Previous Next

251. Flatten 2D Vector

March 21, 2020 | 4K views

Design and implement an iterator to flatten a 2d vector. It should support the following operations: `next` and `hasNext`.

Example:

```
Vector2D iterator = new Vector2D([[1,2],[3],[4]]);

iterator.next(); // return 1
iterator.next(); // return 2
iterator.next(); // return 3
iterator.hasNext(); // return true
iterator.hasNext(); // return true
iterator.next(); // return 4
iterator.hasNext(); // return false
```

Notes:

1. Please remember to **RESET** your class variables declared in Vector2D, as static/class variables are **persisted across multiple test cases**. Please see here for more details.
2. You may assume that `next()` call will always be valid, that is, there will be at least a next element in the 2d vector when `next()` is called.

Follow up:

As an added challenge, try to code it using only iterators in C++ or iterators in Java.

Solution

Overview

This question should be fairly straightforward if you're familiar with what an Iterator is. If you aren't at all familiar with Iterators though, then we suggest having a go at Peeking Iterator. Additionally, the Solution Article for Peeking Iterator has a special introduction section that introduces you to what Iterators are.

Note that this question refers to something called a `Vector`. A `Vector` is simply another name for `Array`. To be consistent with the question, we've chosen to use the term `Vector`, rather than `Array` for this article (Sorry in advance for any confusion this causes, C++ programmers).

Approach 1: Flatten List in Constructor

Intuition

This approach is a bad approach! *We've included it though, to show what it looks like, and to discuss why it's bad. This will help you to design* ***good*** *Iterators.*

In the constructor, we can iterate over the 2D input vector, putting each integer into a `List`. Then, the problem simplifies to being a simple `List` Iterator. Note that the reason we use a `List` rather than an array (vector) is because we don't know in advance how many integers there might be in total.

Our unpack algorithm would be as follows.

```
nums = a new List
for each innerVector in the input 2D Vector:
    for each number in innerVector:
        append number to the end of nums
```

We'll then need to save this `List` as a field of our Iterator class, seeing as the `next(...)` and `hasNext(...)` methods will need to access it repeatedly. By then also having a position field, we can keep track of where the Iterator is up to.

Algorithm

The code shown here makes the `position` field point at the *next element that needs to be returned by* `next`. Therefore, the `hasNext()` method simply needs to check that `position` is a valid index of `nums`. A similar variant would be to make `position` point at the *previous* value that was returned. This would simplify the `next()` method, but complicate the `hasNext()` method.

Java | Python | Copy

```
1  class Vector2D:
2  
3      def __init__(self, v: List[List[int]]):
4          # We need to iterate over the 2D vector, getting all the integers
5          # out of it and putting them into the nums list.
6          for (int[] innerVector : v) {
7          self.nums = []
8          for inner_list in v:
9              for num in inner_list:
10                 self.nums.append(num)
11         # We'll keep position 1 behind the next number to return.
12         self.position = -1
13 
14     def next(self) -> int:
15         # Move up to the current element and return it.
16         self.position += 1
17         return self.nums[self.position]
18 
19     def hasNext(self) -> bool:
20         # If the next position is a valid index of nums, return True.
21         return self.position + 1 < len(self.nums)
```

Complexity Analysis

Let N be the number of integers within the 2D Vector, and V be the number of inner vectors.

- Time complexity.
  - **Constructor:** $O(N+V)$.

    In total, we'll append N integers to the `nums` list. Each of these appends is an $O(1)$ operation. This gives us $O(N)$.

    Something to be cautious of is that inner vectors don't *have* to contain integers. Think of a test cases such as `[[], [2], [], [], []]`. For this test case, $N = 1$, because there's only one integer within it. *However*, the algorithm has to loop through all of the empty vectors. The cost of checking all the vectors is $O(V)$.

    Therefore, we get a final time complexity of $O(N+V)$.
  - **next():** $O(1)$.

    All operations in this method, including getting the integer at a specific index of a list, are $O(1)$.
  - **hasNext():** $O(1)$.

    All operations in this method are $O(1)$.
- Space complexity : $O(N)$.

  We're making a new list that contains all of the integers from the 2D Vector. Notice that this is different from the time complexity; in the example of `[[], [2], [], [], []]`, we only store the `2`. All information about how many inner vectors there were is discarded.

Why is this implementation bad?

This code works, it runs fast here on Leetcode, it seems pretty straightforward to implement.

