CS 331: Algorithms and Complexity (Spring 2024)

Unique numbers: 50930/50935/50940/50945

Discussion Section 5

Problem 1

Solve the following recurrences.

1. $T(n) = 9T(n/3) + n + n^2 + 1$

   We can use the Master theorem. $a = 9$, $b = 3$, and $f(n) = n^2 + n + 1$. $\log_3 9 = 2$, and $f(n) \in \Theta(n^2)$, so this is case two of the theorem. Therefore, $T(n) = \Theta(n^2 \log n)$.

2. $T(n) = 2T(2n/9) + \sqrt{n}$

   We can use Cormen's Master theorem. $a = 2$, $b = 9/2$, and $f(n) = \sqrt{n} = n^{1/2}$. We are looking for $n^{\log_{9/2} 2}$, and to compare it to $n^{1/2}$.

   Some simple arithmetic with the change-of-base formula for logarithms will show that $\log_{9/2} 2 \approx 0.46$, so case 3 from the Master theorem applies. We need to check the so-called regularity condition, which states that $af(n/b) < cf(n)$ for some $c < 1$.

   Since $a * f(n/b) = 2\sqrt{2n/9} = (2\sqrt{2}/3)\sqrt{n} < \sqrt{n}$. So we could pick c from the range $[2\sqrt{2}/3, 1)$ and the regulatory condition holds, and we can use the master theorem to conclude that $T(n) = \Theta(\sqrt{n})$.

3. $T(n) = 2T(n/11) + \sqrt{\sqrt{n}}$

   Again, we can use the master theorem. This time, we would like to compare $n^{\log_{11} 2}$ and $n^{1/4}$.

   We can show this by changing the base of the logarithm and observing what happens to the output (if you don't like this, use the change-of-base formula and skip ahead).

   - $2^1 = 2$, so $\log_2 2 = 1$.
   - $4^{1/2} = 2$, so $\log_4 2 = 1/2$.
   - $8^{1/3} = 2$, so $\log_8 2 = 1/3$.
   - $16^{1/4} = 2$, so $\log_{16} 2 = 1/4$.

Notice that as the base of the logarithm grows, the output shrinks. This is true for any two integer logarithm bases. The proof is left as an exercise to the reader, though it follows almost trivially from the results on this page.

This means that, since $11 < 16$, $\log_{11} 2 > \log_{16} 2$, which implies that $n^{\log_{11} 2} > n^{1/4}$.

This puts us in case 1 of the master theorem, which tells us that $T(n) = \Theta(n^{\log_{11} 2})$.

4. $T(n) = T(n-2) + O(1)$

   We cannot apply the master theorem here. However, this is a very simple recurrence to solve by simple examination. Each step takes constant time to perform, and there are no more than $n/2$ steps in the entire recurrence. Therefore, $T(n) = O(n)$.

5. $T(n) = 2T(n^{1/4}) + 1$ given that $n \geq 2, T(2) = 1$

   We cannot apply the master theorem here since the size of each n does not decrease by a constant factor of b. We can solve this problem by the recurrence tree.

   At each node, the "cost" of that node (which is actually T) is the the sum of the costs of its two children plus 1. We can regard the 1 as the extra cost of this node. Hence, to get the cost of the root node, which is $T(n)$, we have to get the cost of its two children. Therefore, we can calculate the cost recursively and we can get that the cost of the root node will be the sum of extra costs of all the nodes in the tree, according to the recurrence relation.

   Suppose that there are k levels of this tree, we know that at level k the size of n will be $n^{1/4^k} = 2$ since 2 is the base case. We can get $k = \frac{1}{2} \log \log n$.

   Since the extra cost at each node is all 1. The total cost of the root node is just the total number of nodes in this tree, which is calculated as:

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 = 2^{\frac{1}{2} \log \log n + 1} - 1 = 2\sqrt{\log n} - 1$$

   Therefore, $T(n) = 2\sqrt{\log n} - 1$.

6. $T(n) = 2T(n) + n^2$

   This is not a valid recurrence: it does not ever reach a base case. If you ever write down a recurrence like this to describe a program you wrote, something has gone very wrong.

Problem 2

You are climbing a stair case. It takes n steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution. We solve this problem by the following steps:

1. **Define Subproblems:**
   Let S_n be the number of ways to reach the n^{th} step.

