Interview Cake

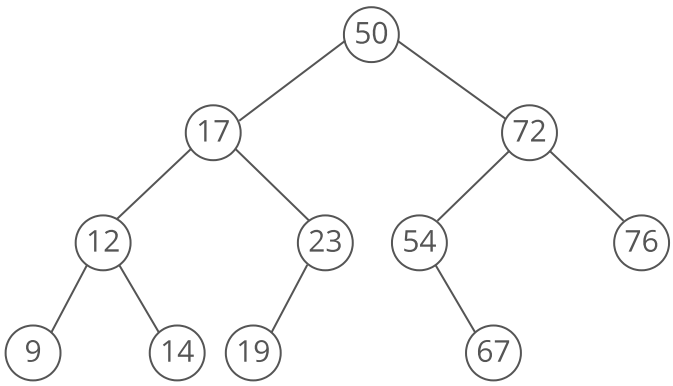
Write a function to check that a binary tree is a valid binary search tree.

A **binary search tree** is a binary tree (/concept/tree) where the nodes are ordered in a specific way. For every node:

- The nodes to the left are *smaller* than the current node.
- The nodes to the right are *larger* than the current node.

	Balanced	Unbalanced (Worst Case)
space	$O(n)$	$O(n)$
insert	$O(lg(n))$	$O(n)$
lookup	$O(lg(n))$	$O(n)$
delete	$O(lg(n))$	$O(n)$

> Checking if a binary tree is a binary *search* tree is a favorite question from interviews (/question/bst-checker).



Strengths:

- **Good performance across the board.** Assuming they're balanced (/concept/tree#balanced), binary search trees are good at lots of operations, even if they're not constant time for anything.

    - Compared to a sorted array (/concept/array), lookups take the same amount of time ($O(lg(n))$), but inserts and deletes are faster ($O(lg(n))$ for BSTs, $O(n)$ for arrays).
    - Compared to dictionaries (/concept/hash-map), BSTs have better *worst case* performance—$O(lg(n))$ instead of $O(n)$. *But*, on average dictionaries perform better than BSTs (meaning $O(1)$ time complexity).

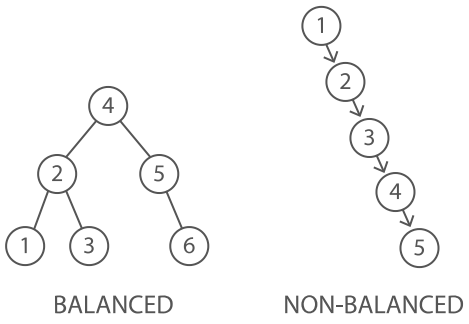
- **BSTs are sorted.** Taking a binary search tree and pulling out all of the elements in sorted order can be done in $O(n)$ using an in-order traversal. Finding the element *closest* to a value can be done in $O(lg(n))$ (again, if the BST is balanced!).

Weaknesses:

- **Poor performance if unbalanced.** Some types of binary search trees balance automatically, but not all. If a BST is not balanced, then operations become $O(n)$.
- **No $O(1)$ operations.** BSTs aren't the *fastest* for anything. On average, a list (/concept/array) or a dictionary (/concept/hash-map) will be faster.

Balanced Binary Search Trees

Two binary search trees can store the same values in different ways:



Some trees (like AVL trees or Red-Black trees) rearrange nodes as they're inserted to ensure the tree is always balanced. With these, the worst case complexity for searching, inserting, or deleting is *always* $O(lg(n))$, not $O(n)$.

