Effective Test Cases:

Input: nums1 = [4, 1, 2], nums2 = [1, 3, 4, 2]. Expected Output: [3, 3, -1].

Explanation:

- For 4 in nums1, the next greater element in nums2 is 3.
- For 1 in nums1, the next greater element in nums2 is 3.
- For 2 in nums1, there is no next greater element in nums2.

Input: nums1 = [2, 4], nums2 = [1, 2, 3, 4]. Expected Output: [3, -1].

Explanation:

- For 2 in nums1, the next greater element in nums2 is 3.
- For 4 in nums1, there is no next greater element in nums2.

Input: nums1 = [3, 4], nums2 = [1, 2, 4, 3]. Expected Output: [4, -1].

Explanation:

- For 3 in nums1, the next greater element in nums2 is 4.
- For 4 in nums1, there is no next greater element in nums2.

Algorithmic Solution:

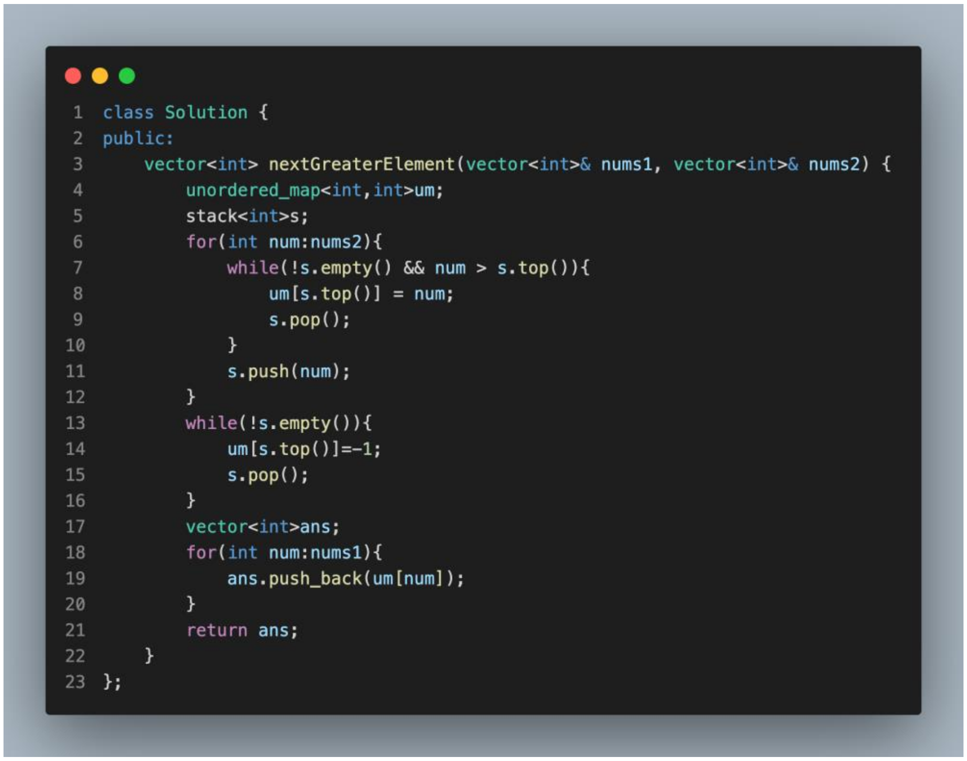
The code uses a stack to find the next greater element for each element in nums1. It iterates through nums2, maintaining a stack of elements in decreasing order. When it encounters an element greater than the top of the stack, it updates the map with the next greater element and pops elements from the stack. Finally, it iterates through nums1 to build the result.

Time and Space Complexity Analysis:

