Lecture 2 – Recursion and Recurrence Relations: Methods and Implementation on the JVM

NURSULTAN KHAIMULDIN

N.KHAIMULDIN@ASTANAIT.EDU.KZ

Content

Recursion on the JVM

Every method invocation creates a new stack frame; the per-thread call stack is finite. Sufficiently deep recursion can therefore terminate with a StackOverflowError.

The HotSpot JVM does not guarantee tail-call optimization; tail-recursive routines should be refactored as iterative loops.

Frequent calls and transient allocations (e.g., boxing, lambda captures) introduce overhead and increase GC activity. On hot paths, prefer primitive types and contiguous arrays.

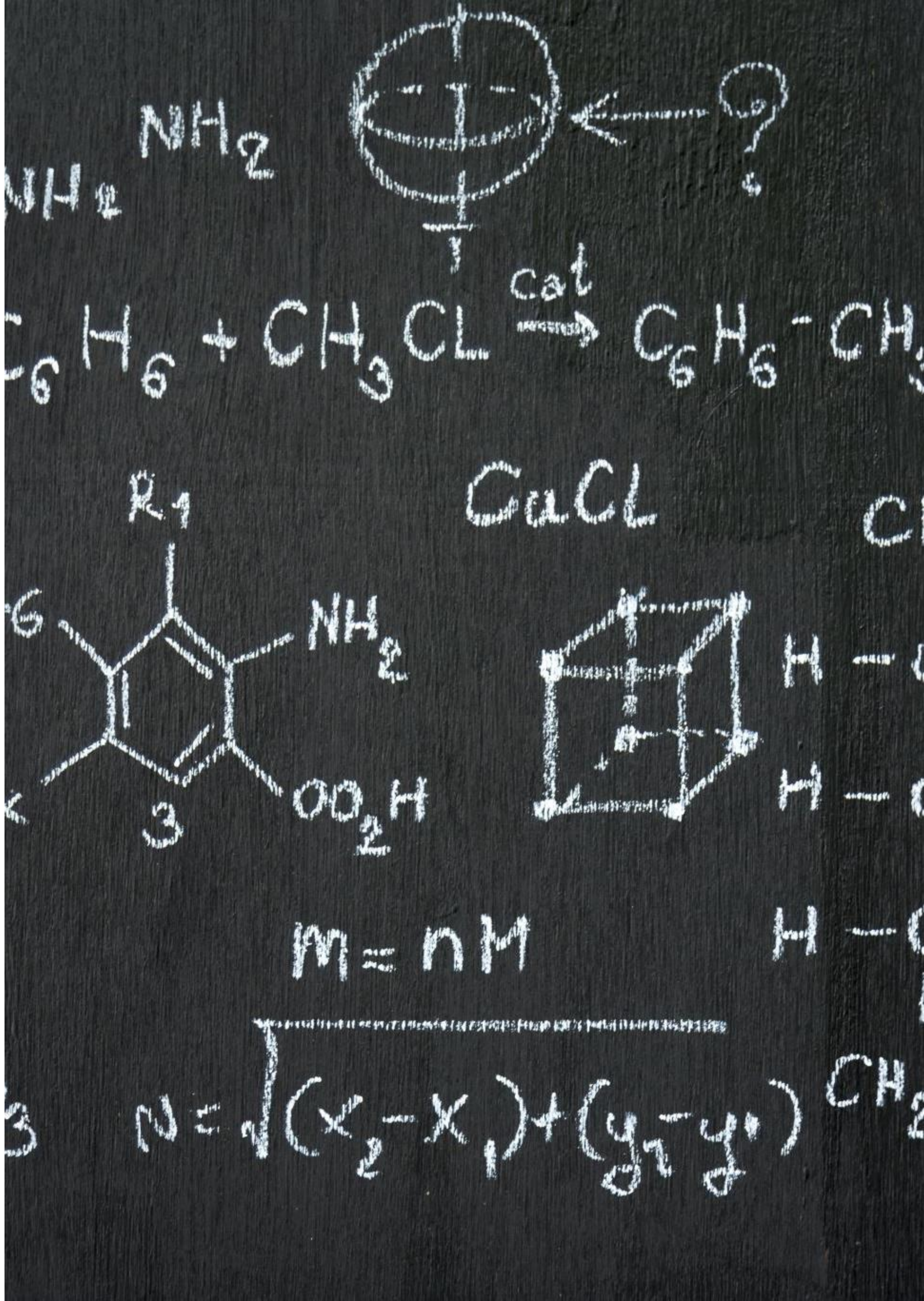
JIT inlining reduces call overhead for small methods, but it is not tail-call elimination and does not mitigate recursion depth.