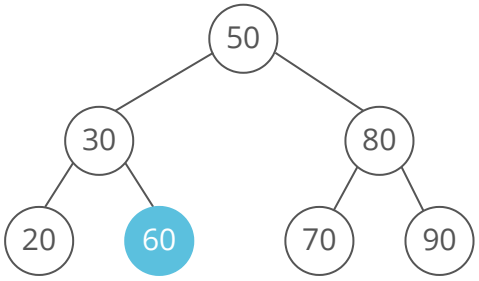
Here's a sample binary tree node class:

Python 3.6

```
class BinaryTreeNode(object):

    def __init__(self, value):
        self.value = value
        self.left  = None
        self.right = None

    def insert_left(self, value):
        self.left = BinaryTreeNode(value)
        return self.left

    def insert_right(self, value):
        self.right = BinaryTreeNode(value)
        return self.right
```

Gotchas

Consider this example:



Notice that when you check the blue node against its parent, it seems correct. However, it's greater than the root, so it should be in the root's right subtree. So we see that **checking a node against its parent isn't sufficient to prove that it's in the correct spot**.

We can do this in $O(n)$ time and $O(n)$ additional space, where n is the number of nodes in our tree. Our additional space is $O(\lg n)$ if our tree is balanced.

Breakdown

One way to break the problem down is to come up with a way to confirm that a single node is in a valid place relative to its ancestors. Then if every node passes this test, our whole tree is a valid BST.

What makes a given node "correct" relative to its ancestors in a BST? Two things:

- if a node is in the ancestor's *left* subtree, then it must be *less* than the ancestor, and
- if a node is in the ancestor's *right* subtree, then it must be *greater* than the ancestor.

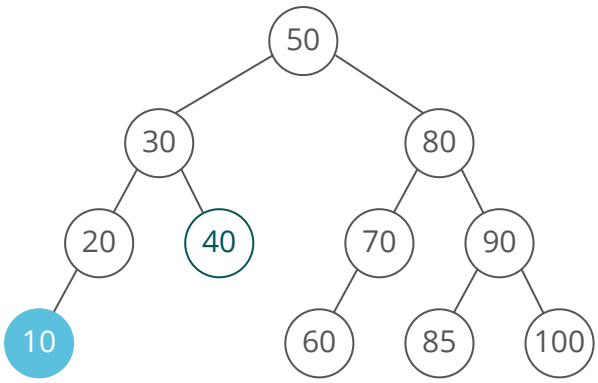
So we could do a walk through our binary tree, **keeping track of the ancestors for each node and whether the node should be greater than or less than each of them**. If each of these greater-than or less-than relationships holds true for each node, our BST is valid.

The simplest ways to traverse the tree are depth-first and breadth-first. They have the same time cost (they each visit each node once). Depth-first traversal of a tree uses memory proportional to the depth of the tree, while breadth-first traversal uses memory proportional to the breadth of the tree (how many nodes there are on the "level" that has the most nodes).

Because the tree's breadth can as much as double each time it gets one level deeper, **depth-first traversal is likely to be more space-efficient than breadth-first traversal**, though they are strictly both $O(n)$ additional space in the worst case. The space savings are obvious if we know our binary tree is balanced—its depth will be $O(\lg n)$ and its breadth will be $O(n)$.

But we're not just storing the nodes themselves in memory, we're also storing the value from each ancestor and whether it should be less than or greater than the given node. Each node has $O(n)$ ancestors ($O(\lg n)$ for a balanced binary tree), so that gives us $O(n^2)$ additional memory cost ($O(n \lg n)$ for a balanced binary tree). We can do better.

Let's look at the inequalities we'd need to store for a given node:



Notice that we would end up testing that the blue node is <20, <30, and <50. Of course, <30 and <50 are implied by <20. So instead of storing each ancestor, we can just keep track of a `lower_bound` and `upper_bound` that our node's value must fit inside.

Solution

We do a depth-first walk through the tree, testing each node for validity as we go. If a node appears in the *left* subtree of an ancestor, it must be less than that ancestor. If a node appears in the *right* subtree of an ancestor, it must be greater than that ancestor.

Instead of keeping track of every ancestor to check these inequalities, we just check the largest number it must be greater than (its `lower_bound`) and the smallest number it must be less than (its `upper_bound`).

