Approach 2: Stack

**Intuition**

The downside of Approach 1 is that 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 at a time, keeping track of where we're currently up to in nestedList.

A better way is to do an iterative depth-first search, based on the following tree traversal algorithm:

define function iterativeDepthFirstSearch(nestedList):

result = []

stack = a new Stack

push all items in nestedList onto stack, in reverse order

while stack is not empty:

nestedInteger = pop top of stack

if nestedInteger.isInteger():

append nestedInteger.getInteger() to result

else:

list = nestedInteger.getList()

push all items in list onto stack, in reverse order

return result

While we could use this algorithm in the constructor like before, a better way would be to store stack on the iterator object and progress the algorithm on each call to next() to get the next integer out.

Notice that if the top of the stack is an integer, then we've already found the next integer. Otherwise, if it's a list, then the else is adding the list contents to stack. On the next loop iteration, the same will happen. We could write an algorithm to get the next integer as follows.

stack = a new Stack

push all items in nestedList onto stack, in reverse order

define function getNextInteger():

while stack is not empty:

nestedInteger = pop top off stack

if nestedInteger.isInteger():

RETURN nestedInteger.getInteger()

else:

list = nestedInteger.getList()

push all items in list onto stack, in reverse order

Notice that the stack is shared between calls. This means that getNextInteger() will find an integer and return it, while still preserving the state of the stack. We can then call getNextInteger() again to get the next integer, and so forth.

To simplify the code a bit, we can change our loop condition so that it checks if the top of the stack is still a list. The loop body should push the contents of the list onto the stack (in reverse). Eventually, there will be an integer on the top of the stack, OR the stack will be empty. Being able to get the next integer to the top of the stack allows the next() and hasNext() methods to access it.

stack = a new Stack

push all items in nestedList onto stack, in reverse order

define function makeStackTopAnInteger():

while stack is not empty AND the nestedInteger at top of stack is a list:

nestedInteger = pop top off stack

list = nestedInteger.getList()

push all items in list onto stack, in reverse order

**Algorithm**

Let's define a private method called makeStackTopAnInteger() that contains the algorithm to make the stack top an integer (as described above). The makeStackTopAnInteger() method never *removes* integers.

The next() and hasNext() methods should call makeStackTopAnInteger() before doing anything else. This means that they can then *assume* that either the stack top is an integer, *or* the stack is empty. Then, their definitions are as follows:

* **hasNext():** Returns true if the stack still contains items, false if not.
* **next():** If the stack still contains items, then it is guaranteed the top is an integer. This integer is popped and returned. If the stack is empty, then the behavior is language-dependent. For example, in Java, a NoSuchElementException should be throw.

**Complexity Analysis**

Let N*N* be the total number of *integers* within the nested list, L*L* be the total number of *lists* within the nested list, and D*D* be the maximum nesting depth (maximum number of lists inside each other).

* Time complexity.
  + **Constructor:** O(N + L)*O*(*N*+*L*).

The worst-case occurs when the initial input nestedList consists entirely of integers and empty lists (everything is in the top-level). In this case, every item is reversed and stored, giving a total time complexity of O(N + L)*O*(*N*+*L*).

* + **makeStackTopAnInteger():** O(\dfrac{L}{N})*O*(*NL*​) or O(1)*O*(1).

If the top of the stack is an integer, then this function does nothing; taking O(1)*O*(1) time.

Otherwise, it needs to process the stack until an integer is on top. The best way of analyzing the time complexity is to look at the total cost across all calls to makeStackTopAnInteger() and then divide by the number of calls made. Once the iterator is exhausted makeStackTopAnInteger() must have seen every integer at least once, costing O(N)*O*(*N*) time. Additionally, it has seen every list (except the first) on the stack at least once also, so this costs O(L)*O*(*L*) time. Adding these together, we get O(N + L)*O*(*N*+*L*) time.

