96. Unique Binary Search Trees

Sept. 10, 2018 | 66.2K views

Average Rating: 4.82 (116 votes)

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1 ... *n*?

Example:

```
Input: 3
Output: 5
Explanation:
Given n = 3, there are a total of 5 unique BST's:

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Solution

Approach 1: Dynamic Programming

Intuition

The problem can be solved in a dynamic programming way.

Given a sorted sequence `1 ... n`, to construct a Binary Search Tree (BST) out of the sequence, we could enumerate each number `i` in the sequence, and use the number as the root, then, the subsequence `1 ... (i-1)` on its left side would lay on the left branch of the root, and similarly the right subsequence `(i+1) ... n` lay on the right branch of the root. We then can construct the subtree from the subsequence recursively. Through the above approach, we could be assured that the BST that we construct are all unique, since they start from unique roots.

As we can see, the problem can be reduced into problems with smaller sizes, instead of recursively (also repeatedly) solve the subproblems, we can store the solution of subproblems and reuse them later, *i.e.* the dynamic programming way.

Algorithm

The problem is to calculate the number of unique BST. To do so, we can define two functions:

1. $G(n)$: the number of unique BST for a sequence of length `n`.
2. $F(i, n)$: the number of unique BST, where the number `i` is served as the root of BST ($1 \leq i \leq n$).

As we can see,

> $G(n)$ is actually the desired function we need in order to solve the problem.

Later we would see that $G(n)$ can be deducted from $F(i, n)$, which at the end, would recursively refers to $G(n)$.

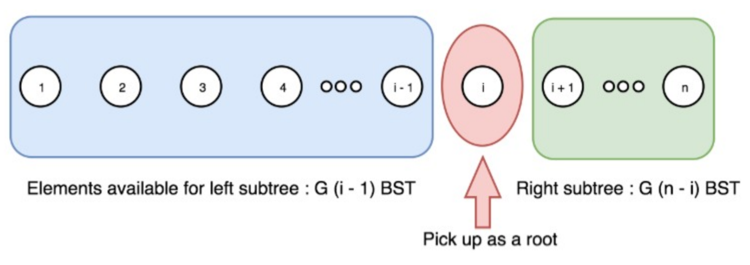
First of all, following the idea in the section of intuition, we can see that the total number of unique BST $G(n)$, is the sum of BST $F(i, n)$ enumerating each number `i` (`1 <= i <= n`) as a root. *i.e.*

$$G(n) = \sum_{i=1}^{n} F(i, n) \qquad (1)$$

Particularly, for the bottom cases, there is only one combination to construct a BST out of a sequence of length 1 (only a root) or nothing (empty tree). *i.e.*

$$G(0) = 1, \qquad G(1) = 1$$

Given a sequence `1 ... n`, we pick a number `i` out of the sequence as the root, then the number of unique BST with the specified root defined as $F(i, n)$, is the **cartesian product** of the number of BST for its left and right subtrees, as illustrated below:



For example, $F(3, 7)$, the number of unique BST tree with the number `3` as its root. To construct an unique BST out of the entire sequence `[1, 2, 3, 4, 5, 6, 7]` with `3` as the root, which is to say, we need to construct a subtree out of its left subsequence `[1, 2]` and another subtree out of the right subsequence `[4, 5, 6, 7]`, and then combine them together (*i.e.* cartesian product). Now the tricky part is that we could consider the number of unique BST out of sequence `[1,2]` as $G(2)$, and the number of of unique BST out of sequence `[4, 5, 6, 7]` as $G(4)$. For $G(n)$, it does not matter the content of the sequence, but the length of the sequence. Therefore, $F(3, 7) = G(2) \cdot G(4)$. To generalise from the example, we could derive the following formula:

$$F(i, n) = G(i-1) \cdot G(n-i) \qquad (2)$$

By combining the formulas (1), (2), we obtain a recursive formula for $G(n)$, *i.e.*

$$G(n) = \sum_{i=1}^{n} G(i-1) \cdot G(n-i) \qquad (3)$$

To calculate the result of function, we start with the lower number, since the value of $G(n)$ depends on the values of $G(0) \ldots G(n-1)$.

With the above explanation and formulas, one can easily implement an algorithm to calculate the $G(n)$. Here are some examples:

Java | Python | Copy

```python
class Solution:
    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        G = [0]*(n+1)
        G[0], G[1] = 1, 1

        for i in range(2, n+1):
            for j in range(1, i+1):
                G[i] += G[j-1] * G[i-j]

        return G[n]
```

Complexity Analysis

- Time complexity : the main computation of the algorithm is done at the statement with `G[i]`. So the time complexity is essentially the number of iterations for the statement, which is $\sum_{i=2}^{n} i = \frac{(2+n)(n-1)}{2}$, to be exact, therefore the time complexity is $O(N^2)$
- Space complexity : The space complexity of the above algorithm is mainly the storage to keep all the intermediate solutions, therefore $O(N)$.

Approach 2: Mathematical Deduction

Intuition

Actually, as it turns out, the sequence of $G(n)$ function results is known as Catalan number C_n. And one of the more convenient forms for calculation is defined as follows:

$$C_0 = 1, \qquad C_{n+1} = \frac{2(2n+1)}{n+2} C_n \qquad (4)$$

We skip the proof here, which one can find following the above reference.

Algorithm

Given the formula (3), it becomes rather easy to calculate G_n which is actually C_n. Here are some examples:

Java | Python | Copy

```python
class Solution(object):
    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        C = 1
        for i in range(0, n):
            C = C * 2*(2*i+1)/(i+2)
        return int(C)
```

Complexity Analysis

- Time complexity : $O(N)$, as one can see, there is one single loop in the algorithm.
- Space complexity : $O(1)$, we use only one variable to store all the intermediate results and the final one.

Rate this article:

Previous Next

‹ 1 2 3 4 ›