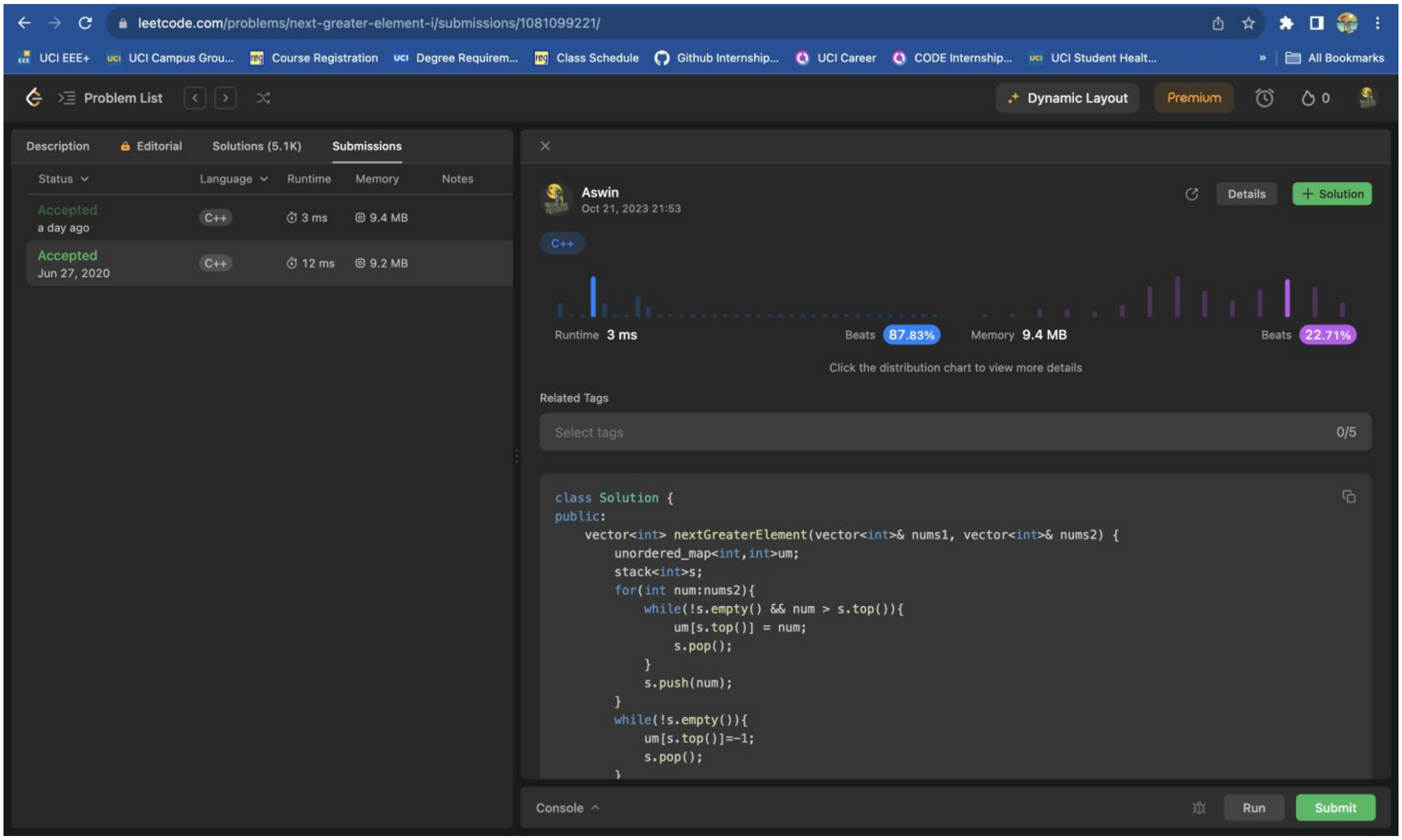
Time Complexity: The code iterates through both nums2 and nums1, so the time complexity is O(m + n), where m is the length of nums2, and n is the length of nums1.

Space Complexity: The code uses a stack to store elements, which can take up to O(m) space, and an unordered map to store the results, which takes up to O(m) space. So, the space complexity is O(m).

Code:



```cpp
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int>um;
        stack<int>s;
        for(int num:nums2){
            while(!s.empty() && num > s.top()){
                um[s.top()] = num;
                s.pop();
            }
            s.push(num);
        }
        while(!s.empty()){
            um[s.top()]=-1;
            s.pop();
        }
        vector<int>ans;
        for(int num:nums1){
            ans.push_back(um[num]);
        }
        return ans;
    }
};
```



Status	Language	Runtime	Memory	Notes
Accepted a day ago	C++	3 ms	9.4 MB	
Accepted Jun 27, 2020	C++	12 ms	9.2 MB	