For inputs that may induce large depth (trees, graphs, QuickSort), use an explicit stack/queue or recurse only on the smaller partition to keep stack usage bounded (typically $O(\log n)$).

ASTANA IT UNIVERSITY

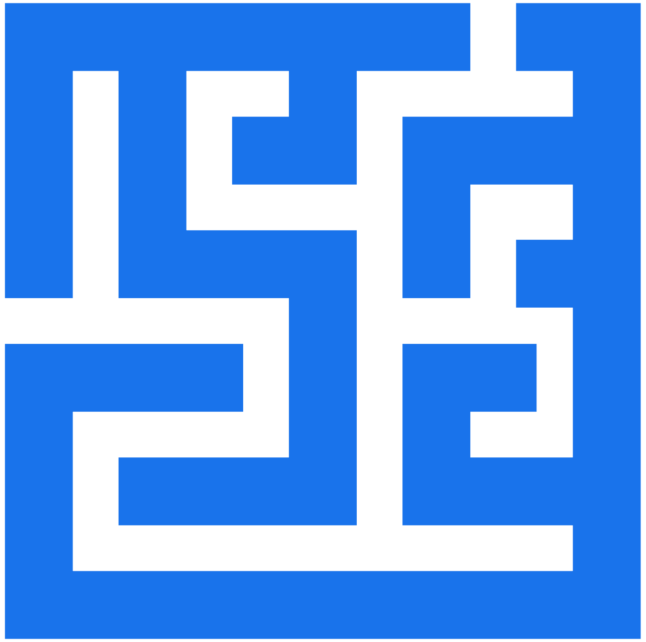
Recursion and Recurrence

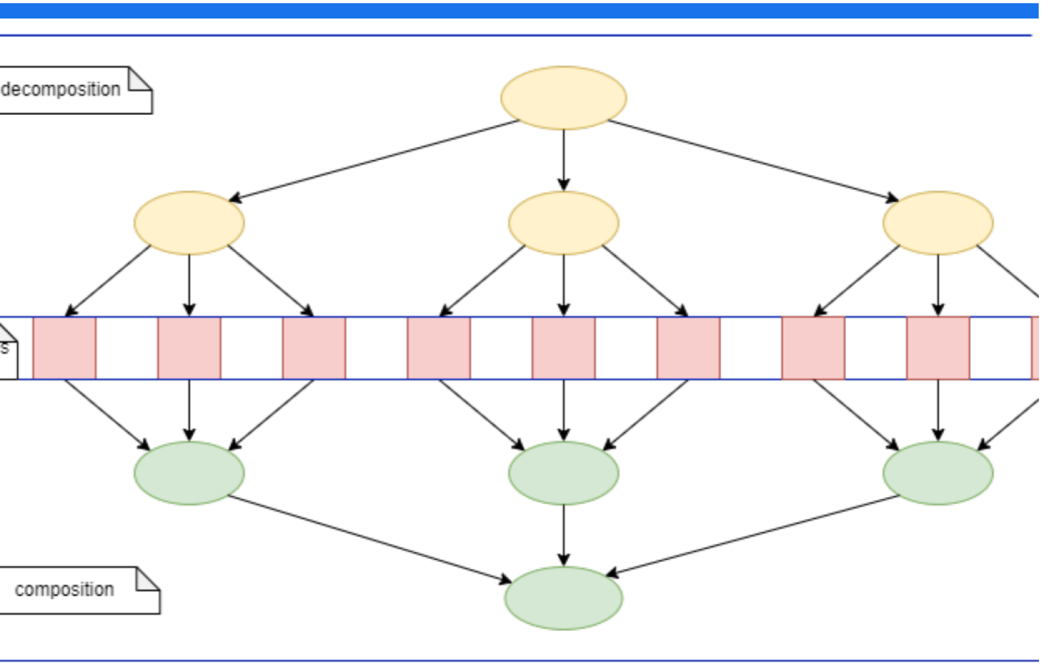
- **Recursion:** a procedure or function defined in terms of itself on smaller or simpler inputs, together with base cases that terminate the process.
- **Recurrence:** an equation that defines a sequence or function by referencing its own earlier values, plus initial conditions.
- **Recurrences for running time:** mathematical models of the cost of recursive algorithms.
- **Relationship:** recursion is a programming technique; a recurrence is a mathematical description (often of that program's behavior).



