Top Down and Bottom Up Approaches

This chapter continues the discussion on the complexity of dynamic programming problems.

Solution

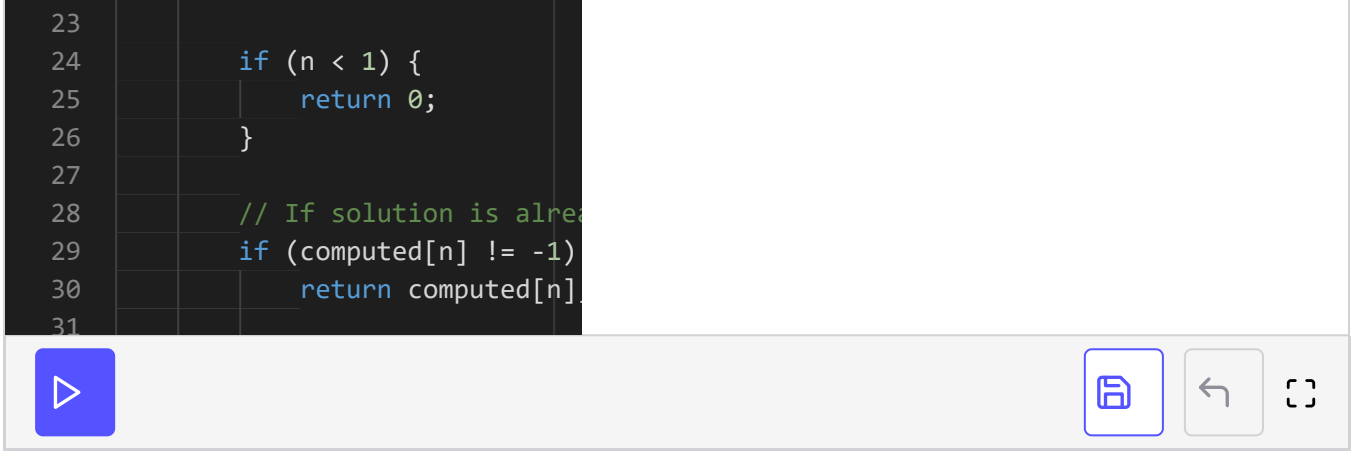
In the previous section, we discussed a recursive solution with exponential complexity. Now we'll look at two ways to solve the same problem, first using the DP top-down approach and then the DP bottom-up approach.

We saw in the recursive solution that subproblems are solved over and over again. We can store the solutions of the subproblems and avoid calculating them repeatedly. This is called *memoization*, where the algorithm remembers solutions to previously computed problems.

The only change required to make our recursive algorithm polynomial is to start storing the solutions to the subproblems; the necessary code appears below.

Top Down Implementation

```
1  class Demonstration {
2
3      static int[] computed;
4
5      public static void main( St
6          long start = System.cu
7
8          int n = 50;
9          computed = new int[n +
10         for (int i = 1; i < n -
11             computed[i] = -1;
12
13         System.out.println(Str
14         long end = System.curre
15         System.out.println("Ti
16     }
17
18     static int StringSumCount(
19
20         if (n == 0) {
21             return 1;
22         }
```



Top Down Approach

Note that we can easily compute the value for n=50, but it will timeout if we try the same value for the recursive solution. Now let's try to reason about the time complexity for our modified algorithm, which is recursive but no longer exponential. The key insight is to realize that we compute each subproblem once. If we can find the number of problems actually computed, we'll get a handle on the complexity. We know that we'll solve the maximum number of subproblems when we generate the longest string consisting of all 1s. The subproblems will include:

StringSumCount(0), StringSumCount(1), StringSumCount(2), ... StringSumCount(n)

Above is the complete list of problems that need to be solved, and it is easy to see that the number of problems will be exactly *n+1* (i.e from 0 to *n*). The complexity will be O(n+1) or O(n). Note how we determined the complexity without undertaking any extensive mathematical analysis!

Also, take stock that storing the solution to subproblems requires extra space. Since there are n+1 problems, we'll need to store n+1 solutions requiring O(n) space.

Bottom Up Approach

We solved the problem using a recursive solution, then a top-down DP solution and finally we'll work out a DP bottom-up solution. In the bottom-up approach, we'll work on the subproblems. From the solution of the subproblems, we'll construct the solution to the next subproblem higher up in the hierarchy. Let H(n) define the number of strings that can sum up to n. The base case appears below:

H (0) = 0

Now we express the solution for H(1) as follows:

H (1) = if 1 - 1 >= 0 then H(1 - 1) +
if 1 - 2 >= 0 then H(1 - 2) +
if 1 - 3 >= 0 then H(1 - 3)

Or more generally,

H (n) = if n - 1 >= 0 then H(n - 1) +
if n - 2 >= 0 then H(n - 2) +
if n - 3 >= 0 then H(n - 3)

The intuition for this recurrence is the same as before. We can tack on a 1 as the prefix to all the strings whose characters sum up to *n-1* to get strings that'll sum up to *n*. Similarly, we can prefix strings with a '2' or a '3'. The implementation can be simplified by initializing the solution array for n = 0, 1, 2, and 3. This helps us avoid putting the if conditions.

Bottom Up Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        long start = System.currentTimeMillis();
        int n = 50;
        System.out.println(StringSumCount(n));
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static int StringSumCount(int n) {

        int[] computed = new int[n + 1];
        computed[0] = 0;
        computed[1] = 1;
        computed[2] = 2;
        computed[3] = 4;

        for (int i = 4; i < n + 1; i++) {
            computed[i] = computed[i - 1] + computed[i - 2] + computed[i - 3];
        }

        return computed[n];
    }

}
```

Bottom-Up Iterative Solution

It is trivial to deduce that the complexity for the iterative bottom-up approach is O(n) which is the length for which the *for* loop runs for.

Comparision

To better appreciate the performance improvements we get by converting an exponential algorithm into a polynomial one, we ran all of the 3 algorithms on different values of n and measured the time it took for the algorithm to complete. This is by no means a definitive or comprehensive test, but it let us crudely measure the performance of the different algorithms. The times are in milliseconds.

n	Recursive	Top Down	Bottom Up
5	0	0	0
25	87	0	0
30	484	0	0
35	7971	0	0
40	182736	0	0

The recursive solution performs very poorly compared to the other two dynamic programming solutions. This should help you appreciate how different implementations using different algorithm strategies can drastically affect running times.