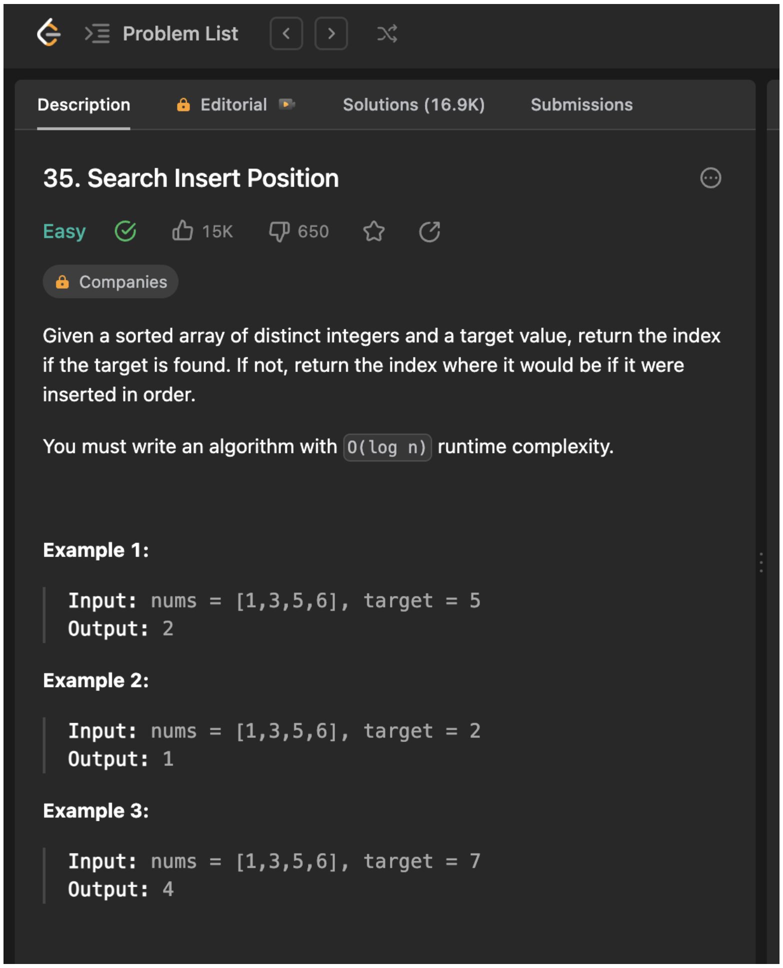
Aswin
Oct 21, 2023 21:53

Runtime 3 ms Beats 87.83%

Memory 9.4 MB Beats 22.71%

```
class Solution {
public:
    vector<int> nextGreaterElement(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int,int>um;
        stack<int>s;
        for(int num:nums2){
            while(!s.empty() && num > s.top()){
                um[s.top()] = num;
                s.pop();
            }
            s.push(num);
        }
        while(!s.empty()){
            um[s.top()]=-1;
            s.pop();
        }
```

Question 3) Search Insert Position



35. Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Understanding the Problem:

The problem asks for the index at which a target value should be inserted into a sorted array of distinct integers to maintain the sorted order. If the target is already present in the array, return its index. The goal is to achieve a runtime complexity of O(log n).

Identifying Edge Cases:

- Empty input array.
- Target value less than the smallest element in the array (should return 0).
- Target value greater than the largest element in the array (should return the last index + 1).

Effective Test Cases:

- Input: nums = [1, 3, 5, 6], target = 5. Expected Output: 2. (Target is present in the array.)
- Input: nums = [1, 3, 5, 6], target = 2. Expected Output: 1. (Insert 2 between 1 and 3.)
- Input: nums = [1, 3, 5, 6], target = 7. Expected Output: 4. (Insert 7 at the end.)

Algorithmic Solution:

