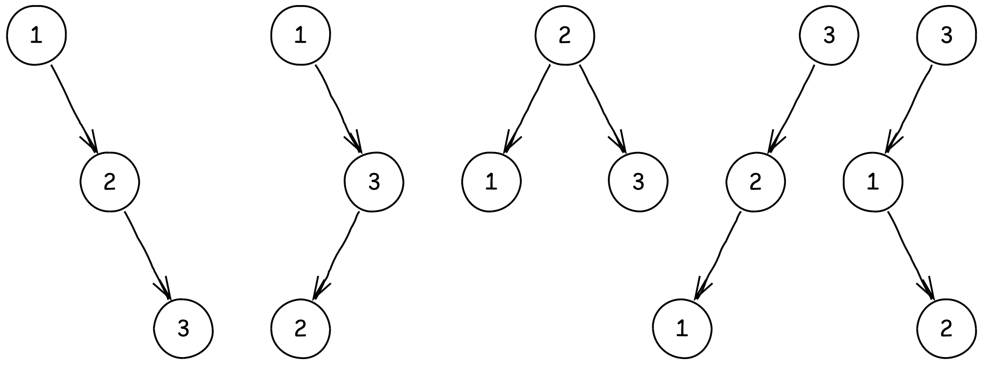
7 The number of binary search trees

Given a list of distinct numbers, count how many distinct binary search trees can be formed to store the numbers.

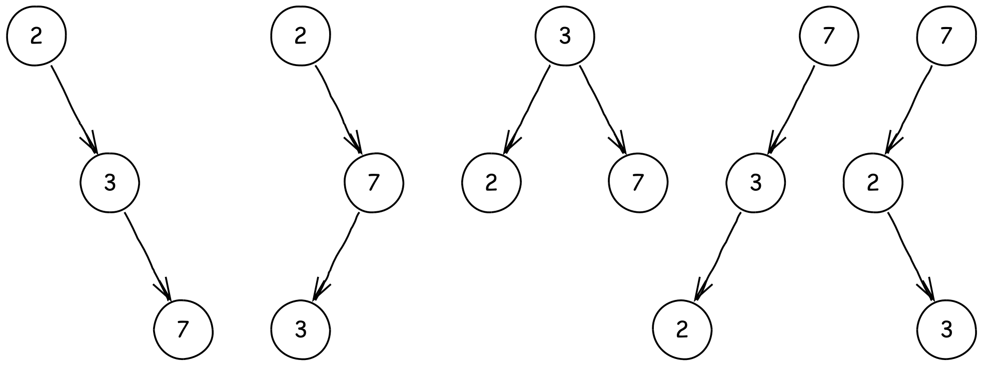
Example: Given the numbers `[1, 2, 3]`, we can form 5 distinct binary search trees:



Solution 1: dynamic programming, top-down, $O(n^2)$ time

An important observation we can make is that the number of unique trees does not depend on the values of the numbers in the list, just on their count.

For example, the number of unique binary search trees (BSTs) that can be formed to store `[2, 3, 7]` is 5 as well:



This is easy to prove: suppose that we have an algorithm to generate all the unique binary trees for a list of distinct numbers `[a1, a2, ..., an]`. For any other list of distinct numbers `[b1, b2, ..., bn]` with the same length, we can use the same trees and just relabel the nodes: `a1` becomes `b1`, `a2` becomes `b2` etc. We can relabel the nodes uniquely since we know that there are no duplicates in either list. Thus the nuber of unique BSTs depends only on the number of elements to be stored.

A second observation that we can make is that we can split the problem into smaller sub-problems if we fix a small piece of the problem that is otherwise variable. Note that this is the same approach we used for the change-making problem (choosing the first coin) and the sentence-splitting problem (choosing the first word).

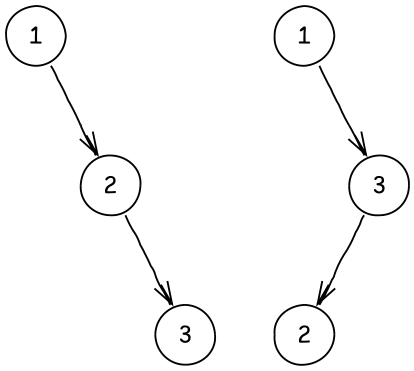
In case of the BSTs, the natural variable to fix is the item to be placed at the root of the tree.

Consider again the example of items `[1, 2, 3]`. It is not clear which one should be placed at the root—in fact, any of the numbers is a valid choice. So we split the problem into 3 subproblems:

- a subproblem where 1 is placed at the root;
- a subproblem where 2 is placed at the root;
- a subproblem where 3 is placed at the root.

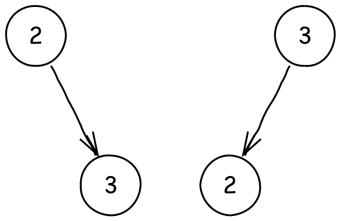
In each case, we need to count the number of ways in which we can place the other nodes to construct the full tree. Let's take the cases one by one.

