This article examines the recurrence obtained for Catalan numbers based on their description of the number of unique binary search tree structures possible with \(n\) distinct keys. The straightforward recurrence obtained, when transformed into a recursive method, obtains the solution in \(3^n\) method invocations. With a small change based on symmetry, this can be changed to obtain the solution in \(2^n\) method invocations. Of course, one can obtain the solution in massively fewer operations (\(O(n^2)\)) in a dynamic programming/memoized solution.

1 Problem Statement
Nick Parlante, in an article “Binary Trees”, copyrighted 2000/2001, available on the Stanford University web site, includes this challenge [1]:

12. countTrees()
This is not a binary-tree programming problem in the ordinary sense — it’s more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.)

Suppose you are building an \(N\) node binary search tree with the values 1...\(N\). How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values. For example, countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it’s just a counting problem.

The first few examples are easy to construct: there is only one binary search tree with no items, bst(0), and similarly only one with a single item, bst(1). After that, things start growing. There are two trees of size two (Figure 1).

After that, different permutations can generate the same tree structure, such as BAC and BCA. Thus bst(3) is 5, not 6, 3! (Figure 2) and bst(4) is 14, not 24, 4! (Figure 3). See next page for Figures 2 and 3.

We can model the problem as building binary search trees of the numbers from 1 to \(N\). The recurrence can easily be seen as constructing a tree of these \(N\) keys based on selecting one of the numbers as the root for that tree, followed by completing the tree based on adding two subtrees totaling \(N-1\) keys as the left- and right-subtrees to the root. The base case is the empty tree, of which there is only one example. As the left subtree contains more and more entries, the right subtree contains fewer and fewer. This can be represented by the following recurrence, pulled from Wikipedia, which is the recurrence relationship for Catalan numbers [2]:

\[
C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \quad \text{for} \quad n \geq 0
\]

This is the recurrence Nick Parlante encodes in his solution to the problem [3].

```java
/*
 * For the key values 1…numKeys, how many structurally unique
 * binary search trees are possible that store those keys?
 * Strategy: consider that each value could be the root.
 * Recursively find the size of the left and right subtrees.
 */
public static int countTrees(int numKeys) {
    if (numKeys <=1) {
        return(1);
    }
    else {
        int sum = 0;
        int left, right, root;
        for (root=1; root<=numKeys; root++) {
            left = countTrees(root-1);
            right = countTrees(numKeys - root);
            // number of possible trees with this root
            sum += left*right;
        }
        return(sum);
    }
}
```
If one includes a global counter within the method, one can capture the total number of method invocations required to compute the number of $N$-node trees and see that there are exactly $3^N$.

Side note: the majority of the work for this paper was done before I encountered Nick Parlante’s paper, based on a version of the above recurrence formulated for $C_n$ rather than $C_{n+1}$. Hence the loops and summations in my work are for $[0...n-1]$ rather than for $[1...n]$.

There is a mirror symmetry within this problem: the trees with a $j$-node left subtree and a $k$-node right subtree are simply mirror images of the trees with a $k$-node left subtree and a $j$-node right subtree. Consequently, one can drive the loop to compute half of the trees and allow the symmetry correction to give the other half. If $n$ is an odd number, then there is one special case (with left- and right-subtrees of size $n/2$) that needs to be added.

```java
static long bst2(int n)
{
long accum = 0L;
int j = 0, k = n-1;
if (n == 0) return 1;
while (j < k)
accum += (bst2(j++) * bst2(k--))<<1; // *2 via left shift
if (j == k) // Odd n
{
long temp = bst2(j);
accum += temp * temp;
}
return accum;
}
```

The startling result, however, is that a global counter within the method shows that the total number of method invocations is now $2^n$ to solve the problem.