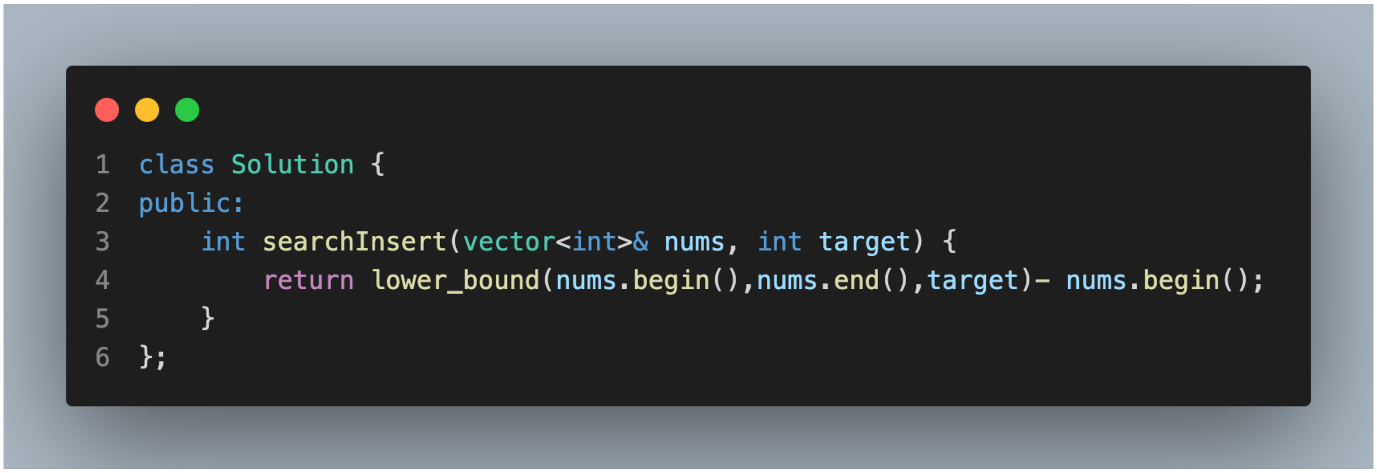
The code uses the C++ lower_bound function to find the position of the target value in the sorted array. lower_bound returns an iterator pointing to the first element in the range [nums.begin(), nums.end()) which is not less than target. By subtracting nums.begin(), it effectively returns the index where the target should be inserted.

Time and Space Complexity Analysis:

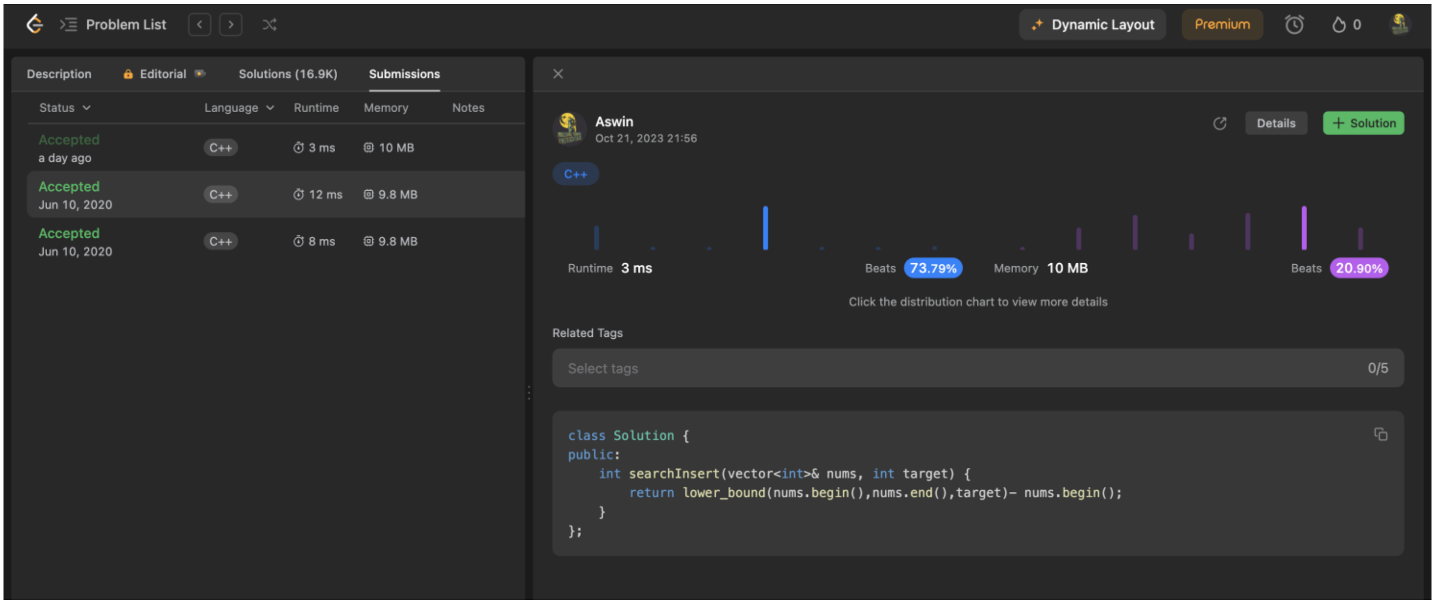
Time Complexity: The lower_bound function uses a binary search approach, which has a time complexity of O(log n) in a sorted array.

