Recurrences, Solution of Recurrences, and Recursion Tree

Recursion is a powerful reduction technique, where a problem is solved by breaking it down into smaller, more manageable parts. When an algorithm calls itself, its running time can often be described by a recurrence equation, which expresses the overall running time of a problem of size n in terms of the running time on smaller inputs.

1. Substitution Method

The substitution method comprises of three steps:

1. **Guess the Solution:** We guess the form of the solution based on the given recurrence relation.
2. **Verify by Induction:** We verify the correctness of our guess by induction, applying it to smaller values.
3. **Solve for Constants:** We solve for the constants in our guessed solution to obtain the final answer.

Example
Recurrence: $T(n) = 4T\left(\frac{n}{2}\right) + n$ and $T(16)$.
Step 1: Guess the solution

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

$$T(n) = 4T\left(\frac{n}{2}\right) \Rightarrow T(2n) = 4T(n) \Rightarrow T(n) = n^2$$

So, $T(n)$ is of the order n^2.
We guess that the solution is $T(n) = O(n^3)$.
Step 2: Prove $T(n) \leq cn^3$ for some constant c
Assume $T(k) \leq ck^3$:

$$\begin{gathered} T(n) = 4T\left(\frac{n}{2}\right) + n \\ \leq 4c\left(\frac{n}{2}\right)^3 + n \\ \leq cn^{\frac{3}{2}} + n \\ \leq cn^3 - \left(\frac{cn^{\frac{3}{2}}}{2} - n\right) \\ T(n) \leq cn^3 \end{gathered}$$

as $\left(\frac{cn^{\frac{3}{2}}}{2} - n\right)$ is always positive.

Therefore, we have shown that $T(n) \leq cn^3$, so $T(n) = O(n^3)$.

Step 3: Solve for constants

$$cn^{\frac{3}{2}} - n \geq 0 \Rightarrow n \geq 1 \Rightarrow c \geq 2$$

Now suppose we guess that $T(n) = O(n^2)$ which is a tight upper bound.

Assume, $T(k) \leq ck^2$:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n \\ &= 4c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 + n \end{aligned}$$

So, $T(n)$ will never be less than cn^2. But if we will take the assumption of $T(k) = c_1k^2 - c_2k$, then we can find that $T(n) = O(n^2)$.

2. Iterative Method

Example

Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + n$

$$\begin{aligned} T(n) &= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 2^2T\left(\frac{n}{2^2}\right) + n + \frac{n}{2} \\ &= 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + n + \frac{n}{2} = 2^3T\left(\frac{n}{2^3}\right) + n + \frac{n}{2} + \frac{n}{4} \\ &= 2^kT\left(\frac{n}{2^k}\right) + kn \text{ after } k \text{ iterations.} \end{aligned}$$

Sub-problem size is 1 after $n2^k = 1 \Rightarrow k = \log n$.

So, after $k = \log n$ iterations, the sub-problem size will be 1.

Substituting $k = \log n$ in the equation above, we obtain:

$$\begin{aligned} T(n) &= nT(1) + n\log n = nc + n\log n \text{ where } c = T(1). \\ &= O(n\log n). \end{aligned}$$

3. Recursion Tree Method

In a recursion tree, each node represents the cost of a single sub-problem. We sum the cost within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of recursion.

Example

Constructing a recursion tree for the recurrence $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

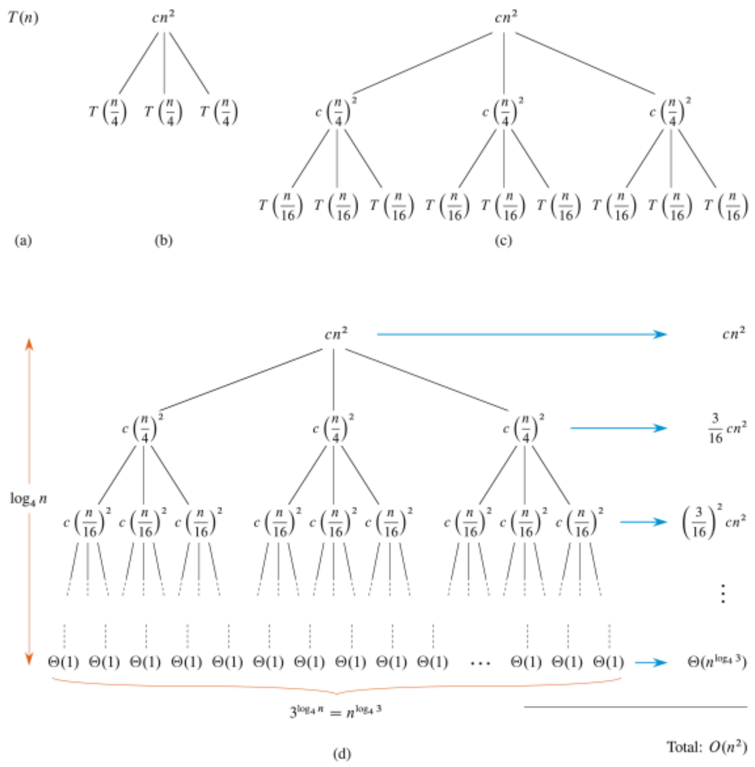

Figure 1: Example of a Recursion Tree

4. Master Theorem

The Master Theorem is a tool used to determine the time complexity of divide-and-conquer algorithms with specific types of recurrence relations. It is particularly useful for algorithms that split a problem into subproblems of equal size.

The standard form of the Master Theorem applies to recurrence relations of the following form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Here:

- a is the number of subproblems,
- b is the factor by which the input size is reduced in each subproblem,

- $f(n)$ is the cost of dividing the problem and combining the results.

The Master Theorem has three cases, and for a recurrence relation of the form above, it helps determine the asymptotic time complexity based on these cases:

1. If $f(n)$ is $O(n^{\log_b a - \epsilon})$ for some positive constant ϵ, where $\log_b a$ is greater than ϵ, then the solution is $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$ for some non-negative constant k, where $\log_b a = \epsilon$, then the solution is $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ for some positive constant ϵ, where $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then the solution is $T(n) = \Theta(f(n))$.

In a nutshell

Imagine you have a problem, and you can break it into smaller versions of the same problem. The Master Theorem helps you figure out how much time this kind of problem-solving process takes.

Here's the basic recipe:

1. **Count the Subproblems:** See how many smaller versions of the problem you create (a).
2. **Shrinkage Factor:** Figure out how much you shrink the problem size (b).
3. **Cost of Work:** Understand how much extra work you do when dividing and combining the results ($f(n)$).

Now, based on these, the Master Theorem gives you three cases:

1. If the extra work is not too much compared to the number of subproblems and their size, then your total time is mainly determined by the number of subproblems and their size.
   - **Example:** If you have a subproblems, each b times smaller than the original, and the extra work is not too significant, your time is roughly $T(n) = \Theta(n^{\log_b a})$.
2. If the extra work is a bit more, like it includes a logarithmic factor, then your total time has an extra logarithmic factor.
   - **Example:** If you have a subproblems, each b times smaller, and the extra work includes $\log n$, your time is roughly $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If the extra work is quite a lot, more than the subproblems and their size combined, then your total time is mainly determined by the extra work.

- **Example:** If the extra work is significantly larger, your time is roughly the same as the extra work, $T(n) = \Theta(f(n))$.

In simpler terms, the Master Theorem helps you predict how fast a problem-solving process grows by looking at how you break down the problem and the extra work you do.