Problem 281: Zigzag Iterator

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given two vectors of integers

v1

and

v2

, implement an iterator to return their elements alternately.

Implement the

ZigzagIterator

class:

ZigzagIterator(List<int> v1, List<int> v2)

initializes the object with the two vectors

v1

and

v2

.

boolean hasNext()

returns

true

if the iterator still has elements, and

false

otherwise.

int next()

returns the current element of the iterator and moves the iterator to the next element.

Example 1:

Input:

v1 = [1,2], v2 = [3,4,5,6]

Output:

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

Explanation:

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,3,2,4,5,6].

Example 2:

Input:

v1 = [1], v2 = []

Output:

[1]

Example 3:

Input:

v1 = [], v2 = [1]

Output:

[1]

Constraints:

0 <= v1.length, v2.length <= 1000

1 <= v1.length + v2.length <= 2000

-2

31

<= v1[i], v2[i] <= 2

31

- 1

Follow up:

What if you are given

k

vectors? How well can your code be extended to such cases?

Clarification for the follow-up question:

The "Zigzag" order is not clearly defined and is ambiguous for

k > 2

cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".

Follow-up Example:

Input:

v1 = [1,2,3], v2 = [4,5,6,7], v3 = [8,9]

Output:

[1,4,8,2,5,9,3,6,7]

Code Snippets

C++:

```
class ZigzagIterator {
public:
ZigzagIterator(vector<int>& v1, vector<int>& v2) {

}

int next() {

}

bool hasNext() {

}
};

/**
* Your ZigzagIterator object will be instantiated and called as such:
* ZigzagIterator i(v1, v2);
* while (i.hasNext()) cout << i.next();
```

```
*/
```

Java:

```
public class ZigzagIterator {

public ZigzagIterator(List<Integer> v1, List<Integer> v2) {

}

public int next() {

}

public boolean hasNext() {

}
}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* ZigzagIterator i = new ZigzagIterator(v1, v2);
* while (i.hasNext()) v[f()] = i.next();
*/
```

Python3:

```
class ZigzagIterator:
def __init__(self, v1: List[int], v2: List[int]):


def next(self) -> int:


def hasNext(self) -> bool:


# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())
```

Python:

```
class ZigzagIterator(object):

def __init__(self, v1, v2):
"""
Initialize your data structure here.
:type v1: List[int]
:type v2: List[int]
"""


def next(self):
"""
:rtype: int
"""


def hasNext(self):
"""
:rtype: bool
"""


# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())
```

JavaScript:

```
/**
* @constructor
* @param {Integer[]} v1
* @param {Integer[]} v1
*/
var ZigzagIterator = function ZigzagIterator(v1, v2) {

};


/**
* @this ZigzagIterator
* @returns {boolean}
*/
ZigzagIterator.prototype.hasNext = function hasNext() {
```

```
};

/**
* @this ZigzagIterator
* @returns {integer}
*/
ZigzagIterator.prototype.next = function next() {

};

/**
* Your ZigzagIterator will be called like this:
* var i = new ZigzagIterator(v1, v2), a = [];
* while (i.hasNext()) a.push(i.next());
*/
```

TypeScript:

```
class ZigzagIterator {
constructor(v1: number[], v2: number[]) {

}

next(): number {

}

hasNext(): boolean {

}
}

/**
* Your ZigzagIterator will be instantiated and called as such:
* var i = new ZigzagIterator(v1, v2), a = [];
* while (i.hasNext()) a.push(i.next());
*/
```

C#:

```
public class ZigzagIterator {

public ZigzagIterator(IList<int> v1, IList<int> v2) {

}

public bool HasNext() {

}

public int Next() {

}
}

/**
* Your ZigzagIterator will be called like this:
* ZigzagIterator i = new ZigzagIterator(v1, v2);
* while (i.HasNext()) v[f()] = i.Next();
*/
```

C:

```
struct ZigzagIterator {

};

struct ZigzagIterator *zigzagIteratorCreate(int* v1, int v1Size, int* v2, int
v2Size) {

}

bool zigzagIteratorHasNext(struct ZigzagIterator *iter) {

}

int zigzagIteratorNext(struct ZigzagIterator *iter) {

}

/** Deallocates memory previously allocated for the iterator */
void zigzagIteratorFree(struct ZigzagIterator *iter) {
```

```
}

/**
* Your ZigzagIterator will be called like this:
* struct ZigzagIterator *i = zigzagIteratorCreate(v1, v1Size, v2, v2Size);
* while (zigzagIteratorHasNext(i)) printf("%d\n", zigzagIteratorNext(i));
* zigzagIteratorFree(i);
*/
```