However, one of the main purposes of an Iterator is to *minimize* the use of auxiliary space. We should try to utilize the existing data structure as much as possible, only adding as much extra space as needed to keep track of the next value. In some situations, the data structure we want to iterate over is too large to even fit in memory anyway (think of file systems).

In the case of our above implementation, we might as well have just had a single function `List<Integer> getFlattenedVector(int[][] v)`, which would return a `List` of integers, that could then be iterated over using the `List` types own standard Iterator.

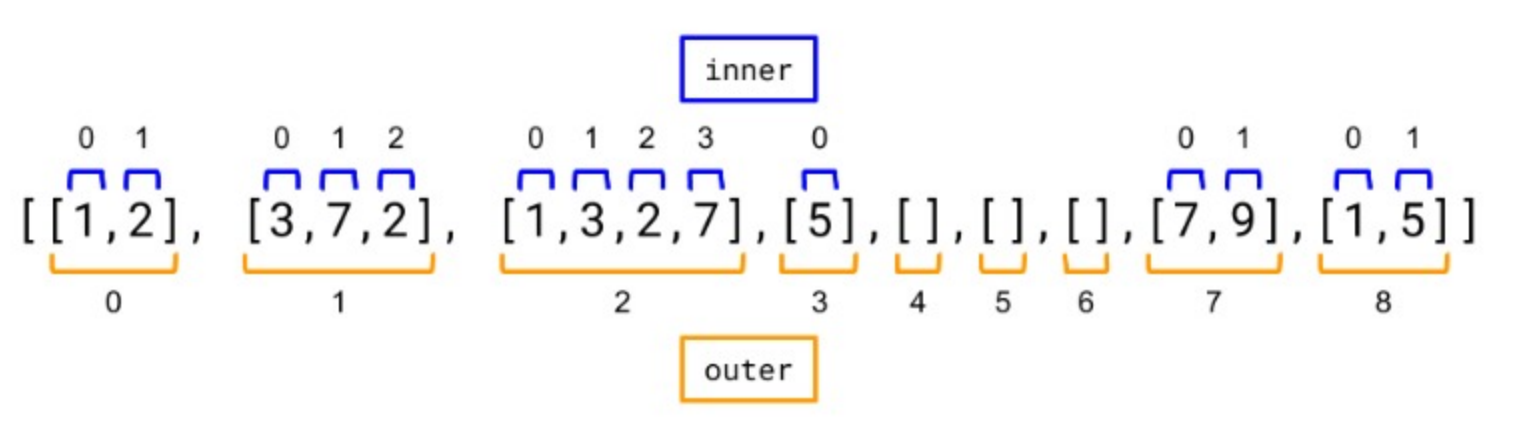
As a general rule, you should be very cautious of implementing Iterators with a high time complexity in the constructor, with a very low time complexity in the `next()` and `hasNext()` methods. If the code using the Iterator only wanted to access the first couple of elements in the iterated collection, then a lot of time (and probably space) has been wasted!

As a side note, modifying the input collection in any way is *bad* design too. Iterators are only allowed to look at, not change, the collection they've been asked to iterate over.

