Recurrence Relations

1 Concept of Recurrence Relations

A recurrence relation is an equation that defines a sequence of values using previous values in the same sequence. In computer science and mathematics, it is often used to calculate the time complexity of recursive algorithms.

If we have a function $T(n)$, a recurrence relation expresses $T(n)$ in terms of $T(n-1)$, $T(n-2)$, or $T(n/b)$.

For example, the famous Fibonacci sequence is defined as:

$$F(n) = F(n-1) + F(n-2)$$

Here, to find the n-th number, you need to know the two numbers immediately preceding it. Every recurrence must have a **Base Case** (stopping condition) to prevent infinite loops.

2 Formation of Recurrence Relations

We can form recurrence relations by analyzing how an algorithm divides a problem into smaller sub-problems.

2.1 Recurrence from Recursive Algorithms

In a standard recursive algorithm, a function calls itself with a smaller input. Consider calculating the factorial of n ($n!$).

$$n! = n \times (n-1)!$$

The time complexity $T(n)$ requires constant time c for the multiplication, plus the time to calculate the factorial of $n-1$.

$$T(n) = T(n-1) + c$$

2.2 Recurrence from Divide and Conquer Algorithms

Divide and Conquer strategies split a problem into b sub-problems, each of size n/b. Consider Merge Sort. It divides the list into two halves ($n/2$), sorts them recursively, and then merges them.

- Dividing takes constant time.
- Solving two sub-problems takes $2T(n/2)$.
- Merging the results takes linear time cn.

The relation is:

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

3 Types of Recurrence Relations

Recurrence relations are classified based on their structure.

3.1 Linear Recurrence

A recurrence is linear if the previous terms ($T(n-1), T(n-2)$, etc.) appear in the first power (not squared, cubed, etc.).

- Example: $T(n) = 3T(n-1) + 5$ (Linear)

3.2 Non-linear Recurrence

If a previous term is raised to a power or multiplied by another previous term, it is non-linear.

- Example: $T(n) = (T(n-1))^2 + 1$ (Non-linear)

3.3 Homogeneous Recurrence

A linear recurrence is homogeneous if there is no extra constant or function of n added to the recursive terms.

- Example: $T(n) - 2T(n-1) = 0$ (Homogeneous)

3.4 Non-homogeneous Recurrence

If there is an extra term $f(n)$ (a constant or a function of n), it is non-homogeneous.

- Example: $T(n) - 2T(n-1) = n$ (Non-homogeneous due to n)

4 Methods to Solve Recurrence Relations

There are four primary methods used to solve these relations to find the asymptotic complexity (Big-O).

4.1 Substitution Method

We guess a solution (bound) and use mathematical induction to prove that the guess is correct. This method is powerful but requires a good initial guess.

4.2 Iteration Method

We expand the recurrence step-by-step. We substitute $T(n-1)$ into the equation, then $T(n-2)$, and so on, until we see a pattern (usually a summation series).

4.3 Recursion Tree Method

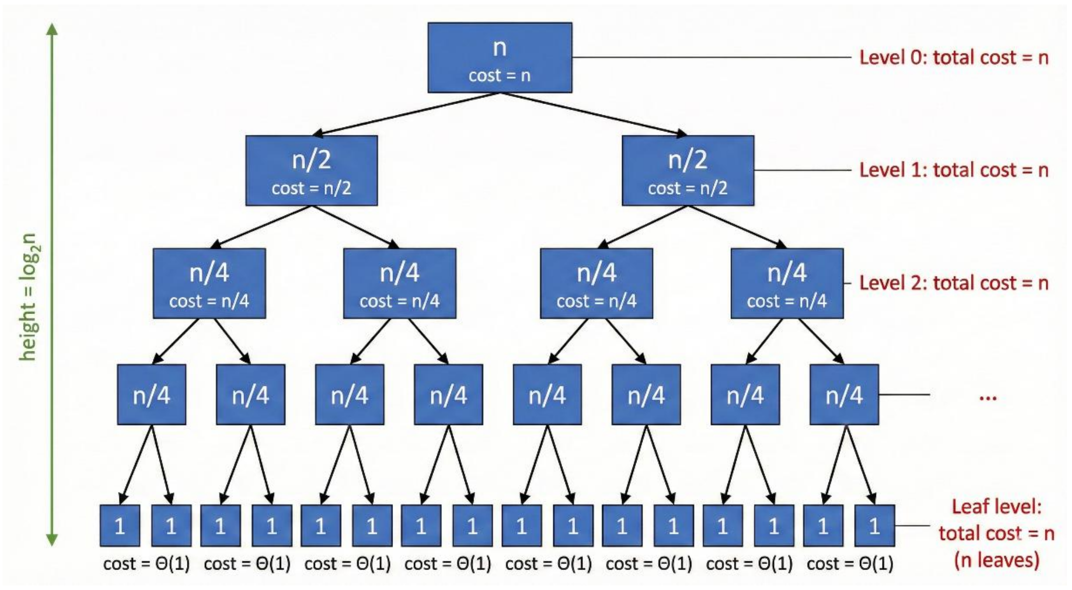
We draw a tree where each node represents the cost of a sub-problem. We sum the costs at each level and then sum the total costs of all levels.