Python 3.6

```
def is_binary_search_tree(root):

    # Start at the root, with an arbitrarily low lower bound
    # and an arbitrarily high upper bound
    node_and_bounds_stack = [(root, -float('inf'), float('inf'))]

    # Depth-first traversal
    while len(node_and_bounds_stack):
        node, lower_bound, upper_bound = node_and_bounds_stack.pop()

        # If this node is invalid, we return false right away
        if (node.value <= lower_bound) or (node.value >= upper_bound):
            return False

        if node.left:
            # This node must be less than the current node
            node_and_bounds_stack.append((node.left, lower_bound, node.value))
        if node.right:
            # This node must be greater than the current node
            node_and_bounds_stack.append((node.right, node.value, upper_bound))

    # If none of the nodes were invalid, return true
    # (at this point we have checked all nodes)
    return True
```

Instead of allocating a stack⤵ ourselves, we could write a **recursive function** that uses the **call stack**.⤵

Overview

The **call stack** is what a program uses to keep track of function calls. The call stack is made up of **stack frames**—one for each function call.

For instance, say we called a function that rolled two dice and printed the sum.

Python 3.6

```
def roll_die():
    return random.randint(1, 6)

def roll_two_and_sum():
    total = 0
    total += roll_die()
    total += roll_die()
    print(total)

roll_two_and_sum()
```

First, our program calls `roll_two_and_sum()`. It goes on the call stack:

```
roll_two_and_sum()
```

That function calls `roll_die()`, which gets pushed on to the top of the call stack:

```
roll_die()
```

```
roll_two_and_sum()
```

Inside of `roll_die()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

When `random.randint()` finishes, we return back to `roll_die()` by removing ("popping") `random.randint()`'s stack frame.

```
roll_die()
```

```
roll_two_and_sum()
```

Same thing when `roll_die()` returns:

```
roll_two_and_sum()
```

We're not done yet! `roll_two_and_sum()` calls `roll_die()` *again*:

```
roll_die()
```

```
roll_two_and_sum()
```

Which calls `random.randint()` again:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

`random.randint()` returns, then `roll_die()` returns, putting us back in `roll_two_and_sum()`:

```
roll_two_and_sum()
```

Which calls `print()()`:

```
print()()
```

```
roll_two_and_sum()
```

What's stored in a stack frame?

What *actually* goes in a function's stack frame?

A stack frame usually stores:

- Local variables
- Arguments passed into the function
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each function call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

Python 3.6

```
def product_1_to_n(n):
    return 1 if n <= 1 else n * product_1_to_n(n - 1)
```

What would the call stack look like when `n = 10`?

First, `product_1_to_n()` gets called with `n = 10`:

```
product_1_to_n()    n = 10
```

This calls `product_1_to_n()` with `n = 9`.

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Which calls `product_1_to_n()` with `n = 8`.

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

And so on until we get to `n = 1`.

```
product_1_to_n()    n = 1
```

```
product_1_to_n()    n = 2
```

```
product_1_to_n()    n = 3
```

```
product_1_to_n()    n = 4
```

```
product_1_to_n()    n = 5
```

```
product_1_to_n()    n = 6
```

```
product_1_to_n()    n = 7
```

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our function itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

Python 3.6

```
def product_1_to_n(n):
    # We assume n >= 1
    result = 1
    for num in range(1, n + 1):
        result *= num

    return result
```

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product_1_to_n()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product_1_to_n()    n = 10, result = 2, num = 2
```

```
product_1_to_n()    n = 10, result = 6, num = 3
```

```
product_1_to_n()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

> What happens if we run out of space? It's a **stack overflow**! In Python 3.6, you'll get a `RecursionError`.

