284. Peeking Iterator Medium Design Array Iterator **Leetcode Link** 

### **Problem Description** The task is to design a PeekingIterator that extends the functionality of a standard iterator. Unlike a regular iterator that only

to look at the next element the iterator would return without actually moving the iterator forward. This can be particularly useful when you need to make decisions based on the next item without actually removing it from the iteration sequence. Intuition

supports next and hasNext operations, the PeekingIterator should also support a peek operation. The peek method allows the user

The intuitive approach to solving this problem involves caching the next element of the iterator after a peek call, because a regular iterator would not allow us to see the next element without advancing to it. Thus, we need an additional state in PeekingIterator to remember whether we've peeked at the next element and what that element is.

1. Initialize State: When we create a PeekingIterator, we store the original Iterator and initialize a state indicating whether the iterator has been peeked or not (has\_peeked), and a variable to store the peeked value (peeked\_element).

Here is a step-by-step break-down of the intuition behind the solution:

peeked yet (has\_peeked is False), we call next on the underlying iterator and store the value. We then return this stored value without changing the position of the underlying iterator.

2. Peek Operation: For the peek function we check if we have already called a peek without a subsequent next. If we haven't

3. Next Operation: If a peek has not occurred, the regular next operation of the underlying iterator is used. If we've already peeked (has\_peeked is True), we need to return the stored peeked\_element and reset our state so the iterator can advance when next is called again.

4. HasNext Operation: For the hasNext method, we simply return True if we are in the peeked state (as we already have a next

- element), or we defer to the underlying iterator's hasNext method. The critical part of this solution is to carefully manage the has\_peeked and peeked\_element variables so as to not lose track of the iterator's state or prematurely advance the underlying iterator.
- The solution involves implementing a wrapper class around the provided iterator to support the additional peek operation. We make use of the object-oriented concept of class to create our own PeekingIterator that maintains some internal state.

## variables:

**Solution Approach** 

self.has\_peeked: A boolean that indicates if we have already peeked at the next element.

• self.peeked\_element: An integer to hold the next element after a peek. These variables are critical to manage the state

1. Initialization: In the constructor (\_\_init\_\_ method), we accept an Iterator as an argument. We then initialize two instance

2. Peek: The peek method checks if we have already peeked (self.has\_peeked), and if not, it 'peeks' ahead by calling next on the stored iterator, saving this value to self.peeked\_element and setting self.has\_peeked to True. The value of self.peeked\_element

the next item without yet removing it from the iterator.

Let's walk through an example to illustrate the solution approach:

between successive peek and next calls.

Here's a step-by-step walk through the implementation:

is returned. 3. Next: In the next method, we first check if self.has\_peeked is True. If it is, we already have a peeked element to return, which

value. If self.has\_peeked is False, we directly return the next element from the underlying iterator.

4. HasNext: For hasNext, we return True if either self.has\_peeked is True (since we have a peeked element ready) or if the underlying iterator has more elements, as indicated by its hasNext method.

we do. We then reset self.has\_peeked to False and self.peeked\_element to None to reflect that we no longer have a peeked

method is called). This strategy minimizes the number of calls to the underlying iterator's next method, as it only advances the iterator when we're certain we need to consume an element. This approach effectively implements a look-ahead buffer with a capacity of one element.

Such an implementation could be useful in scenarios like comparison-based sorting or merging, where you might need to compare

This implementation uses lazy evaluation for the peek operation. The element is not advanced until necessary (i.e., until the next

Suppose we have an iterator over a collection with elements [1, 2, 3]. When we wrap this iterator with the PeekingIterator, here's how we can interact with it and what happens internally: 1. We instantiate PeekingIterator with our iterator. self.has\_peeked is initialized to False, and self.peeked\_element is initialized to

2. We call peek() method. Since self.has\_peeked is False, the next() method is called on the underlying iterator, giving us 1. Now

self.peeked\_element becomes 1, and self.has\_peeked is now True. The peek() method returns 1, but our iterator's position

3. We call the next() method. As self.has\_peeked is True, it will return self.peeked\_element, which is 1. After this,

### self.has\_peeked is set to False and self.peeked\_element to None. The underlying iterator's next element is now 2.

method returns 2.

so hasNext() returns True.

hasn't advanced.

None.

iterator.