4.4 Master Theorem

A ”cookbook” formula for solving recurrences of the form $T(n) = aT(n/b) + f(n)$. It provides a direct answer by comparing terms, without needing expansion or induction.

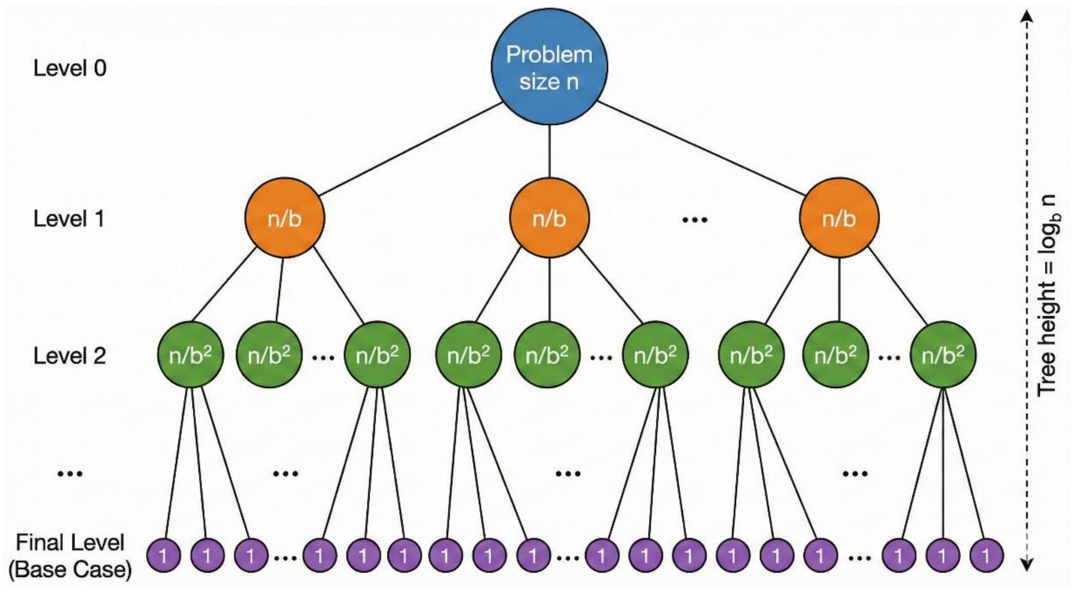
5 Analysis Using Recursion Tree

The recursion tree is a visual way to analyze Divide and Conquer algorithms.



5.1 Concept of Recursion Tree

The root of the tree represents the cost of the original problem. The children of the root represent the costs of the sub-problems. This continues until we reach the base case (leaves of the tree).



5.2 Cost per Level

We calculate the sum of costs for all nodes at a specific depth i. For example, at depth 0, the cost is usually $f(n)$. At depth 1, it might be $a \times f(n/b)$.

5.3 Height of the Tree

The height determines how many levels the recursion goes down. If the problem size divides by b at every step, the height is generally $\log_b n$.

