Analysis of Recursion

In the previous post, we discussed analysis of loops. Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1. **Substitution Method**: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

```
For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induct
ion
to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it
is true
for values smaller than n.

T(n) = 2T(n/2) + n
    <= 2cn/2Log(n/2) + n
    =  cnLogn - cnLog2 + n
    =  cnLogn - cn + n
    <= cnLogn
```
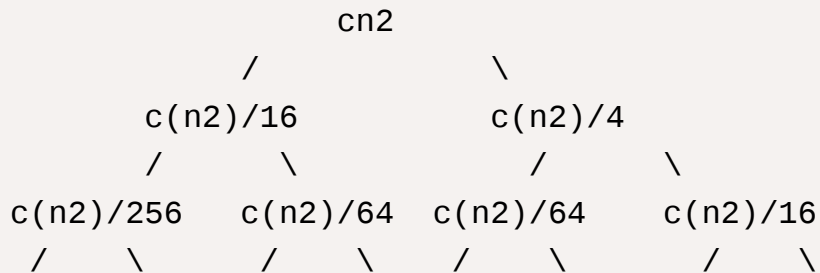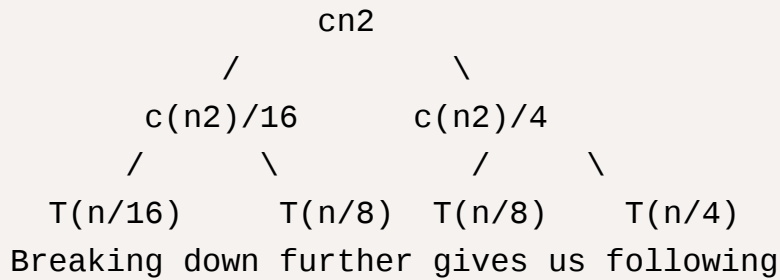
2. **Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

```
For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn2
```

```
        cn2
       /      \
   T(n/4)    T(n/2)
```
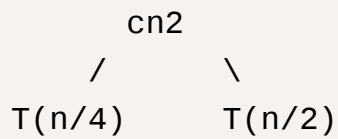
If we further break down the expression T(n/4) and T(n/2),
we get following recursion tree.

```
             cn2
          /          \
     c(n2)/16      c(n2)/4
     /      \       /      \
 T(n/16)   T(n/8) T(n/8)   T(n/4)
```
Breaking down further gives us following
```
              cn2
           /          \
      c(n2)/16        c(n2)/4
      /      \         /      \
c(n2)/256  c(n2)/64 c(n2)/64   c(n2)/16
  /    \     /    \   /    \      /    \
```

To know the value of T(n), we need to calculate sum of tree
nodes level by level. If we sum the above tree level by lev
el,
we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/1
6.

To get an upper bound, we can sum the infinite series.
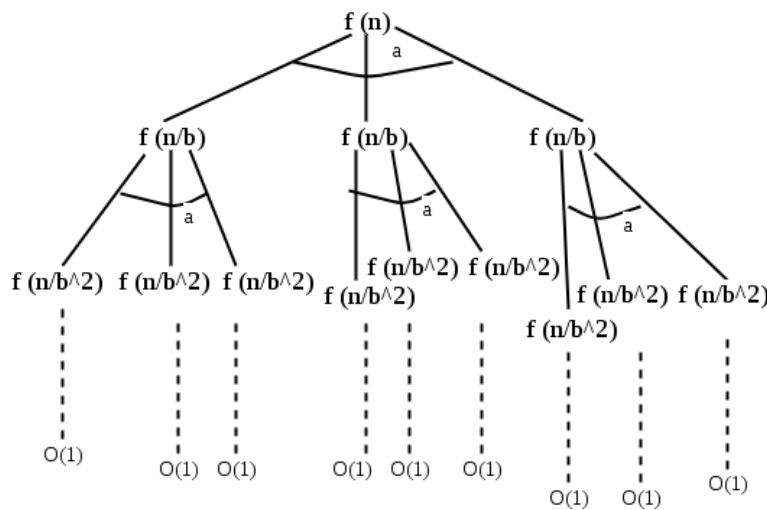We get the sum as (n2)/(1 - 5/16) which is O(n2)
```

3. **Master Method:**

   Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

```
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is Logba. And the height of recurrence tree is Logbn

In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: T(n) = 2T(n/2) + $\Theta(n)$. It falls in case 2 as c is 1 and Logba] is also 1. So the solution is $\Theta(n \text{ Log } n)$

Binary Search: T(n) = T(n/2) + $\Theta(1)$. It also falls in case 2 as c is 0 and Logba is also 0. So the solution is $\Theta(\text{Log } n)$

Notes:

1. It is not necessary that a recurrence of the form T(n) = aT(n/b) + f(n) can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence T(n) = 2T(n/2) + n/Log n cannot be solved using master method.

2. Case 2 can be extended for $f(n) = \Theta(n^c \, \text{Log}^k \, n)$

If $f(n) = \Theta(n^c * (\text{Log}^k) * n)$ for some constant $k >= 0$ and $c = \text{Log}_b a$, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$