11

20

24

25

26

27

29

30

Python Solution

# A class that represents an iterator.

def \_\_init\_\_(self, nums):

def has\_next(self):

class Iterator:

Example Walkthrough

4. We call peek() again. The process from step 2 repeats; self\_peeked\_element is set to 2, self\_has\_peeked becomes True, and the

self.peeked\_element are reset as before. 6. We call hasNext(). Since self.has\_peeked is False, we check with the underlying iterator, which still has an element (which is 3),

5. We call next(). Since self.has\_peeked is True, self.peeked\_element (which is 2) is returned. Both self.has\_peeked and

further elements. This example did not include calls to hasNext() prior to next() or peek() calls, but in practice, the hasNext() method can be used anytime to check if there is a next element available before calling next() or peek() to avoid exceptions from reaching the end of the

8. Finally, when we call hasNext() again, it returns False because both self.has\_peeked is False and the underlying iterator has no

:param nums: A list of integers. # Your Iterator class here.

Returns the next element in the iteration.

:return: The next integer in the iteration.

# Your next logic here.

def \_\_init\_\_(self, iterator):

Initialize the iterator object to the beginning of a list.

7. We call next() and receive 3, which is the last element of the iterator.

Checks if the iteration has more elements. 14 :return: True if there are more elements, False otherwise. # Your has\_next logic here. 18 def next(self): 19

### 31 Initialize a PeekingIterator with an iterator. 32 33 :param iterator: An instance of the Iterator class. 34

class PeekingIterator:

```
35
           self.iterator = iterator
36
           self.peeked_element = None # The element that has been peeked at.
37
           self.has_peeked = False
                                      # Flag to check if we have a peeked element.
38
39
       def peek(self):
40
           Returns the next element in the iteration without advancing the iterator.
41
42
43
           :return: The next integer in the iteration.
44
45
           if not self.has_peeked:
               # If we haven't peeked yet, retrieve the next element
46
               self.peeked_element = self.iterator.next()
47
               self.has_peeked = True # Set the flag to indicate that we have peeked
48
           return self.peeked_element
49
50
       def next(self):
51
52
53
           Returns the next element and advances the iterator.
54
55
           :return: The next integer in the iteration.
56
57
           if not self.has_peeked:
               # If we haven't peeked, we simply get the next element from the iterator
58
59
               return self.iterator.next()
60
61
           # If we have peeked, we return the peeked element and reset the peeked state
           result = self.peeked_element
62
           self.peeked_element = None
63
           self.has_peeked = False
           return result
       def has_next(self):
67
68
69
           Checks if there are more elements in the iteration.
70
           :return: True if there are more elements, False otherwise.
73
           # We have an element if we already peeked or if the iterator has more elements
           return self.has_peeked or self.iterator.has_next()
74
75
76
77 # Usage example
78 # iter = PeekingIterator(Iterator(nums))
  # while iter.has_next():
         val = iter.peek() # View the next element but do not advance the iterator.
         iter.next()
                             # Should return the same value as [val].
81 #
82
Java Solution
   import java.util.Iterator;
   // The PeekingIterator class wraps a standard Iterator and allows for peeking ahead at the next element without advancing the iterato
   public class PeekingIterator implements Iterator<Integer> {
       private Iterator<Integer> iterator; // An iterator over a collection of Integers
       private boolean hasPeeked; // A flag indicating if we've already peeked at the next element
       private Integer peekedElement; // The next element, if we've peeked
       // Constructor initializes the PeekingIterator using an existing Iterator.
       public PeekingIterator(Iterator<Integer> iterator) {
10
           this.iterator = iterator;
12
13
14
       // The peek method returns the next element in the iteration without advancing the iterator.
       public Integer peek() {
15
           if (!hasPeeked) {
               if (iterator.hasNext()) {
                   peekedElement = iterator.next(); // Store the next element so we can return it later
19
               } else {
20
                   throw new NoSuchElementException(); // In case there are no more elements to peek
               hasPeeked = true;
24
           return peekedElement;
25
26
       // The next method should behave the same as if using the Iterator directly.
28
       @Override
       public Integer next() {
29
           if (!hasPeeked) {
               return iterator.next(); // Directly return the next element from the iterator
31
32
33
           Integer result = peekedElement; // Return the stored peeked element
           hasPeeked = false; // Reset the peeked flag
34
           peekedElement = null; // Clear the stored element
35
           return result;
36
```

return hasPeeked || iterator.hasNext(); // Return true if we've peeked or if the iterator has more elements

// Flag to check if we already peeked at the next element

// Constructor inherits from the Iterator and initializes hasPeeked flag to false.

// Override the next() method to consider if we have already peeked at the next element.