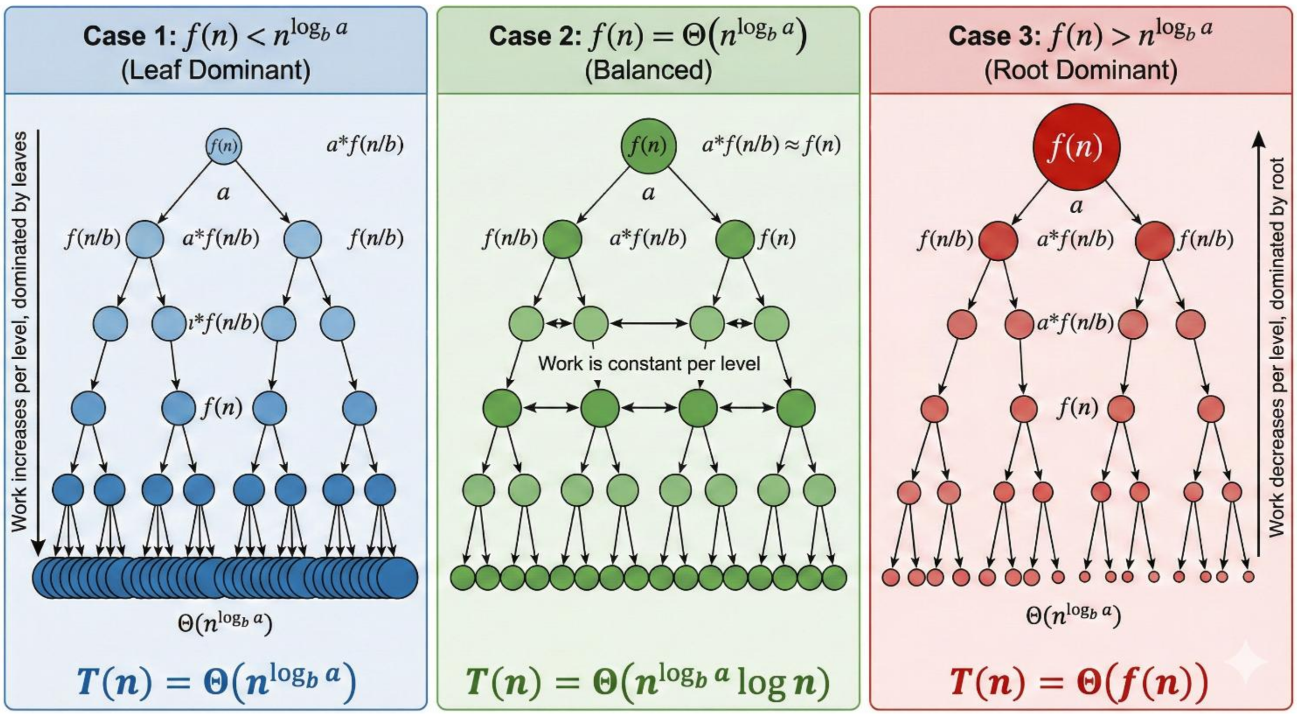
5.4 Total Cost Calculation

The total complexity $T(n)$ is the sum of costs of all levels:

$$T(n) = \sum(\text{Cost of Level } i)$$

6 Master Theorem

The Master Theorem is the quickest way to solve standard Divide and Conquer recurrences.



Case 1: $f(n) < n^{\log_b a}$
(Leaf Dominant)
$f(n)$
a
$a*f(n/b)$
$f(n/b)$
$a*f(n/b)$
$f(n/b)$
$a*f(n/b)$
$f(n)$
Work increases per level, dominated by leaves
$\Theta(n^{\log_b a})$
$T(n) = \Theta(n^{\log_b a})$
Case 2: $f(n) = \Theta(n^{\log_b a})$
(Balanced)
$f(n)$
$a*f(n/b) \approx f(n)$
a
$f(n/b)$
$a*f(n/b)$
$f(n)$
Work is constant per level
$T(n) = \Theta(n^{\log_b a} \log n)$
Case 3: $f(n) > n^{\log_b a}$
(Root Dominant)
$f(n)$
a
$f(n/b)$
$a*f(n/b)$
$f(n/b)$
$a*f(n/b)$
$f(n)$
Work decreases per level, dominated by root
$\Theta(n^{\log_b a})$
$T(n) = \Theta(f(n))$

6.1 General Form of Recurrence

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

Where:

- $a \geq 1$: The number of sub-problems.
- $b > 1$: The factor by which input size reduces.