Tail-Call Elimination

- **When it applies.** Single recursive call in tail position; no work after the call; clear base cases.
- **Idea.** Lift the evolving state into parameters
- **Transform.** Pick accumulator → set initial value → replace the tail call with parameter updates → use a while guarded by the base condition → return the accumulator.
- **Soundness.** State a loop invariant and prove it by induction.
- **Cost model**. Same asymptotics; stack usage becomes constant; fewer call boundaries.
- **Where it shines.** factorial, running sums/scans, Euclid's gcd, linear traversals, simple linear recurrences.
- **Where it doesn't.** Branching recursion or any work after the call (e.g., combine left/right) → use D&C, an explicit stack, or DP.
- **JVM motive.** HotSpot does not promise tail-call optimization; loops are the portable, robust form.
- **Caveats.** Accumulator overflow, accidental boxing/allocations on hot paths, off-by-one bases.





Divide-and-Conquer Patterns

Idea: split problem → solve subproblems → combine; explicit **base cases**.

Decompositions:

- Balanced binary: $n \to (n/2, n/2)$
- Multiway: a subproblems of size n/b
- Uneven: $(n/3, 2n/3)$ **"smaller–larger"** (Example: QuickSort)
- Transform-and-conquer: reduce/transform, then solve (Example: Karatsuba, FFT)

Combine step: cost $f(n)$, example: merge/sum/min/max/partition; aim to keep $f(n)$ linear or constant.

Recurrence templates (cheat-sheet):

- $T(n)=2T(n/2)+n \Rightarrow \Theta(n\log n)$
- $T(n)=T(n/3)+T(2n/3)+n \Rightarrow \Theta(n\log n)$
- $T(n)=aT(n/b)+n^k$

Depth and space: recursion depth ≈ $\log_b n$ (balanced) or larger if uneven; watch stack usage.

Cut-offs: for small nnn switch to a simple routine (Example: insertion sort) to reduce overhead.

Memory model (JVM): prefer in-place where possible; avoid per-level allocations; reuse buffers.

Parallelism: independent subproblems → parallel tasks; span ≈ recursion depth.

Robustness: for skewed splits, recurse on the **smaller** part or iterate the larger.

Recurrence Trees: Intuition & Practice

Model. Represent the recurrence as a rooted tree: the root is the original instance; children are subproblems produced by the divide step; leaves are base cases.

Per-level accounting. For each level, consider (i) how many subproblems appear and (ii) the work performed by each; assess the total work per level.

Regimes of contribution.

- *Top-dominated:* per-level work decreases with depth; upper levels determine the total.
- *Balanced:* per-level work remains approximately constant; total scales with the number of levels.
- *Bottom-dominated:* per-level work increases with depth; leaves determine the total.

Procedure.

1. Unroll several levels of the tree.
2. Record the growth of node counts and per-node work across levels.
3. Classify the regime (top-dominated / balanced / bottom-dominated).
4. Estimate the number of levels (e.g., logarithmic under balanced binary splits; linear under unit decreases).
5. Aggregate per-level contributions and include leaf costs when non-trivial.

Practice notes.

- For uneven splits, analyze branches separately and sum their per-level contributions.
- Account explicitly for the combine step at each node.
- Employ small-input cut-offs to reduce constant-factor overheads.

Master Theorem

A classification rule to estimate running time of divide-and-conquer algorithms with **equal-size subproblems** and an extra **combine** step.

Each call splits into the same number of subproblems of the **same size** (e.g., halves); the combine cost is "smooth" (no wild oscillations). Floors/ceilings don't change the asymptotics.