Space Complexity: The code uses only a few variables, so the space complexity is O(1).

Code:



```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        return lower_bound(nums.begin(),nums.end(),target)- nums.begin();
    }
};
```



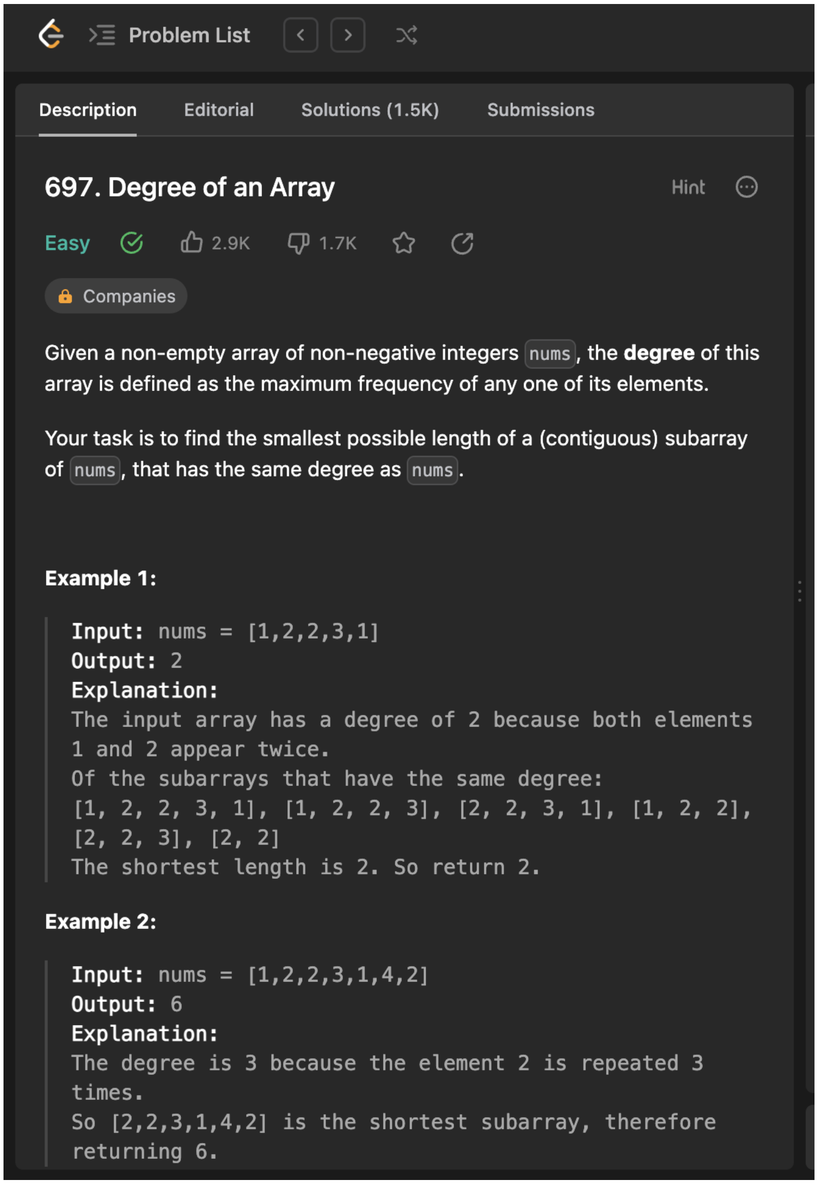
Status	Language	Runtime	Memory	Notes
Accepted a day ago	C++	3 ms	10 MB	
Accepted Jun 10, 2020	C++	12 ms	9.8 MB	
Accepted Jun 10, 2020	C++	8 ms	9.8 MB	

Aswin
Oct 21, 2023 21:56

C++

```cpp
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        return lower_bound(nums.begin(),nums.end(),target)- nums.begin();
    }
};
```

Question 4) Degree of an Array



697. Degree of an Array

Given a non-empty array of non-negative integers `nums`, the **degree** of this array is defined as the maximum frequency of any one of its elements.

Your task is to find the smallest possible length of a (contiguous) subarray of `nums`, that has the same degree as `nums`.