Consider the case where 1 is chosen as the root of the tree. We need to place the other items, 2 and 3. In the end we would like to form the following trees:

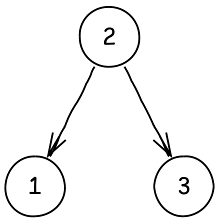


Since both 2 and 3 are greater than the root 1, they must be placed in the right subtree. In other words, the number of unique BSTs with 3 items and having the smallest item at the root is equal to the number of ways we can form unique BSTs from 2 items (and have each BST become the right subtree of the root).

We have just reduced case 1 of our problem to a smaller subproblem. The solution of the subproblem is computed as 2, since there are only 2 possible trees that can be formed with 2 items:

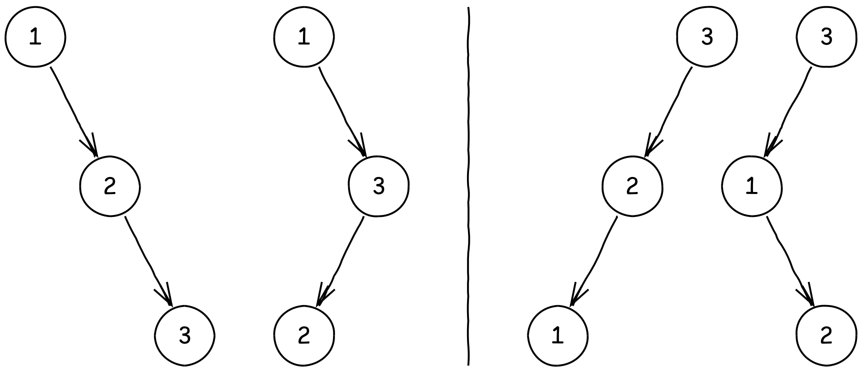


Consider now the case where 2 is chosen as the root of the tree. We need to place the other items, 1 and 3, to form the following trees:



Item 1 must be part of the left subtree of 2, while item 3 must be part of the right subtree. In this case there is only one way to place them. In general, we should count the number of BSTs that can be formed by the items in the left subtree, and the number of BSTs that can be formed by the items in the right subtree. The total number of trees that can be formed is the product of the two numbers (since we can choose any combination to pair them when we construct the full tree).

Finally, consider the case where 3 is chosen as the root of the tree. We need to place items 1 and 2 on the left subtree. This case is symmetrical to the first case (1 fixed at the root), as we can see in the following diagram, so we will not discuss it:



We can now write the implementation of the algorithm. Given that the items are not important for solving the problem, only their count, we can start by defining a helper function that only takes as parameter the number of items:

```python
def count_bsts(items):
    """
    Returns the number of unique binary search trees that
    can be formed using the given items.
    The items must be distinct.
    """
    def helper(num_items):
        ...
    return helper(len(items))
```

The helper function should return 1 if the number of items is 1. Otherwise, it should compute how many trees can be formed by placing at the root one of the items, then return the sum of all these numbers:

```
def count_bsts(items):
    """
    Returns the number of unique binary search trees that
    can be formed using the given items.
    The items must be distinct.
    """
    def helper(num_items):
        # A single item can be placed in a BST in exactly 1 way:
        # at the root.
        # No items form an empty tree, which is also unique.
        if num_items <= 1:
            return 1
        result = 0
        # Consider all the possible choices of having 1 item at the root,
        # and the rest in the left and right subtrees.
        for num_items_left in range(num_items):
            num_bsts_left = helper(num_items_left)
            num_items_right = num_items - 1 - num_items_left
            num_bsts_right = helper(num_items_right)
            result += num_bsts_left * num_bsts_right
        return result
    return helper(len(items))
```

The final piece we need for solving this problem efficiently is to cache the result of the helper function (to avoid exponential complexity due to redundant computations):

```
def count_bsts(items):
    """
    Returns the number of unique binary search trees that
    can be formed using the given items.
    The items must be distinct.
    """
    @lru_cache(maxsize=None)
    def helper(num_items):
        # A single item can be placed in a BST in exactly 1 way:
        # at the root.
        # No items form an empty tree, which is also unique.
        if num_items <= 1:
            return 1
        result = 0
        # Consider all the possible choices of item at the root.
        for num_items_left in range(num_items):
            # Fixing the root, compute how many possible left subtrees
            # we can form.
            num_bsts_left = helper(num_items_left)
            # Now compute how many possible right subtrees
            # we can form.
            num_items_right = num_items - 1 - num_items_left
            num_bsts_right = helper(num_items_right)
            # This is the number of trees having the root fixed to
            # the current item.
            result += num_bsts_left * num_bsts_right
        return result
    return helper(len(items))
```

Since the maximum number of non-cached calls of the function is n (the number of items), and at each call we make $O(n)$ operations, the time complexity of the solution is $O(n^2)$.