> If the *very last* thing a function does is call another function, then its stack frame might not be needed any more. The function *could* free up its stack frame before doing its final call, saving space.
>
> This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.
>
> In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

This would work, but it would be **vulnerable to stack overflow**. However, the code does end up quite a bit cleaner:

Python 3.6

```
def is_binary_search_tree(root,
                          lower_bound=-float('inf'),
                          upper_bound=float('inf')):
    if not root:
        return True

    if (root.value >= upper_bound or root.value <= lower_bound):
        return False

    return (is_binary_search_tree(root.left, lower_bound, root.value)
            and is_binary_search_tree(root.right, root.value, upper_bound))
```

Checking if an in-order traversal of the tree is sorted is a great answer too, especially if you're able to implement it without storing a full list of nodes.

Complexity

$O(n)$ time and $O(n)$ space.

The time cost is easy: for valid binary search trees, we'll have to check *all* n *nodes*.

Space is a little more complicated. Because we're doing a depth first search, `node_and_bounds_stack` will hold at most d nodes where d is the depth of the tree (the number of levels in the tree from the root node down to the lowest node). So we *could* say our space cost is $O(d)$.

But we can also relate d to n. In a balanced tree, d is $\log_2 n$ (/concept/binary-tree#property2). And the *more unbalanced* the tree gets, the closer d gets to n.

In the worst case, the tree is a straight line of right children from the root where every node in that line also has a left child. The traversal will walk down the line of right children, adding a new left child to the stack at each step. When the traversal hits the rightmost node, the stack will hold *half* of the n total nodes in the tree. Half n is $O(n)$, so our worst case space cost is $O(n)$.

Bonus

What if the input tree has duplicate values?

What if `-float('inf')` or `float('inf')` appear in the input tree?

What We Learned

We could think of this as a **greedy**.

A **greedy** algorithm builds up a solution by choosing the option that looks the best at every step.

Say you're a cashier and need to give someone 67 cents (US) using as few coins as possible. How would you do it?

Whenever picking which coin to use, you'd take the highest-value coin you could. A quarter, another quarter, then a dime, a nickel, and finally two pennies. That's a *greedy* algorithm, because you're always *greedily* choosing the coin that covers the biggest portion of the remaining amount.

Some other places where a greedy algorithm gets you the best solution:

- Trying to fit as many overlapping meetings as possible in a conference room? At each step, schedule the meeting that *ends* earliest.
- Looking for a minimum spanning tree in a graph (/concept/graph)? At each step, greedily pick the cheapest edge that reaches a new vertex.

Careful: sometimes a greedy algorithm *doesn't* give you an optimal solution:

- When filling a duffel bag with cakes of different weights and values (/question/cake-thief), choosing the cake with the highest value per pound doesn't always produce the best haul.
- To find the cheapest route visiting a set of cities, choosing to visit the cheapest city you haven't been to yet doesn't produce the cheapest overall itinerary.

Validating that a greedy strategy always gets the best answer is tricky. Either prove that the answer produced by the greedy algorithm is as good as an optimal answer, or run through a rigorous set of test cases to convince your interviewer (and yourself) that its correct.

approach. We start off by trying to solve the problem in just one walk through the tree. So we ask ourselves what values we need to track in order to do that. Which leads us to our stack that tracks upper and lower bounds.

We could also think of this as a sort of "**divide and conquer**" approach. The idea in general behind divide and conquer is to break the problem down into two or more subproblems, solve them, and then use that solution to solve the original problem.

In *this* case, we're dividing the problem into subproblems by saying, "This tree is a valid binary search tree if the left subtree is valid and the right subtree is valid." This is more apparent in the recursive formulation of the answer above.

Of course, it's just fine that our approach *could be* thought of as greedy or *could be* thought of as divide and conquer. It can be both. The point here isn't to create strict categorizations so we can debate whether or not something "counts" as divide and conquer.

Instead, the point is to recognize the underlying *patterns* behind algorithms, so we can get better at thinking through problems.

Sometimes we'll have to kinda smoosh together two or more different patterns to get our answer.