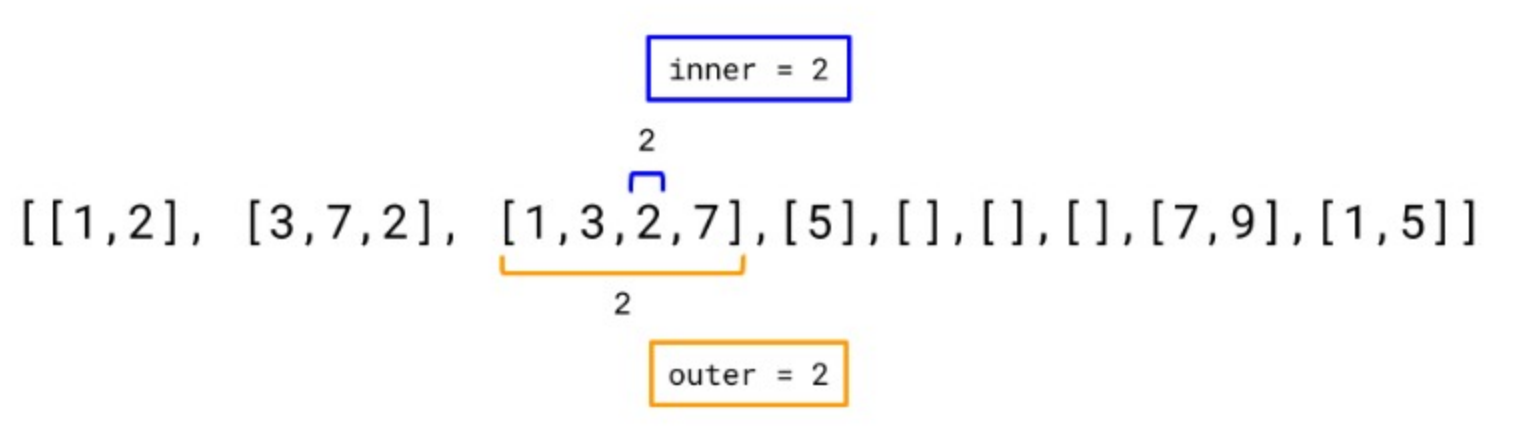
Approach 2: Two Pointers

Intuition

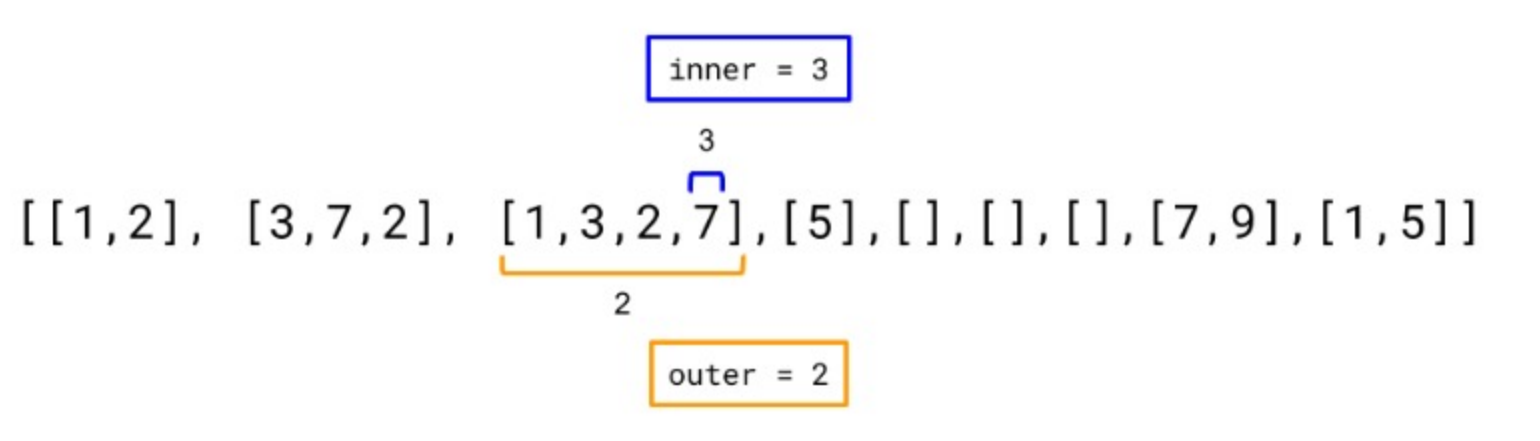
Like we said above, Approach 1 is bad because it creates a new data structure instead of simply iterating over the given one. Instead, we should find a way to step through the integers one-by-one, keeping track of where we currently are in the 2D vector. The location of each number is represented with 2 indexes; the index of the inner vector, and the index of the integer within its inner vector. Here's an example 2D vector, with the indexes marked on it.



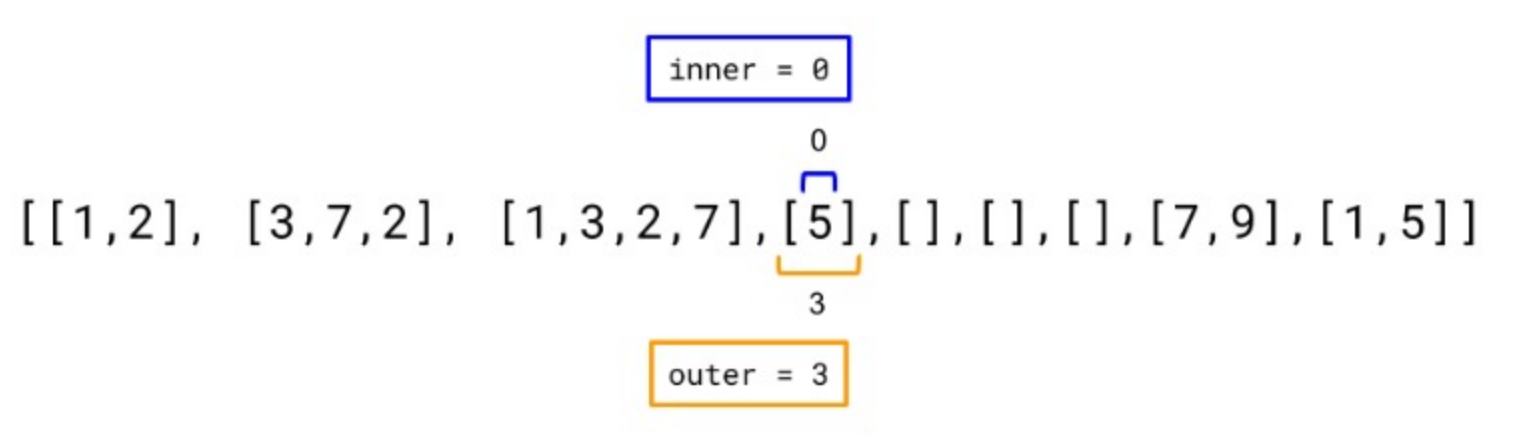
Suppose we are at the following position:



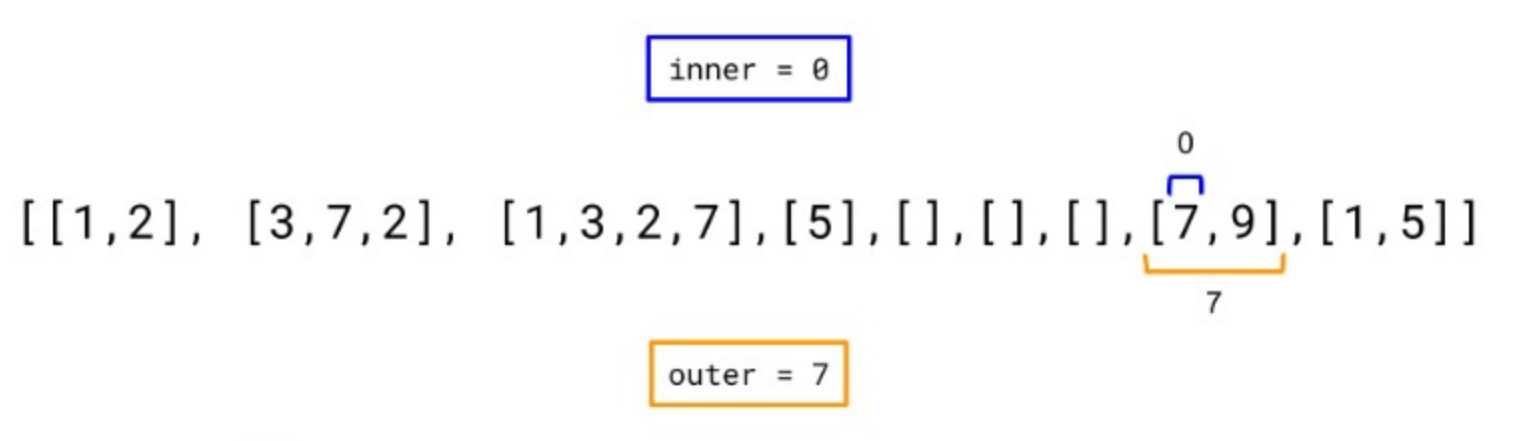
How do we find the next position? Well the current integer has another integer after it, within the same inner vector. Therefore, we can just increment `inner` index by `1`. This gives the next position as shown below.



Now `inner` is at the end of the current inner vector. In order to get to the next integer we'll need to increment `outer` by `1`, and set `inner` to `0` (as `0` is first index of the new vector).



This time, it's a bit trickier, because we need to skip over empty vectors. To do that we repeatedly increment `outer` until we find an inner vector that is not empty (programmatically, this would be an `outer` where `inner = 0` is valid). Once we find one, we stop and set `inner` to `0` (the first integer of the inner vector).



Note that when `outer` becomes equal to the length of the 2D vector, this means there are no more inner vectors and so there are no more numbers left.

Algorithm

In Approach 1, we used $O(N)$ auxiliary space and $O(N+V)$ time in the constructor. In this approach though, we perform the necessary work incrementally during calls to `hasNext()` and `next()`. This means that if the caller stops using the iterator before it's exhausted, we won't have done any unnecessary work.

We'll define an `advanceToNext()` helper method that checks if the current `inner` and `outer` values point to an integer, and if they don't, then it moves them forward until they point to an integer (in the way described above). If `outer == vector.length` becomes true, then the method terminates (because there's no integers left).