- $f(n)$: The cost of dividing and combining.

We compare $f(n)$ with $n^{\log_b a}$.

6.2 Case 1

If $f(n)$ is smaller (polynomially) than $n^{\log_b a}$:

$$T(n) = \Theta(n^{\log_b a}) \text{ (The}$$

work at the leaves dominates).

6.3 Case 2

If $f(n)$ is roughly equal to $n^{\log_b a}$:

$$T(n) = \Theta(n^{\log_b a} \log n) \text{ (The}$$

work is evenly distributed across levels).

6.4 Case 3

If $f(n)$ is larger (polynomially) than $n^{\log_b a}$, and satisfies the regularity condition ($af(n/b) \leq cf(n)$ for some $c < 1$):

$$T(n) = \Theta(f(n)) \text{ (The}$$

work at the root dominates).

6.5 Conditions and Limitations

The Master Theorem cannot be used if:

- $T(n)$ is not monotonic.
- $f(n)$ is not a polynomial (e.g., $f(n) = 2^n$).
- a is not a constant (e.g., $a = 2n$).
- The difference between $f(n)$ and $n^{\log_b a}$ is not polynomial (e.g., differs by only $\log n$).

7 Solving Standard Recurrences

Here are solutions to common recurrences found in algorithms.

7.1 T(n) = T(n - 1) + c

This reduces size by 1 each time.

$$T(n) = O(n) \text{ Example:}$$

Linear Search.

7.2 T(n) = T(n - 1) + n

The cost increases linearly at each step.

$$T(n) = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$$

$T(n) = O(n^2)$ Example:

Bubble Sort, Selection Sort (Worst Case).

7.3 T(n) = 2T(n/2) + c

Problem size halves, but we branch twice.

$T(n) = O(n)$ Example:

Building a binary tree.

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

This matches Case 2 of the Master Theorem ($a = 2, b = 2, \log_2 2 = 1$). Since $f(n) = n^1$, they are equal.

$T(n) = O(n \log n)$ Example:

Merge Sort.

7.5 T(n) = 2T(n/2) + n log n

This is an extension of Case 2. Since $f(n)$ has an extra log factor:

$$T(n) = O(n \log^2 n)$$

8 Comparison of Methods for Solving Recurrences

- **Substitution:** Best for proving a known guess correct. Hard to use if you have no idea what the answer is.
- **Iteration:** Good for simple algebra, but calculation can get messy with complex sums.
- **Recursion Tree:** Excellent for visualization and ”getting a feel” for the answer before proving it.
- **MasterTheorem:** Thefastestmethod, butonlyworksforthespecificform $T(n) = aT(n/b) + f(n)$.

9 Practical Considerations in Recurrence Analysis

When implementing recursive solutions based on these relations, consider:

1. **Stack Overflow:** Deep recursion (like $T(n) = T(n-1) + c$) creates a stack frame for every n. For large n, this crashes the program.

2. **Overlapping Subproblems:** Recurrences like Fibonacci $F(n) = F(n-1) + F(n-2)$ recalculate the same values many times. This is inefficient ($O(2^n)$) unless Dynamic Programming is used.
3. **IntegerOverflow:** Rapidly growing functions (like factorials) exceed integer storage limits very quickly.

10 Summary of Key Concepts

- A recurrence relation defines a sequence using previous terms.
- They naturally model recursive algorithms and Divide and Conquer strategies.
- The Master Theorem is the standard tool for solving $T(n) = aT(n/b) + f(n)$.
- Understanding the ”height” and ”work per level” of a recursion tree helps visualize complexity.
- Common complexities include $O(n)$, $O(n^2)$, and $O(n \log n)$.

11 Practice Problems and Exercises

Try solving the following recurrences to test your understanding:

1. $T(n) = 3T(n/2) + n^2$ (Hint: Use Master Theorem)
2. $T(n) = T(n-1) + \log n$
3. $T(n) = 4T(n/2) + n$
4. $T(n) = 2T(n/2) + n^2$
5. $T(n) = 2T(n) + 1$ (Careful: Does this terminate?)