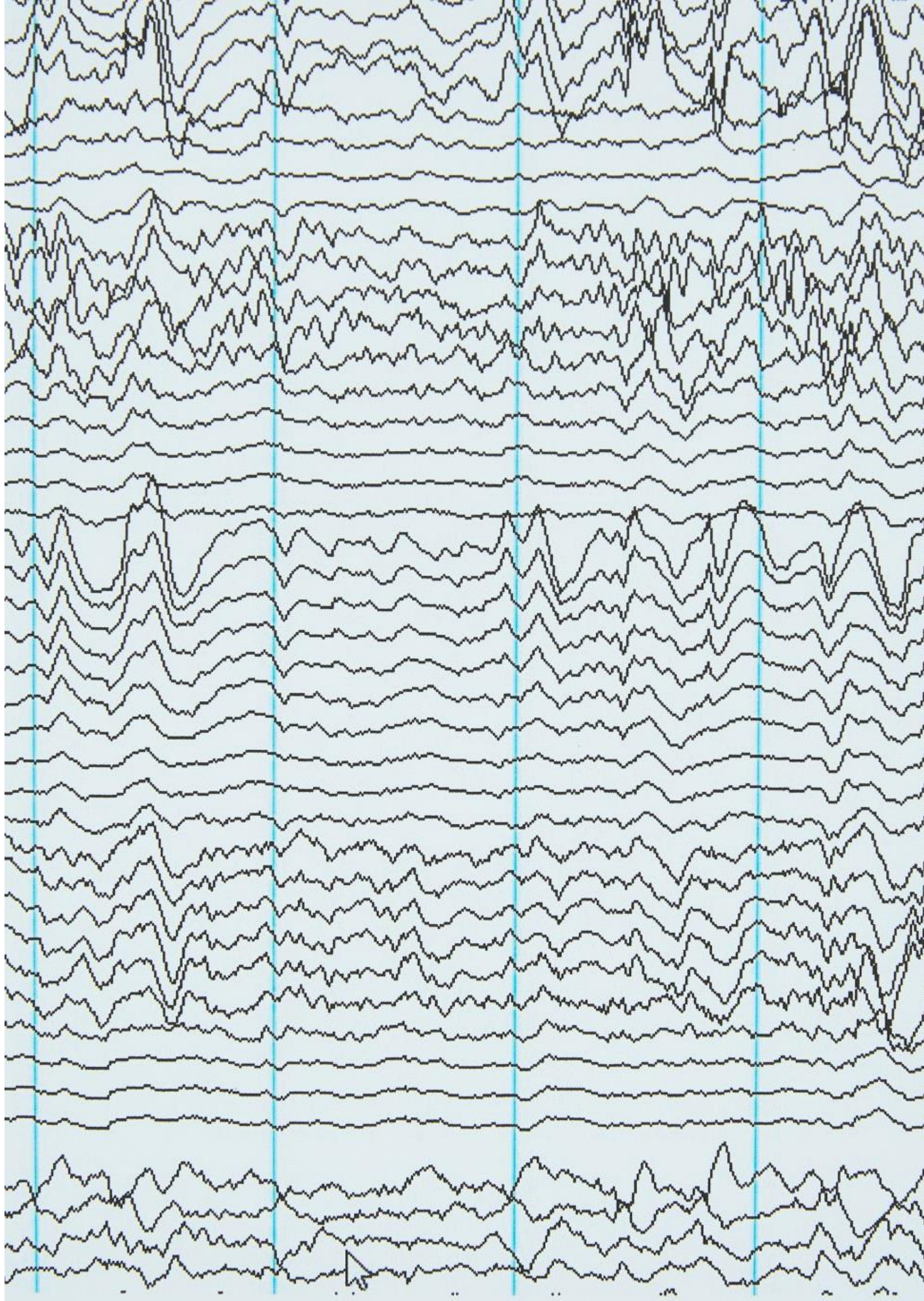
Compare the total work **done by all subproblems** at a level with the **combine work** at that level.