In order to ensure no unnecessary work is done, the *constructor* doesn't check whether or not `vector[0][0]` points to an integer. This is because there might be an arbitrary number of empty inner vectors at the start of the input vector; potentially costing up to $O(V)$ operations to skip past.

Both `hasNext()` and `next()` start by calling `advanceToNext()` to ensure that `inner` and `outer` point to an integer, **or** that `outer` is at its "stop" value of `outer = vector.length`.

`next()` returns the integer at `vector[inner][outer]`, and then increments `inner` by `1`, so that the next call to `advanceToNext()` will start searching from after the integer we've just returned.

It is important to note that calling the `hasNext()` method will only cause the pointers to move if they don't point to an integer. Once they point to an integer, repeated calls to `hasNext()` will not move them further. Only `next()` is able to move them *off* a valid integer. This design ensures that the client code calling `hasNext()` multiple times will not have unusual side effects.

Java | Python | Copy

```
11         # when this method terminates.
12     def advance_to_next(self):
13         # While outer is still within the vector, but inner is over the
14         # end of the inner list pointed to by outer, we want to move
15         # forward to the start of the next inner vector.
16         while self.outer < len(self.vector) and self.inner == len(self.vector[self.outer]):
17             self.outer += 1
18             self.inner = 0
19 
20     def next(self) -> int:
21         # Ensure the position pointers are moved such they point to an integer,
22         # or put outer = vector.length.
23         self.advance_to_next()
24         # Return current element and move inner so that is after the current
25         # element.
26         result = self.vector[self.outer][self.inner]
27         self.inner += 1
28         return result
29 
30 
31     def hasNext(self) -> bool:
32         # Ensure the position pointers are moved such they point to an integer,
33         # or put outer = vector.length.
34         self.advance_to_next()
35         # If outer = vector.length then there are no integers left, otherwise
36         # we've stopped at an integer and so there's an integer left.
37         return self.outer < len(self.vector)
```

Complexity Analysis

Let N be the number of integers within the 2D Vector, and V be the number of inner vectors.

- Time complexity.
  - **Constructor:** $O(1)$.

    We're only storing a reference to the input vector—an $O(1)$ operation.
  - **advanceToNext():** $O(\frac{V}{N})$.

    If the iterator is completely exhausted, then all calls to `advanceToNext()` will have performed $O(N+V)$ total operations. (Like in Approach 1, the V comes from the fact that we go through all V inner vectors, and the N comes from the fact we perform one increment for each integer).

    However, because we perform N `advanceToNext()` operations in order to exhaust the iterator, the amortized cost of this operation is just $\frac{O(N+V)}{N} = O(\frac{N}{N} + \frac{V}{N}) = O(\frac{V}{N})$.
  - **next() / hasNext():** $O(\frac{V}{N})$ or $O(1)$.

    The cost of both these methods depends on how they are called. If we just got a value from `next()`, then the next call to either method will involve calling `advanceToNext()`. In this case the time complexity is $O(\frac{V}{N})$.

    However if we call `hasNext()`, then all successive calls to `hasNext()`, or the next call to `next()`, will be $O(1)$. This is because `advanceToNext()` will only perform an $O(1)$ check and immediately return.
- Space complexity : $O(1)$.

  We only use a fixed set of $O(1)$ fields (remember `vector` is a reference, not a copy!). So the space complexity is $O(1)$.

bshaibu ★ 175 April 11, 2020 6:55 AM

Nice article! There's a really good amount of detail added even for explaining why the invalid solution is not the right approach for an iterator. The complexity analysis was pretty useful, especially for the flatten constructor and the 2 pointer's `advanceToNext`.

I think there's a small bug in the Python code for the cheat/flatten list solution on line 6

Read More

2 Share Reply

hope-yao ★ 1 June 15, 2020 9:28 PM

just wondering why [[[]]] is a valid input for initialization in the testing cases...

1 Share Reply

infoparadox9 ★ 92 March 23, 2020 2:25 PM

Very well explained! Thanks a lot @Hai_dee

1 Share Reply

ausafahmed ★ 5 June 27, 2020 6:53 AM Report

Nice article, but wondering why use `inner == vector[outer].length` in the while loop of the second approach?

0 Share Reply

SHOW 1 REPLY

kandulsh ★ 3 June 5, 2020 11:49 PM

Best article!

0 Share Reply

cindy0092 ★ 63 June 3, 2020 4:40 AM

Problem with the 2nd approach implementation is if you don't ensure the content of [i,j] [0,0] is valid in the constructor, then the first next() call will be invalid.

0 Share Reply