

217. Contains Duplicate

Easy

Topics

Companies

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

Example 1:

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

Output: `true`

Example 2:

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

Output: `false`

Example 3:

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

Output: `true`

Constraints:

- `1 <= nums.length <= 105`
- `-109 <= nums[i] <= 109`

Intuition:

The hash set approach uses a hash set data structure to store encountered elements. It iterates through the array, checking if an element is already in the set. If so, it returns true. Otherwise, it adds the element to the set. This approach has a time complexity of $O(n)$ and provides an efficient way to check for duplicates.

Explanation:

A more efficient approach is to use a hash set data structure to store the encountered elements. While iterating through the array, if an element is already present in the hash set, return true. Otherwise, add the element to the hash set. If the loop completes without finding any duplicates, return false.

CODE:

```
def containsDuplicate(nums: list[int]) -> bool:
    seen = set()
    for num in nums:
```

```
        if num in seen:
            return True
        seen.add(num)
    return False

nums = [1,1,1,3,3,4,3,2,4,2]
print(containsDuplicate(nums))
```


Output:



```
● PS C:\Users\91882\Desktop\CF\setup> & 'c:\msys64\ucrt64\bin\python.exe'
  acode.py'
  True
○ PS C:\Users\91882\Desktop\CF\setup>
```

Time Complexity:

T.C = $O(N)$, N = Size of nums

300. Longest Increasing Subsequence

Solved 

Medium  Topics  Companies

Given an integer array `nums`, return *the length of the longest **strictly increasing subsequence***.

Example 1:

Input: `nums = [10,9,2,5,3,7,101,18]`

Output: 4

Explanation: The longest increasing subsequence is `[2,3,7,101]`, therefore the length is 4.

Example 2:

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

Output: 4

Example 3:

Input: `nums = [7,7,7,7,7,7,7]`

Output: 1

Constraints:

- `1 <= nums.length <= 2500`
- `-104 <= nums[i] <= 104`

Follow up: Can you come up with an algorithm that runs in `O(n log(n))` time complexity?

Intuition

Lets say you have the input `[1,2,3,0,1,2,3]`

Obviously as you go through it, you're gonna want to keep track of the longest subsequence at any time.

So let's do that:

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

^

Seq: `[1]`

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

^

Seq: `[1,2]`

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

^

Seq: [1,2,3]

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

^

Seq: [1,2,3]

Now we have a problem - We know by looking at the input, the longest subsequence will use a 0. But if we add a 0 to our Seq, it won't be in order.

What we really want is to reduce the value at each index in the Seq array by as much as possible - That will make it easier to append onto the sequence later.

So let's just find where this element would go if Seq was a sorted array, and replace it:

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

^

Seq: [0,2,3] # We replaced the 1 with a 0!

(Wait! That's not a valid subsequence >:[- It doesn't matter, if you look closely the "valid" subsequence will always be when we're changing the last value in Seq)

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

^

Seq: [0,1,3] # We replaced the 2 with a 1!

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

^

Seq: [0,1,2]

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

^

Seq: [0,1,2,3] # A valid subsequence reappears!

You can test this with any input and it works.

(You might not "end" with a valid subsequence in your `Seq` array, but that doesn't actually matter, since any time you're appending the `Seq` is in a valid state. See the comments for more examples)

CODE:

```
import bisect

def lengthOfLIS(nums: list[int]) -> int:
    subseq = []
    for n in nums:
        index = bisect.bisect_left(subseq, n) # Binary search to find where
        the element would go in a sorted array
        if index >= len(subseq):
            subseq.append(n)
        else:
            subseq[index] = n
    return len(subseq)

nums = [10,9,2,5,3,7,101,18]
print(lengthOfLIS(nums))
```

OUTPUT:

```
PS C:\Users\91882\Desktop\CF\setup> c:; cd 'c:\Users\91882\Desktop\CF\setup'
2' '--' 'c:\Users\91882\Desktop\CF\setup\dacode.py'
4
PS C:\Users\91882\Desktop\CF\setup> []
```

Time Complexity:

T.C = $O(N \cdot \log(N))$, where N = size of `nums`

1312. Minimum Insertion Steps to Make a String Palindrome

Hard

Topics

Companies

Hint

Given a string `s`. In one step you can insert any character at any index of the string.

Return the minimum number of steps to make `s` palindrome.

A **Palindrome String** is one that reads the same backward as well as forward.

Example 1:

Input: `s = "zzazz"`

Output: `0`

Explanation: The string "zzazz" is already palindrome we do not need any insertions.

Example 2:

Input: `s = "mbadm"`

Output: `2`

Explanation: String can be "mbdadb" or "mdbabdm".

Example 3:

Input: `s = "leetcode"`

Output: `5`

Explanation: Inserting 5 characters the string becomes "leetcodocteel".

Constraints:

- `1 <= s.length <= 500`
- `s` consists of lowercase English letters.

Approach:

The problem can be solved using dynamic programming. We can create a 2D table `dp`, where `dp[i][j]` represents the minimum number of insertions required to make the substring `s[i:j+1]` a palindrome. The base case is when `i=j`, where the substring is already a palindrome and no insertions are needed. If `s[i] == s[j]`, then the substring is already a palindrome and we can use the result of `dp[i+1][j-1]`. Otherwise, we need to insert either a character at index `i` or `j` to make them equal, so we take the minimum of `dp[i+1][j]` and `dp[i][j-1]` and add 1 to it.

We can fill the table `dp` using a bottom-up approach. We start with the smallest substrings (length 1) and work up to the entire string. The final answer is stored in `dp[0][n-1]`, where `n` is the length of the input string `s`.

However, we can further optimize the space complexity of the solution by using only one row of the 2D table dp at a time. We can iterate over the columns of the table in reverse order and update the row accordingly. By updating the row in reverse order and using a variable to store the previous value of dp[j], we avoid overwriting the values in the row before they are used in the current iteration.

Intuition:

The problem requires us to find the minimum number of insertions needed to make a string palindrome. We can approach this by breaking down the string into smaller substrings and computing the minimum number of insertions needed for each substring.

If we consider a substring s[i:j+1], we can make it a palindrome by inserting characters at the beginning or end of the substring. We can use dynamic programming to solve the problem by breaking down the string into smaller substrings and computing the minimum number of insertions needed for each substring. By computing the minimum number of insertions needed for each substring, we can find the minimum number of insertions needed to make the entire string palindrome.

The dynamic programming solution uses a 2D table to store the minimum number of insertions needed for each substring. By breaking down the string into smaller substrings and using the results of the smaller substrings to compute the results of the larger substrings, we can find the minimum number of insertions needed to make the entire string palindrome.

CODE:

```
def minInsertions(s: str) -> int:
    n = len(s)
    dp = [0] * n
    for i in range(n - 2, -1, -1):
        prev = 0
        for j in range(i + 1, n):
            temp = dp[j]
            if s[i] == s[j]:
                dp[j] = prev
            else:
                dp[j] = min(dp[j], dp[j-1]) + 1
            prev = temp
    return dp[n-1]

s = "leetcode"
print(minInsertions(s))
```

OUTPUT:

```
PS C:\Users\91882\Desktop\CF\setup>
PS C:\Users\91882\Desktop\CF\setup> c:; cd 'c:\Users\91882\Desktop\CF\setup'
6' '--' 'c:\Users\91882\Desktop\CF\setup\dacode.py'
5
PS C:\Users\91882\Desktop\CF\setup> 
```

TC Analysis:

Time: $O(n^2)$, space: $O(n)$, where $n = s.length()$