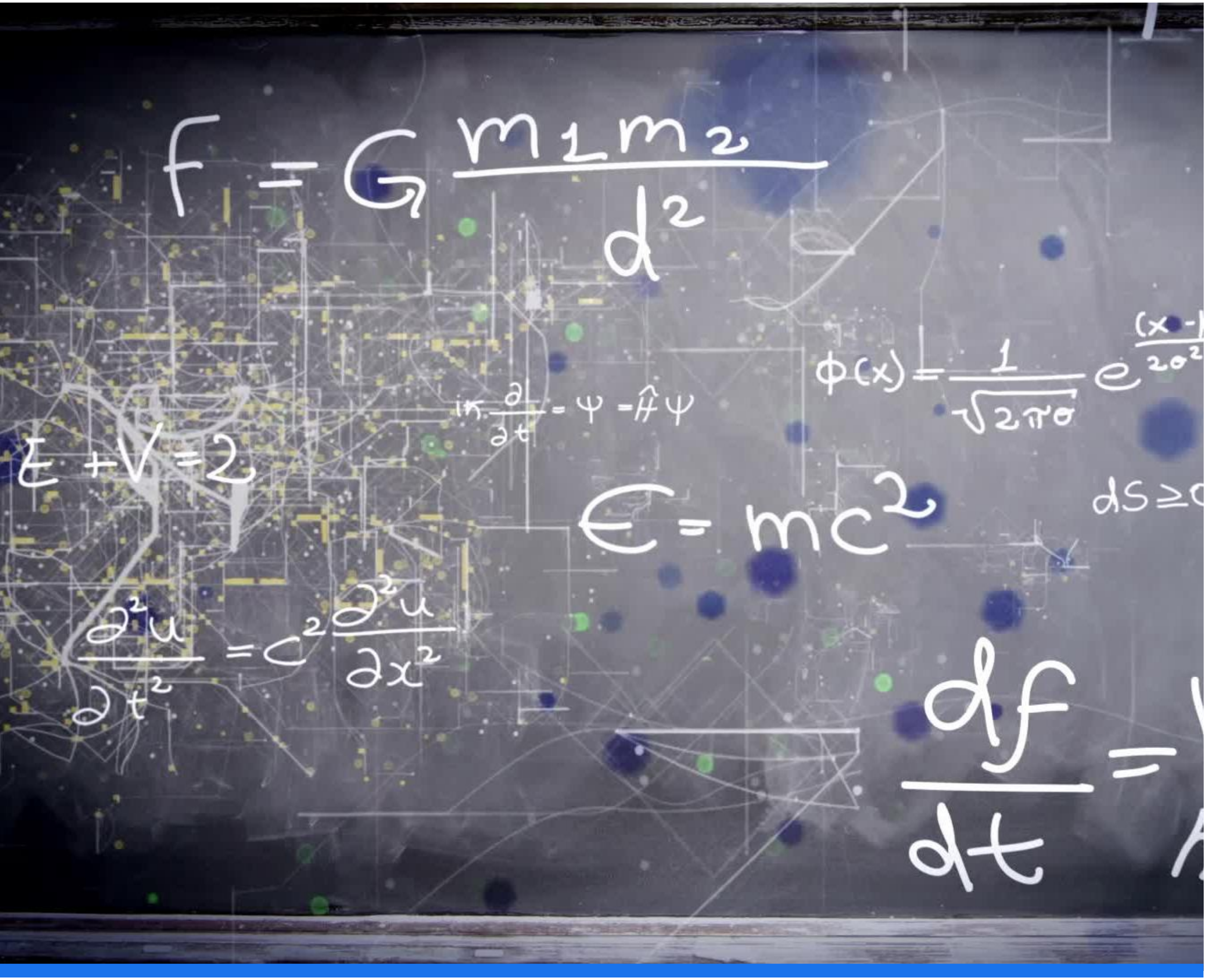
Three outcomes.

1. **Recursive-dominated:** subproblem work is larger → total follows the recursive branching (e.g., four half-size subcalls with linear combine → **quadratic time**).
2. **Balanced:** subproblem and combine work are on the same scale → same growth **with an extra logarithmic factor** (classic **merge sort** → **n log n**).
3. **Combine-dominated:** combine work is larger (and regularity holds) → total **matches the combine work** (e.g., two half-size subcalls with **quadratic** combine → **quadratic** overall).

Examples

- Merge sort: split into two equal halves; combine by linear merge → **n log n**.
- Two equal halves + quadratic combine → **quadratic** time.
- Four equal halves + linear combine → **quadratic** time (recursive part dominates).





Beyond Master: Akra–Bazzi

A general tool to solve running-time recurrences for divide-and-conquer algorithms when subproblems are of **unequal sizes** and/or the combine cost is not a simple polynomial that fits the Master Theorem neatly.

Master Theorem assumes the same subproblem size in each branch (e.g., n/2 and n/2). Many real recurrences are asymmetric (e.g., n/3 and 2n/3) or have additive shifts. Akra–Bazzi covers these cases.

It finds a critical exponent p that “balances” the recursion tree so that the total work per level is roughly flat. Then it compares the non-recursive work g(n) (the combine/overhead term) against n^p to decide what dominates the total cost.
• Uneven splits (T(n) = T(n/3) + T(2n/3) + …).
• Multiple different fractions (T(n) = T(n/2) + T(n/3) + …).
• Combine terms that are awkward for Master (e.g., n / log n, n log n, etc.).
• Small additive shifts in subproblem sizes (like T(⌊n/2⌋) or T(n/2 + O(1))).

1.Write the recurrence in the Akra–Bazzi form (sum of a_i * T(b_i * n + h_i(n)) + g(n)).

2.Solve for p from the balance equation: sum a_i * b_i^p = 1.

3.Compare g(n) with n^p: smaller → T(n) ≈ n^p; similar (up to logs) → add a log factor; larger → T(n) ≈ g(n) (under a regularity condition).

4.State the final Θ-bound.

•How it differs from Master:
• Handles **unequal** subproblem sizes.
• Allows small additive shifts h_i(n).
• Uses a balancing exponent p instead of the single threshold n^{log_b a} from Master.

Thank you