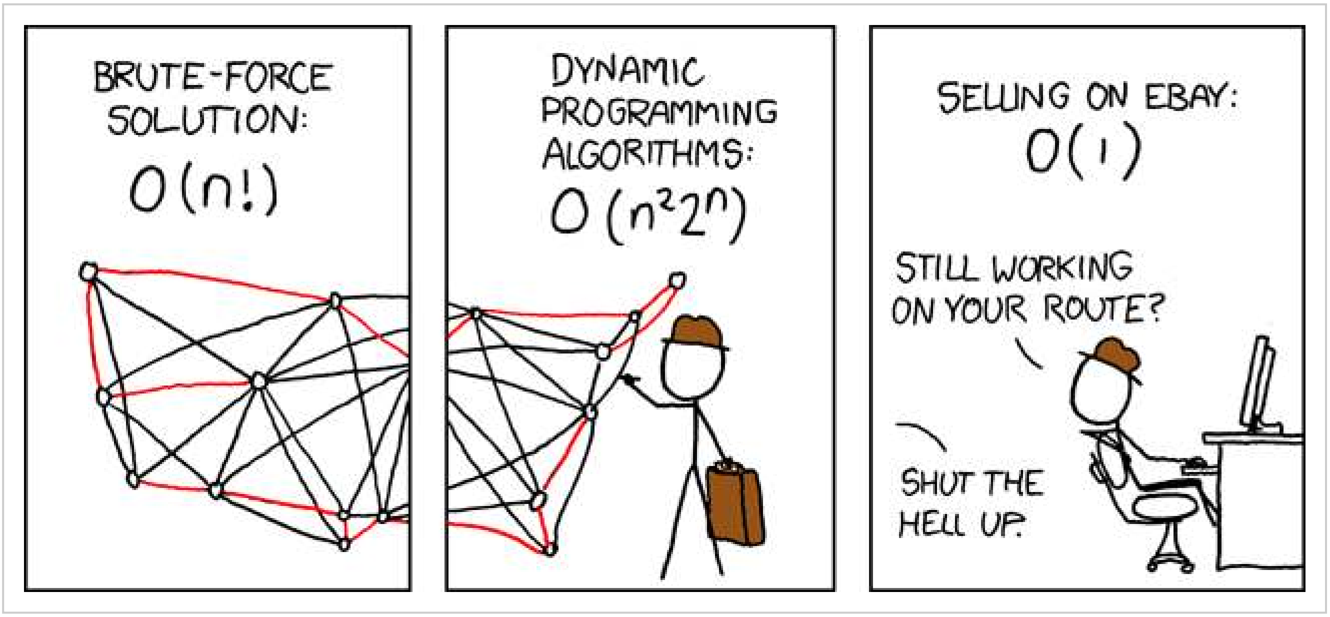
Dynamic Programming

This chapter works on a sample dynamic programming problem to show how complexity for this class of problems can be worked out.



Dynamic programming problems are similar to the divide-and-conquer problems with one important difference: The subproblems in divide-and-conquer are distinct and disjoint, whereas in the case of dynamic programming, the subproblems may overlap with one another. Also, dynamic programming problems can be solved in a bottom-up fashion instead of just a top-down approach.

Computing Fibonacci numbers is a textbook example of a dynamic programming problem. Furthermore, dynamic programming problems are usually optimization problems, and there may be several optimal solutions. However, the *value* of the optimal solution will be same.

The word "programming" in dynamic programming doesn't mean coding in the literal sense; in fact, it was used to mean a *tabular* solution.

Since dynamic programming problems involve subproblems that are overlapping or common, the answer to these subproblems can be stored in a table and reused when needed. This approach is called memoization and avoids the need to recompute the answer to subproblems each time they are encountered.

The scope of this text is limited to teaching evaluation of time and space complexities for various algorithms categories. Therefore, we'll restrict ourselves to one simple DP problem example and slice and dice it various ways to understand how complexity can be worked out for DP class of algorithms.

String Sum Representation

You are provided a positive integer *n* and asked to construct all strings of 1s, 2s, and 3s that would sum up to n. For example,

if n = 3, then the following strings will sum up to n:

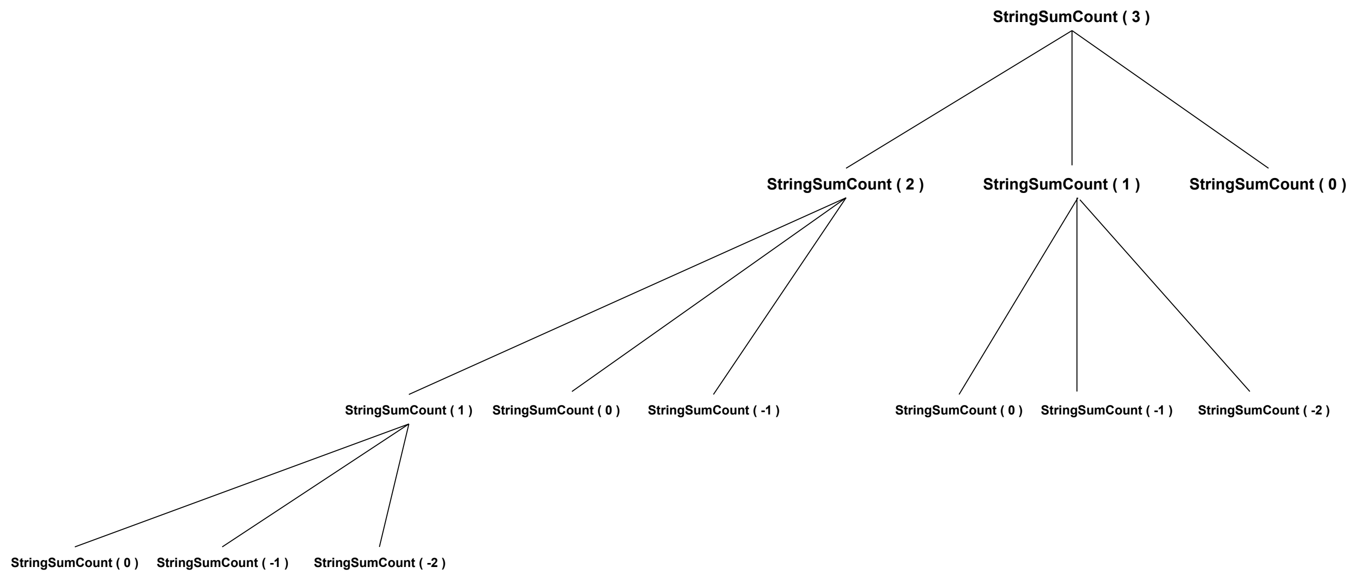
- 111
- 12
- 21
- 3

Solution

Think of 1s, 2s, and 3s as building blocks to construct n. Let's say we pick 1 as the first block. If we know how many strings can represent the integer *(n-1)*, we can deduce that the same number of strings can also represent n, as prefixing 1 to the strings making up the integer *n-1* would make up n. This leads to a recursive solution, where we ask ourselves how many strings can sum up to integers *n-1*, *n-2*, and *n-3*? We can prefix those strings with 1, 2, or 3 respectively to sum up to *n*.

We also need to define our base case. If we keep making recursive calls by subtracting 1, 2, and 3 from *n*, we'll eventually hit 0 or a negative number. A negative number implies that the string can't sum up to n and we return 0. However, if we hit 0, we know that the string we are currently considering neatly sums up to *n* and is one of the solutions. We return 1 to account for this solution.

Below is the recursion tree for the algorithm.



Recursion Tree

The recursive code for the problem appears below. From drawing out the recursion tree, one can observe that the same problems are being solved repeatedly. For instance the *StringSumCount(2)* is solved twice. As mentioned earlier, dynamic programming problems exhibit overlapping subproblems.

String Sum Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        long start = System.currentTimeMillis();
        int result = StringSumCount(5);
        long end = System.currentTimeMillis();
        System.out.println("Number of Strings : " + result + "    Time taken = " + (end - sta
    }

    static int StringSumCount(int n) {

        if (n == 0) {
            return 1;
        }

        if (n < 1) {
            return 0;
        }

        int sum = 0;

        // Number of ways to represent with 1
        sum += StringSumCount(n - 1);

        // Number of ways to represent with 2
        sum += StringSumCount(n - 2);

        // Number of ways to represent with31
        sum += stringSumCount(n - 3);

        return sum;


    }
}
```

The above code is recursive and inefficient, but before we improve upon it we'll figure out the time complexity using the recursion tree method. Let's start with the tree height or the number of levels the recursion tree can possibly have. The longest string that can sum up to a given n will consist of exactly n 1s. The longest path that we'll traverse from the root to a leaf would consist of all 1s, which implies the number of levels would be n+1. The work done in each recursive call is constant since we are only executing statements that take constant time. However, a newbie mistake would be to think that the time complexity would be the product of (n+1)*O(1). Even though the work done in each recursive call is constant, the number of recursive calls are exponentially increasing at each level!

The right way to think about complexity in this scenario is to think about the number of nodes that'll be generated in the recursion tree. Each node represents a recursive call with a constant execution time; therefore, the complexity can be expressed as:

$$T(n) = Number\ of\ Nodes * O(1)$$

Calculating an upper bound on the number of nodes in the recursion tree is easy. We simply work out the number of nodes for a full tree with every parent having exactly three children. At level 0 there is a single root, at level 1 there are 3 nodes, at level 2 there are 9 nodes, so and so forth. At the *n+1*th level the number of nodes will be 3^n. The sum of all the nodes will then be,

$$Numberofnodes = 3^0 + 3^1 + 3^0 \ldots 3^n$$

At this point, it is clear that the algorithm we devised is exponential in

At this point, it is clear that the algorithm we devised is exponential in complexity. Even if you don't know how to mathematically transform the

above summation into a neat looking formula, you can conclude two facts: One, the number of nodes will be at least 3^n or $\Omega(3^n)$; Two, the number of nodes must be less than 3^{n+1} or $O(3^{n+1})$. The intuition for the second fact comes from experience with binary trees, where a tree with n levels has 2^n-1 nodes. We can extend the same concept and quickly work out a few verified examples to see that a n level 3-ary tree must have less than 3^{n+1} nodes. In summary, we can claim that the time complexity of our recursive algorithm will be $O(3^{n+1})$ or $O(3^n)$.