Rules of Game

1. Never jump to work session without an warmup.
2. When hitting bottlenecks back track to do some related easy problems you already mastered.
3. See, do, teach, apply and show off.
4. Our brain treats skin in the game and non-skin in the game differently.
5. Increase your Germane Load, load to connect with your previously solved problem.
6. Be humble, know your constraints(working memory limits(avoid long parameters, nested structure, ambigious name, no helper function, jump to low level too soon, use chunk, use familiar pattern)).
7. Deducive and guess from the first principles. ### The Master Theorem
8. Intensity(Rather work 10 minutes with 100% than 30 minutes with 70%)
9. Rest and break to make you recover
10. Stop loss, when stuck more than five minutes ask for help.
11. Fight first, do not give up under two minutes.
12. Use Pareto 80/20 at every level

First principle

The Master Theorem is a fundamental law for analyzing the time complexity of divide-and-conquer algorithms. It provides a mathematical framework that algorithm designers **must obey** when creating recursive solutions.

Theorem Statement

For recurrence relations of the form:

```
T(n) = aT(n/b) + f(n)
```

Where:

- `a ≥ 1` (number of subproblems)
- `b > 1` (factor by which problem size is reduced)
- `f(n)` is the cost of work done outside recursive calls

Three Cases (Laws of Complexity)

Case 1: Leaves Dominate

- If `f(n) = O(n^c)` where `c < log_b(a)`
- Then `T(n) = Θ(n^(log_b(a)))`
- *Law*: When recursive work dominates, complexity is determined by leaf nodes

Case 2: Balanced Work

- If `f(n) = Θ(n^c * log^k(n))` where `c = log_b(a)` and `k ≥ 0`

- Then `T(n) = Θ(n^c * log^(k+1)(n))`
- *Law*: When work is balanced across levels, add one logarithmic factor

Case 3: Root Dominates

- If `f(n) = Ω(n^c)` where `c > log_b(a)` and regularity condition holds
- Then `T(n) = Θ(f(n))`
- *Law*: When non-recursive work dominates, it determines complexity

Mandatory Application Examples

Binary Search

```
T(n) = T(n/2) + O(1)
a=1, b=2, f(n)=O(1)=O(n^0)
c=0 < log_2(1)=0 → Case 2
T(n) = Θ(log n)
```

Merge Sort

```
T(n) = 2T(n/2) + O(n)
a=2, b=2, f(n)=O(n)=O(n^1)
c=1 = log_2(2)=1 → Case 2
T(n) = Θ(n log n)
```