Go:

```
type ZigzagIterator struct {

}

func Constructor(v1, v2 []int) *ZigzagIterator {

}

func (this *ZigzagIterator) next() int {

}

func (this *ZigzagIterator) hasNext() bool {

}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* obj := Constructor(param_1, param_2);
* for obj.hasNext() {
* ans = append(ans, obj.next())
* }
*/
```

Kotlin:

```
class ZigzagIterator {
constructor(v1: IntArray, v2: IntArray) {

}
```

```
fun next(): Int {

}

fun hasNext(): Boolean {

}
}

// Your ZigzagIterator object will be instantiated and called as such:
// var i = ZigzagIterator(v1, v2)
// var ret = ArrayList<Int>()
// while(i.hasNext()){
// ret.add(i.next())
// }
```

Swift:

```
class ZigzagIterator {
init(_ v1: [Int], _ v2: [Int]) {

}

func next() -> Int {

}

func hasNext() -> Bool {

}
}

// Your ZigzagIterator object will be instantiated and called as such:
// var i = ZigzagIterator(v1, v2)
// var ret = [Int]()
// while i.hasNext() {
// ret.append(i.next())
// }
```

Rust:

```
struct ZigzagIterator {

}

/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl ZigzagIterator {
/** initialize your data structure here. */

fn new(v1: Vec<i32>, v2: Vec<i32>) -> Self {

}

fn next(&self) -> i32 {

}

fn has_next(&self) -> bool {

}
}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* let obj = ZigzagIterator::new(v1, v2);
* let ret_1: i32 = obj.next();
* let ret_2: bool = obj.has_next();
*/
```

Ruby:

```
class ZigzagIterator
# @param {Integer[]} v1
# @param {Integer[]} v2
def initialize(v1, v2)

end

# @return {Boolean}
def has_next
```

```
end

# @return {Integer}
def next

end
end

# Your ZigzagIterator will be called like this:
# i, v = ZigzagIterator.new(v1, v2), []
# while i.has_next()
# v << i.next
# end
```

PHP:

```
class ZigzagIterator {
/**
* Initialize your data structure here.
* @param Integer[] $v1
* @param Integer[] $v2
*/
function __construct($v1, $v2) {

}

/**
* @return Integer
*/
function next() {

}

/**
* @return Boolean
*/
function hasNext() {

}
}

/**
```

```
* Your ZigzagIterator object will be instantiated and called as such:
* $obj = ZigzagIterator($v1, $v2);
* while ($obj->hasNext()) {
* array_push($ans, $obj->next())
* }
*/
```

Scala:

```
class ZigzagIterator(_v1: Array[Int], _v2: Array[Int]) {
/** initialize your data structure here. */

def next(): Int = {

}

def hasNext(): Boolean = {

}
}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* var obj = new ZigzagIterator(v1, v2)
* while (obj.hasNext()) {
* ans += obj.next()
* }
*/
```

Solutions

C++ Solution:

```
/*
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
```

```
* Space Complexity: O(1) to O(n) depending on approach
*/

class ZigzagIterator {
public:
ZigzagIterator(vector<int>& v1, vector<int>& v2) {

}

int next() {

}

bool hasNext() {

}
};

/**
* Your ZigzagIterator object will be instantiated and called as such:
* ZigzagIterator i(v1, v2);
* while (i.hasNext()) cout << i.next();
*/
```

Java Solution:

```
/**
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class ZigzagIterator {

public ZigzagIterator(List<Integer> v1, List<Integer> v2) {

}
```

```
public int next() {

}

public boolean hasNext() {

}
}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* ZigzagIterator i = new ZigzagIterator(v1, v2);
* while (i.hasNext()) v[f()] = i.next();
*/
```

Python3 Solution:

```
"""
Problem: Zigzag Iterator
Difficulty: Medium
Tags: array, queue

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class ZigzagIterator:
def __init__(self, v1: List[int], v2: List[int]):


def next(self) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class ZigzagIterator(object):

def __init__(self, v1, v2):
```

```
"""
Initialize your data structure here.
:type v1: List[int]
:type v2: List[int]
"""


def next(self):
"""
:rtype: int
"""


def hasNext(self):
"""
:rtype: bool
"""


# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())
```

JavaScript Solution:

```
/**
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @constructor
* @param {Integer[]} v1
* @param {Integer[]} v1
*/
var ZigzagIterator = function ZigzagIterator(v1, v2) {
```

```
};


/**
* @this ZigzagIterator
* @returns {boolean}
*/
ZigzagIterator.prototype.hasNext = function hasNext() {

};

/**
* @this ZigzagIterator
* @returns {integer}
*/
ZigzagIterator.prototype.next = function next() {

};

/**
* Your ZigzagIterator will be called like this:
* var i = new ZigzagIterator(v1, v2), a = [];
* while (i.hasNext()) a.push(i.next());
*/
```