The amortized time of a single makeStackTopAnInteger is the total cost, O(N + L)*O*(*N*+*L*), divided by the number of times it's called. In order to get all integers, we need to have called it N*N* times. This gives us an amortized time complexity of \dfrac{O(N + L)}{N} = O(\dfrac{N}{N} + \dfrac{L}{N}) = O(\dfrac{L}{N})*NO*(*N*+*L*)​=*O*(*NN*​+*NL*​)=*O*(*NL*​).

* + **next():** O(\dfrac{L}{N})*O*(*NL*​) or O(1)*O*(1).

All of this method is O(1)*O*(1), except for possibly the call to makeStackTopAnInteger(), giving us a time complexity the same as makeStackTopAnInteger().

* + **hasNext():** O(\dfrac{L}{N})*O*(*NL*​) or O(1)*O*(1).

All of this method is O(1)*O*(1), except for possibly the call to makeStackTopAnInteger(), giving us a time complexity the same as makeStackTopAnInteger().

* Space complexity : O(N + L)*O*(*N*+*L*).

In the worst case, where the top list contains N*N* integers, or L*L* empty lists, it will cost O(N + L)*O*(*N*+*L*) space. Other expensive cases occur when the nesting is very deep. However, it's useful to remember that D ≤ L*D*≤*L* (because each layer of nesting requires another list), and so we don't need to take this into account.

Approach 3: Two Stacks

**Intuition**

Reversing the lists to put them onto the stack can be an expensive operation, and it turns out it isn't necessary.

Instead of pushing every item of a sub-list onto the stack, we can instead associate an index pointer with each sub-list, that keeps track of how far along that sub-list we are. Adding a new sub-list to the stack now becomes an O(1)*O*(1) operation instead of a O(length of sublist)*O*(*lengthofsublist*) one.

Here is an animation showing this approach.

Graphical user interface, application, email

Description automatically generated

Step 1

Application

Description automatically generated with medium confidence

Step 2

A picture containing text

Description automatically generated

Step 3

A picture containing application

Description automatically generated

Step 4

A picture containing graphical user interface

Description automatically generated

Step 5

Text

Description automatically generated with low confidence

Step 6

A picture containing chart

Description automatically generated

Step 7

The *total* time complexity across all method calls for using up the entire iterator remains the same, *but* work is only done when it's necessary, thus improving performance when we only use part of the iterator. This is a desirable property for an iterator.

**Algorithm**

This approach can be implemented as either one stack of pairs/ tuples, or two stacks with one for NestedIntegers and the other for indexes. The best decision for this is language-dependent. I tried both for the Java and found that attempting to put Pair objects onto a single stack doesn't work well because updating an index count requires popping and then reconstructing the entire Pair due to immutability (alternatives such as using length-2 Listss as pairs are possible, but I don't think ideal). Using two stacks is cleaner.

**Complexity Analysis**

Let N*N* be the total number of *integers* within the nested list, L*L* be the total number of *lists* within the nested list, and D*D* be the maximum nesting depth (maximum number of lists inside each other).

* Time complexity:
  + **Constructor:** O(1)*O*(1).

Pushing a list onto a stack is *by reference* in all the programming languages we're using here. This means that instead of creating a new list, some information about how to get to the existing list is put onto the stack. The list is not traversed, as it doesn't need reversing this time, and we're not pushing the items on one-by-one. This is, therefore, an O(1)*O*(1) operation.

* + **makeStackTopAnInteger() / next() / hasNext():** O(\dfrac{L}{N})*O*(*NL*​) or O(1)*O*(1).

Same as Approach 2.

* Space complexity : O(D)*O*(*D*).

At any given time, the stack contains only *one* nestedList reference for each level. This is unlike the previous approach, wherein the worst case we need to put almost all elements onto the stack.

Because there's one reference on the stack at each level, the worst case is when we're looking at the deepest leveled list, giving a space complexity is O(D)*O*(*D*).