2 Method Invocation Modeling

From the computational code, one can obtain program schemata to model just the method invocations involved. These count up the method invocation that enters the method and then those coming from the recursion. The code below uses memoization.

```java
static long[] b1, // Calls to compute bstC1, created in main
   b2; // Calls to compute bstC2

static long bstC1(int n)
{
if (b1[n] == 0)
{
long accum = 1; // The call that got us here
int j; // Loop variable
for (j = 0; j < n; j++)
accum += bstC1(j) + bstC1(n-j-1);
b1[n] = accum;
}
return b1[n];
}

static long bstC2(int n)
{
if (b2[n] == 0)
{
long accum = 1; // The call that got us here
int j = 0, k = n-1; // Loop variables
while (j < k)
accum += bstC2(j++) + bstC2(k--);
if (j == k)
accum += bstC2(j);
b2[n] = accum;
}
return b2[n];
}
```
3 Analytic Proofs of Behaviors

The method call behavior for the first recurrence, \( bst_1 \), is slightly easier to prove: it turns into a power series. That is, \( bst_1(n) = 3^n \). In addition, it provides an example of extremely strong induction: the proof depends not on a couple of earlier results (like the Fibonacci series), but on all of the earlier results.

Base case: \( bst_1(0) = 1 \)
Inductive proof

\[
\begin{align*}
bst_1(n>0) &= 1 + \sum_{j=0}^{n-1} bst_1(j) + bst_1(n-1-j) & \text{Recurrence} \\
&= 1 + \sum_{j=0}^{n-1} bst_1(j) + \sum_{k=0}^{n-1} bst_1(k) & \text{Change of variable} \\
&= 1 + 2 \cdot \sum_{j=0}^{n-1} bst_1(j) & \text{Simplification} \\
&= 1 + 2 \cdot \sum_{j=0}^{n-1} 3^j & \text{Inductive hypothesis} \\
&= 1 + 2 \cdot \frac{3^n - 1}{2} & \text{Power series, } c = 3 \\
&= 3^n & \text{Simplification}
\end{align*}
\]

The method call behavior for the second, \( bst_2 \), requires separate proofs for even and odd \( n \). It also collapses into a power series, but now \( bst_2(n) = 2^n \).

Base case (for both even and odd \( n \)): \( bst_2(0) = 1 \)
Inductive proof, even \( n \):

\[
\begin{align*}
bst_2(n>0) &= 1 + \sum_{j=0}^{(n/2)-1} bst_2(j) + bst_2(n-1-j) & \text{Recurrence} \\
&= 1 + \sum_{j=0}^{(n/2)-1} bst_2(j) + \sum_{k=n/2}^{n-1} bst_2(k) & \text{Change of variable} \\
&= 1 + \sum_{j=0}^{n-1} bst_2(j) & \text{Simplification} \\
&= 1 + \sum_{j=0}^{n-1} 2^j & \text{Inductive hypothesis} \\
&= 1 + (2^n - 1) & \text{Power series, } c = 2 \\
&= 2^n & \text{Simplification}
\end{align*}
\]
Preferable method

As is the case with many recursive methods, this easily generates a solution based on memoization, computing each result exactly once. Since the language used is Java, the memoization vector is a global one.

\[
\text{BSTvect} = \{ 1l \}; // Memoization vector
\]

\[
\text{BSTvect}[n] = 1 + \sum_{j=0}^{[n/2] - 1} \text{BST}(j) \cdot \text{BST}(n-j-1);
\]

Simplification

For each uncomputed BSTvect[n], the calculation requires n multiplications and additions, and each cell is computed exactly once. Thus filling the memoization vector of size n will be accomplished in O(n^2) time [5], nicely exchanging base and exponent from 2^n.

Summary

This provides an example of how one can make a small change in an algorithm and reap massive benefits in the algorithm’s performance. Proving the behaviors could be an interesting problem for upper division undergraduates, particularly if they are required to obtain the recurrences from the original algorithms.

References

[4] For n = 1, \[\sum_{j=0}^{n-1} j = 0\], reflecting a loop that executes zero times.
[5] \[\sum_{j=0}^{n-1} j \cdot \binom{n}{j} = 2^n\]

TIMOTHY ROLFE
Emeritus, Department of Computer Science, Eastern Washington University, 319F CEB Cheney, Washington 99004-2493, USA
http://penguin.ewu.edu/~trolfe
Timothy.Rolfe@mail.ewu.edu

Categories and Subject Descriptors: D.2.8 Metrics — Performance measures; F.2.2 Nonnumerical Algorithms and Problems — Computations on discrete structure

General Terms: Algorithms, Performance

Keywords: Catalan number, Binary search tree, Optimization

DOI: 10.1145/2038876.2038889 © 2011 ACM 2153-2184/11/12 $10.00