2. **Find the Recurrence:**

   - Consider one possible solution $n = x_1 + x_2 + \cdots + x_m$, where x_i can be either 1 or 2.
   - If $x_m = 1$, the rest of the terms must sum to $n - 1$.
   - Thus, the number of sums that end with $x_m = 1$ is equal to S_{n-1}.
   - Similarly, if $x_m = 2$, the number of sums that end with $x_m = 2$ is equal to S_{n-2}

   Now we have the recurrence: $S_n = S_{n-1} + S_{n-2}$.

3. **Recognize and Solve the Base Problem:**
   When $n = 1$, we have only 1 way to climb to the top.
   When $n = 2$, we have two ways, namely 2 steps by one time and 1 step by two times.

4. **Design the Algorithm:**

Algorithm 1 Climbing Stairs

```
    Input: n
 1: if n ≤ 2 then
 2:     return n
 3: else
 4:     Set prev_1 to be 2 (S_{n-1})
 5:     Set prev_2 to be 1 (S_{n-2})
 6:     Set cur_cnt to be 0 (S_n)
 7:     for i from 3 to n do
 8:         cur_cnt = prev_1 + prev_2
 9:         prev_2 = prev_1
10:         prev_1 = cur_cnt
11: return cur_cnt
```

Problem 3

Given n, how many structurally unique binary search trees (BST's) can you construct that store values $1 \dots n$?
For example, given $n = 3$, there are a total of 5 unique BST's:

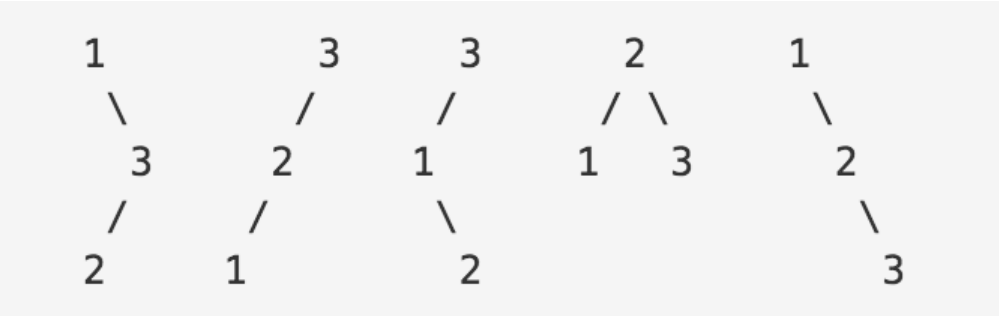

Figure 1: All the unique BST's given $n = 3$

Solution. We still apply the four steps to solve this problem:

1. **Define Subproblems:**
   Notice that with arbitrary n consecutive positive integers, we always have the same number of unique BST's, regardless of the actual range of these n numbers. Therefore, we can always assume that the range is from 1 to n.
   Let T_n be the number of unique BST's can be constructed by any n consecutive intergers. Let T_n^i be the number of BST's rooted at i. Therefore, we have $T_n = \sum_{i=1}^{n} T_n^i$
2. **Find the Recurrence**
   - Consider now we have a BST rooted at i, all the nodes with values less than i should be in the left subtree of the root node, hence there are $i - 1$ nodes in the left subtree. Similarly, there are $n - i$ nodes in the right subtree.
   - According to the recursive definition of BST, we know that both the left subtree and the right subtree of the root node are also BST. Therefore, the number of unique left subtrees is T_{i-1} and the number of unique right subtrees is T_{n-i}.
   - For each unique left subtree, it can be combined with any of the right subtrees. Therefore, the total number of unique BST's rooted at i should be $T_n^i = T_{i-1} \times T_{n-i}$.
   - The final goal is to get $T_n = \sum_{i=1}^{n} T_n^i$
3. **Recognize and Solve the Base Problem:**
   We have: $T_0 = 1, T_1 = 1$
4. **Design the Algorithm:**

Algorithm 2 Unique Binary Search Trees

getUniqueBST(n)

We use an Array A with size $n+1$, first all initialized to 0, $A[i]$ indicating T_i.
cnt is the incremental counter initialized to 0
/*Base Case*/
$A[0] = 1$ and $A[1] = 1$
/*Let each i to be the root*/
for i from 1 to n **do**
    /*Check the number of unique BST's of the left subtree*/
    if $A[i-1] = 0$ **then** /*If not calculated yet, recursively call the function*/
        $A[i-1]$ =getUniqueBST$(i-1)$
    if $A[n-i] = 0$ **then** /*If not calculated yet, recursively call the function*/
        $A[n-i]$ =getUniqueBST$(n-i)$
    $cnt = cnt + A[i-1] * A[n-i]$
return cnt