Text:

1. Iteration Method

In the iteration method we iteratively "unfold" the recurrence until we "see the pattern". The method of iterating a recurrence doesn't require us to guess the answer, but it may require more algebra than the substitution method. The idea is to expand (iterate) the recurrence and express it as a summation of terms dependent only on *n* and the initial conditions. Techniques for evaluating summations can then be used to provide bounds on the solution.

The iteration method usually leads to lots of algebra, and keeping everything straight can be a challenge. The key is to focus on two parameters: the number of times the recurrence needs to be iterated to reach the boundary condition, and the sum of the terms arising from each level of the iteration process. Sometimes, in the process of iterating a recurrence, you can guess the solution without working out all the math. Then, the iteration can be abandoned in favor of the substitution method, which usually requires less algebra. When a recurrence contains floor and ceiling functions, the math can become especially complicated. Often, it helps to assume that the recurrence is defined only on exact powers of a number.

Let us consider the following snippet of code and generate the recurrence for the same.

```
void disp (int n)
{
        if(n>1)                                          ----------------------------> 1
        {
                for(i=0;i<n;i++)
                        printf("%d", n);                 ----------------------------> n
                disp(n/2);                               ----------------------------> T(n/2)
                disp(n/2);                               ----------------------------> T(n/2)
        }
}
```

We get the recurrence as:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

$T(n) = 2T(n/2) + n$

$= 2\{2T(n/2^2) + n\} + n$

$= 2^2T(n/2^2) + 2n$

$= 2^2\{2T(n/2^3) + n\} + 2n$

$= 2^3 T(n/2^3) + 3n$

$= \ldots$

$= 2^k T(n/2^k) + kn$ *(1)*

Here, the stopping condition is *T(1) i.e. $n/2^k = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$*
Using in *(1)*

$T(n) = n\,T(1) + (\log n)\,n \Rightarrow n + n \log n \Rightarrow O(n \log n)$

Example:

Let us look at another snippet of code and generate the recurrence for the same.

```
void disp (int n)
{
        if(n>0)                          ----------> 1
        {
                printf("%d", n);         ----------> 1
                disp(n-1);               ----------> T(n-1)
        }
}
```

We get the recurrence as:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

T(n) = T(n-1) + 1
= [T(n-2) + 1] + 1
= T(n-2) + 2
= [T(n-3) + 1] + 2
= T(n-3) + 3
T(n) = T(n-k) + k
Here stopping condition is *T(0) => n-k = 0 => n=k*
T(n) = T(0) + n
=> T(n) = O(n)

2. Recursion Trees

A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem. We can sum up the numbers in each node to get the cost of the entire algorithm.

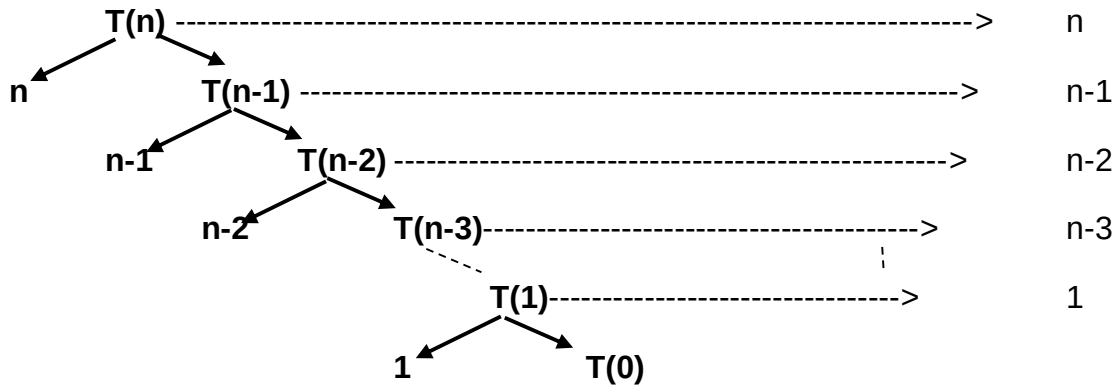
Note: We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them. However, if you are very careful when drawing out a recursion tree and summing the costs, you can actually use a recursion tree as a direct proof of a solution to a recurrence.

If we are only using recursion trees to generate guesses and not prove anything, we can tolerate a certain amount of "sloppiness" in our analysis. For example, we can ignore floors and ceilings when solving our recurrences, as they usually do not affect the final guess.

Let us take the following recurrence and try to solve it using iteration tree method

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Generating the recursion tree for *T(n)*, we get



n +(n-1) + (n-2) + … + 1 = n(n+1)/2 = n^2

therefore, *T(n) = O(n^2)*