TypeScript Solution:

```
/**
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class ZigzagIterator {
constructor(v1: number[], v2: number[]) {
```

```
}

next(): number {

}

hasNext(): boolean {

}
}

/**
* Your ZigzagIterator will be instantiated and called as such:
* var i = new ZigzagIterator(v1, v2), a = [];
* while (i.hasNext()) a.push(i.next());
*/
```

C# Solution:

```
/*
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class ZigzagIterator {

public ZigzagIterator(IList<int> v1, IList<int> v2) {

}

public bool HasNext() {

}

public int Next() {
```

```
}
}

/**
* Your ZigzagIterator will be called like this:
* ZigzagIterator i = new ZigzagIterator(v1, v2);
* while (i.HasNext()) v[f()] = i.Next();
*/
```

C Solution:

```
/*
* Problem: Zigzag Iterator
* Difficulty: Medium
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

struct ZigzagIterator {

};

struct ZigzagIterator *zigzagIteratorCreate(int* v1, int v1Size, int* v2, int
v2Size) {

}

bool zigzagIteratorHasNext(struct ZigzagIterator *iter) {

}

int zigzagIteratorNext(struct ZigzagIterator *iter) {

}

/** Deallocates memory previously allocated for the iterator */
void zigzagIteratorFree(struct ZigzagIterator *iter) {
```

```
}

/**
* Your ZigzagIterator will be called like this:
* struct ZigzagIterator *i = zigzagIteratorCreate(v1, v1Size, v2, v2Size);
* while (zigzagIteratorHasNext(i)) printf("%d\n", zigzagIteratorNext(i));
* zigzagIteratorFree(i);
*/
```

Go Solution:

```
// Problem: Zigzag Iterator
// Difficulty: Medium
// Tags: array, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type ZigzagIterator struct {

}

func Constructor(v1, v2 []int) *ZigzagIterator {

}

func (this *ZigzagIterator) next() int {

}

func (this *ZigzagIterator) hasNext() bool {

}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* obj := Constructor(param_1, param_2);
* for obj.hasNext() {
* ans = append(ans, obj.next())
* }
```

```
*/
```

Kotlin Solution:

```
class ZigzagIterator {
constructor(v1: IntArray, v2: IntArray) {

}

fun next(): Int {

}

fun hasNext(): Boolean {

}
}

// Your ZigzagIterator object will be instantiated and called as such:
// var i = ZigzagIterator(v1, v2)
// var ret = ArrayList<Int>()
// while(i.hasNext()){
// ret.add(i.next())
// }
```

Swift Solution:

```
class ZigzagIterator {
init(_ v1: [Int], _ v2: [Int]) {

}

func next() -> Int {

}

func hasNext() -> Bool {

}
}
```

```
// Your ZigzagIterator object will be instantiated and called as such:
// var i = ZigzagIterator(v1, v2)
// var ret = [Int]()
// while i.hasNext() {
// ret.append(i.next())
// }
```

Rust Solution:

```
// Problem: Zigzag Iterator
// Difficulty: Medium
// Tags: array, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct ZigzagIterator {

}

/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl ZigzagIterator {
/** initialize your data structure here. */

fn new(v1: Vec<i32>, v2: Vec<i32>) -> Self {

}

fn next(&self) -> i32 {

}

fn has_next(&self) -> bool {

}
}
```

```
/**
* Your ZigzagIterator object will be instantiated and called as such:
* let obj = ZigzagIterator::new(v1, v2);
* let ret_1: i32 = obj.next();
* let ret_2: bool = obj.has_next();
*/
```

Ruby Solution:

```
class ZigzagIterator
# @param {Integer[]} v1
# @param {Integer[]} v2
def initialize(v1, v2)

end

# @return {Boolean}
def has_next

end

# @return {Integer}
def next

end
end

# Your ZigzagIterator will be called like this:
# i, v = ZigzagIterator.new(v1, v2), []
# while i.has_next()
# v << i.next
# end
```

PHP Solution:

```
class ZigzagIterator {
/**
* Initialize your data structure here.
* @param Integer[] $v1
* @param Integer[] $v2
*/
```

```
function __construct($v1, $v2) {

}

/**
* @return Integer
*/
function next() {

}

/**
* @return Boolean
*/
function hasNext() {

}
}

/**
* Your ZigzagIterator object will be instantiated and called as such:
* $obj = ZigzagIterator($v1, $v2);
* while ($obj->hasNext()) {
* array_push($ans, $obj->next())
* }
*/
```

Scala Solution:

```
class ZigzagIterator(_v1: Array[Int], _v2: Array[Int]) {
/** initialize your data structure here. */

def next(): Int = {

}

def hasNext(): Boolean = {

}
}
```

```
/**
* Your ZigzagIterator object will be instantiated and called as such:
* var obj = new ZigzagIterator(v1, v2)
* while (obj.hasNext()) {
* ans += obj.next()
* }
*/
```