Example 1:

```
Input: nums = [1,2,2,3,1]
Output: 2
Explanation:
The input array has a degree of 2 because both elements
1 and 2 appear twice.
Of the subarrays that have the same degree:
[1, 2, 2, 3, 1], [1, 2, 2, 3], [2, 2, 3, 1], [1, 2, 2],
[2, 2, 3], [2, 2]
The shortest length is 2. So return 2.
```

Example 2:

```
Input: nums = [1,2,2,3,1,4,2]
Output: 6
Explanation:
The degree is 3 because the element 2 is repeated 3
times.
So [2,2,3,1,4,2] is the shortest subarray, therefore
returning 6.
```

Understanding the Problem:

The problem is about finding the smallest possible length of a contiguous subarray within a given array such that the subarray has the same degree as the original array. The degree of an array is defined as the maximum frequency of any of its elements.

Identifying Edge Cases:

- An empty input array.
- An input array with only one element.

Effective Test Cases:

- Input: nums = [1,2,2,3,1]. Expected Output: 2. (Degree is 2, and the shortest subarray with the same degree is [2,2].)
- Input: nums = [1,2,2,3,1,4,2]. Expected Output: 6. (Degree is 3, and the shortest subarray with the same degree is [2,2,3,1,4,2].)

- Input: nums = [1,2,2,3,1,4]. Expected Output: 5. (Degree is 2, and the shortest subarray with the same degree is [2,2,3,1,4].)

Algorithmic Solution:

The provided code uses an unordered map to store information about each element in the array, including its frequency and the starting and ending indices of its appearance. It iterates through the array, updating this information, and keeps track of the maximum frequency (degree). Afterward, it goes through the unordered map again to find the shortest subarray with the same degree.

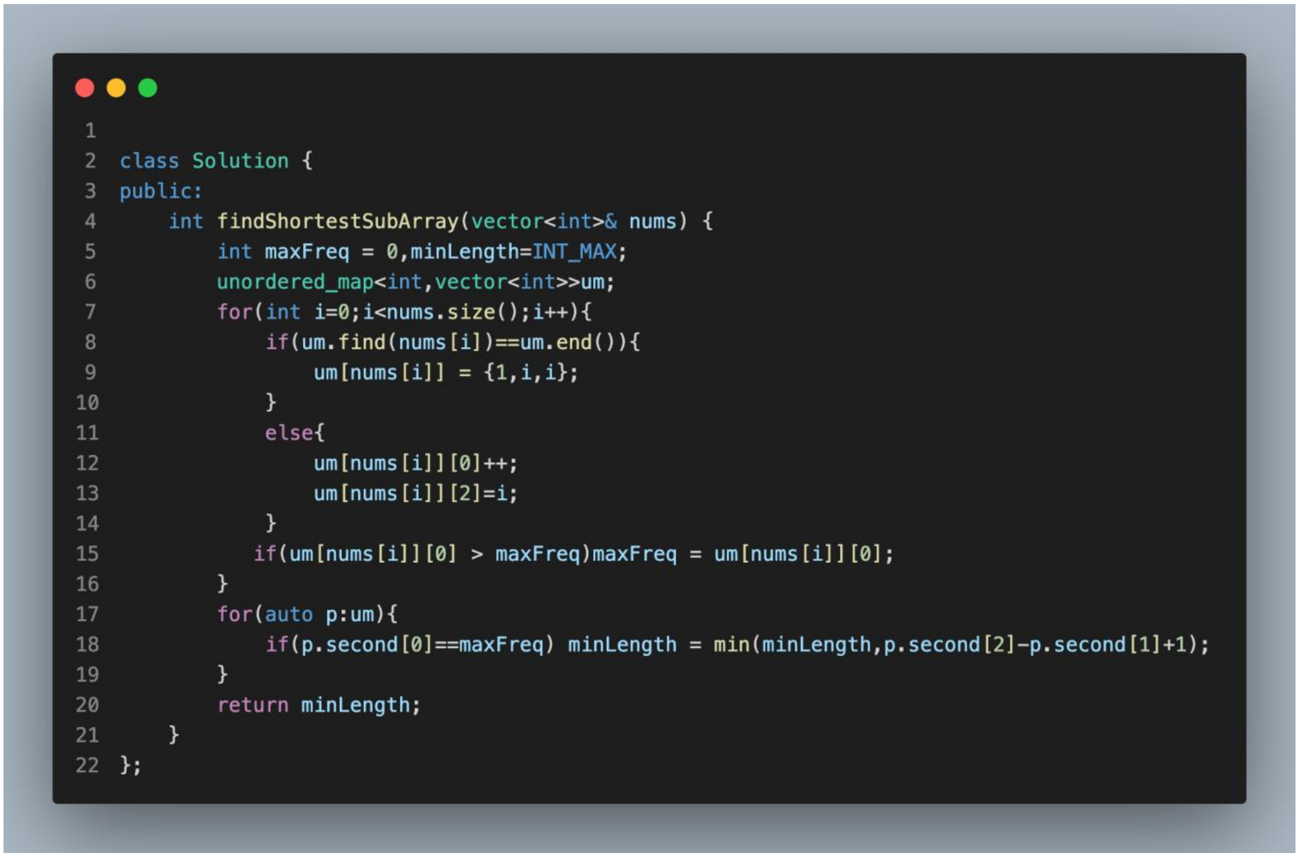
Time and Space Complexity Analysis:

Time Complexity: The code iterates through the input array twice: once to populate the unordered map, and once to find the shortest subarray. Therefore, the time complexity is O(n), where n is the length of the input array.

Space Complexity: The code uses an unordered map to store information about each element, so the space complexity is O(n) in the worst case.

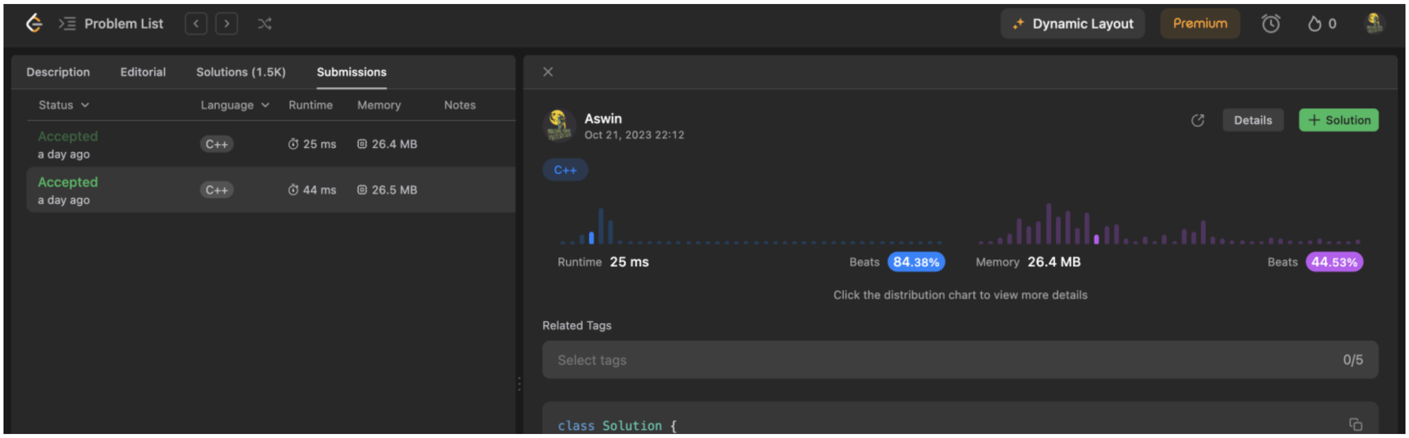
The provided code offers a correct solution for finding the smallest possible length of a subarray with the same degree as the original array.

Code:



```cpp

class Solution {
public:
    int findShortestSubArray(vector<int>& nums) {
        int maxFreq = 0,minLength=INT_MAX;
        unordered_map<int,vector<int>>um;
        for(int i=0;i<nums.size();i++){
            if(um.find(nums[i])==um.end()){
                um[nums[i]] = {1,i,i};
            }
            else{
                um[nums[i]][0]++;
                um[nums[i]][2]=i;
            }
           if(um[nums[i]][0] > maxFreq)maxFreq = um[nums[i]][0];
        }
        for(auto p:um){
            if(p.second[0]==maxFreq) minLength = min(minLength,p.second[2]-p.second[1]+1);
        }
        return minLength;
    }
};
```



Status	Language	Runtime	Memory	Notes
Accepted a day ago	C++	25 ms	26.4 MB	
Accepted a day ago	C++	44 ms	26.5 MB	

Aswin
Oct 21, 2023 22:12

Runtime 25 ms Beats 84.38%

Memory 26.4 MB Beats 44.53%

```
class Solution {
```