// If we haven't peeked, return the next element from the iterator directly

// Note that initialization of hasPeeked is done above in the member initializer list.

// Peek method allows us to see what the next element in the iterator is without advancing the iterator.

// Set the peek flag

// Stores the value of the peeked element if hasPeeked is true

### 15 int peek() { if (!hasPeeked) { 16 // If we haven't already peeked, fetch the next element and set the hasPeeked flag 17 peekedValue = Iterator::next(); // Get the next element from the iterator 18 hasPeeked = true; 20

int next() {

if (!hasPeeked) {

return Iterator::next();

37

38

39

40

43

45

9

10

11

12

13

14

21

22

23

24

25

26

27

28

44 }

@Override

C++ Solution

bool hasPeeked;

int peekedValue;

private:

public:

public boolean hasNext() {

1 class PeekingIterator : public Iterator {

PeekingIterator(const vector<int>& nums)

: Iterator(nums), hasPeeked(false)

return peekedValue; // Return the peeked value

// The hasNext method indicates whether the iteration has more elements.

```
29
           // If there is a peeked value, reset the hasPeeked flag since next() consumes the peeked element
30
           hasPeeked = false;
31
32
           return peekedValue; // Return the peeked value which is the next element
33
34
35
       // Override the hasNext() method to consider the peek scenario.
       bool hasNext() const {
36
           // If either we have a peeked element or if the underlying iterator has a next element,
37
           // we have a next element
           return hasPeeked || Iterator::hasNext();
39
40
41 };
42
Typescript Solution
 1 let hasPeeked: boolean = false;
                                              // Flag to check if we already peeked at the next element
   let peekedValue: number;
                                              // Stores the value of the peeked element if hasPeeked is true
   let iterator: Iterator<number>;
                                              // The iterator instance which is adapted by the PeekingIterator functionality
   function PeekingIterator(nums: number[]) {
       // Initialize the iterator with nums.
       iterator = nums[Symbol.iterator]();
8 }
 9
   function peek(): number {
       if (!hasPeeked) {
           // If we haven't already peeked, fetch the next element and set the hasPeeked flag
13
           const result = iterator.next();
           if (!result.done) {
14
               peekedValue = result.value;
15
               hasPeeked = true;
                                               // Set the peek flag
16
17
18
       return peekedValue; // Return the peeked value
19
20 }
21
   function next(): number {
23
       if (!hasPeeked) {
           // If we haven't peeked, extract the next element from the iterator directly and return it
           const result = iterator.next();
26
           if (!result.done) {
               return result.value;
28
       } else {
29
           // If there is a peeked value, reset the hasPeeked flag since next() consumes the peeked element
30
           hasPeeked = false;
31
32
            return peekedValue; // Return the peeked value which is the next element
33
34
35
```

// If there is a peeked element or if the underlying iterator has a next element, we have a next element

# **Time Complexity:**

function hasNext(): boolean {

Time and Space Complexity

return hasPeeked || !iterator.next().done;

operations are involved. PeekingIterator.peek(self): • If we have not peeked yet (self.has\_peeked is False), we call self.iterator.next() once. Since the next() operation of the

PeekingIterator.\_\_init\_\_(self, iterator):

- element, we just return it, which is again an 0(1) operation.
- PeekingIterator.next(self): • When we haven't peeked (self.has\_peeked is False), the time complexity is 0(1) as it directly returns the next element by
- calling the iterator's next() method. Once we have peeked (self.has\_peeked is True), it involves returning the peeked element and resetting has\_peeked and peeked\_element which is also 0(1) time. PeekingIterator.hasNext(self):
- hasNext() method of the underlying iterator, both of these operations have a time complexity of 0(1). **Space Complexity:**

This method checks whether there's a next element by either checking the self.has\_peeked attribute or calling the

The additional space used by the PeekingIterator class consists of two variables: self.has\_peeked and

This method only performs some basic variable assignments. The time complexity here is 0(1), since no loops or significant

underlying iterator is assumed to be 0(1), the overall time complexity for peek() is also 0(1). If we already have a peeked

self.peeked\_element, which take up 0(1) space. The underlying iterator (self.iterator) object is not counted towards the

Overall space complexity of the PeekingIterator class:

space complexity as it is an input to the class. Therefore, the space complexity of the PeekingIterator is 0(1) as it uses a constant amount of extra space. In conclusion, both the operations peek(), next(), and hasNext() in the PeekingIterator class have a time complexity of 0(1) and the space complexity of the PeekingIterator class is 0(1).