S. KIMMEL

Tree Method (Master Method)

Can be used to solve recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d)$$
$$T(n) = O(1) \quad \text{for } n < C$$

a, b, d don't depend on n

Q: If $T(n)$ is runtime of an algorithm,

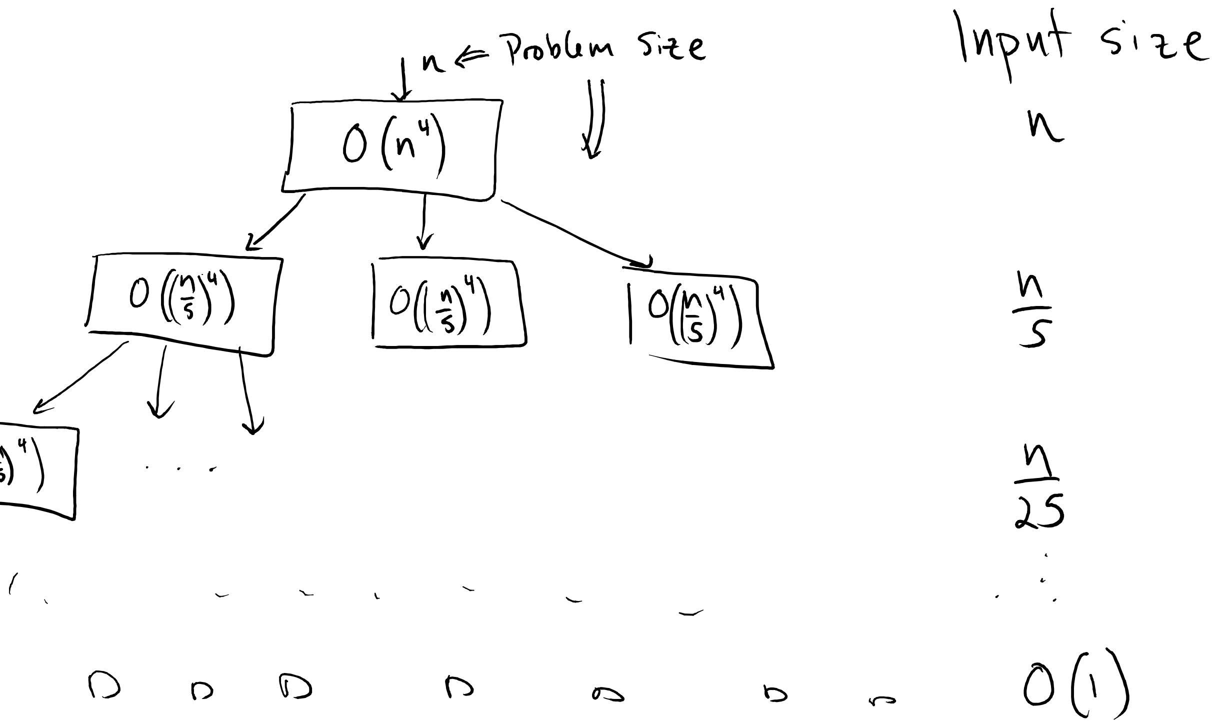
what are a, b, d in words?

A: a: # of recursive calls

b: factor by which problem shrinks in recursive call

d: characterizes extra work outside recursive call

ex: $a = 3$
$b = 5$
$d = 4$

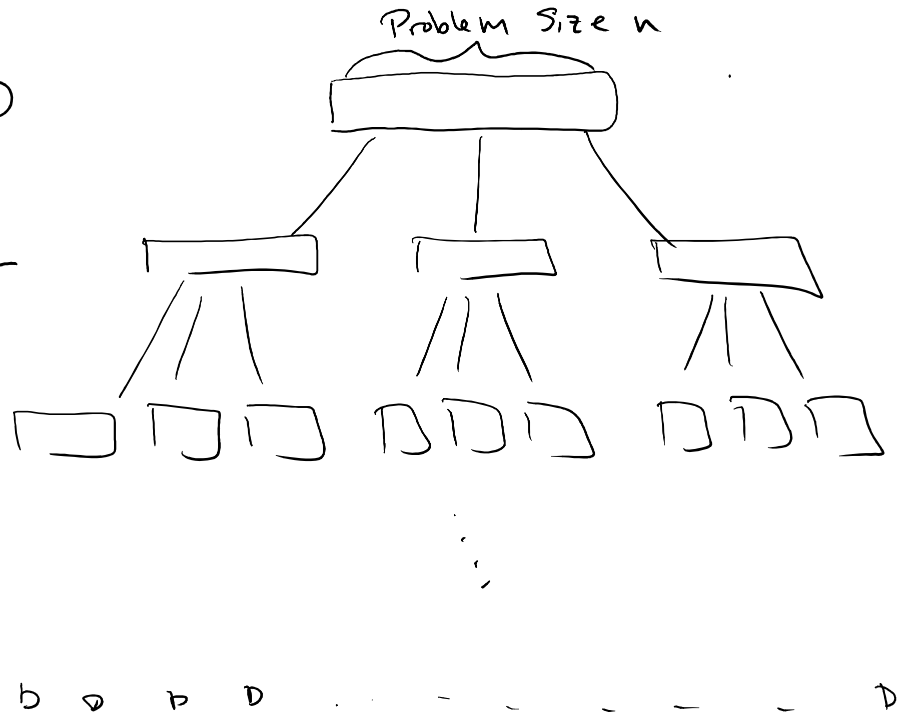


S.KIMMEL

Proof of Tree Method

Problem size

level 0 — n



level 1 — $\frac{n}{b}$

level 2 — $\frac{n}{b^2}$

$\vdots$

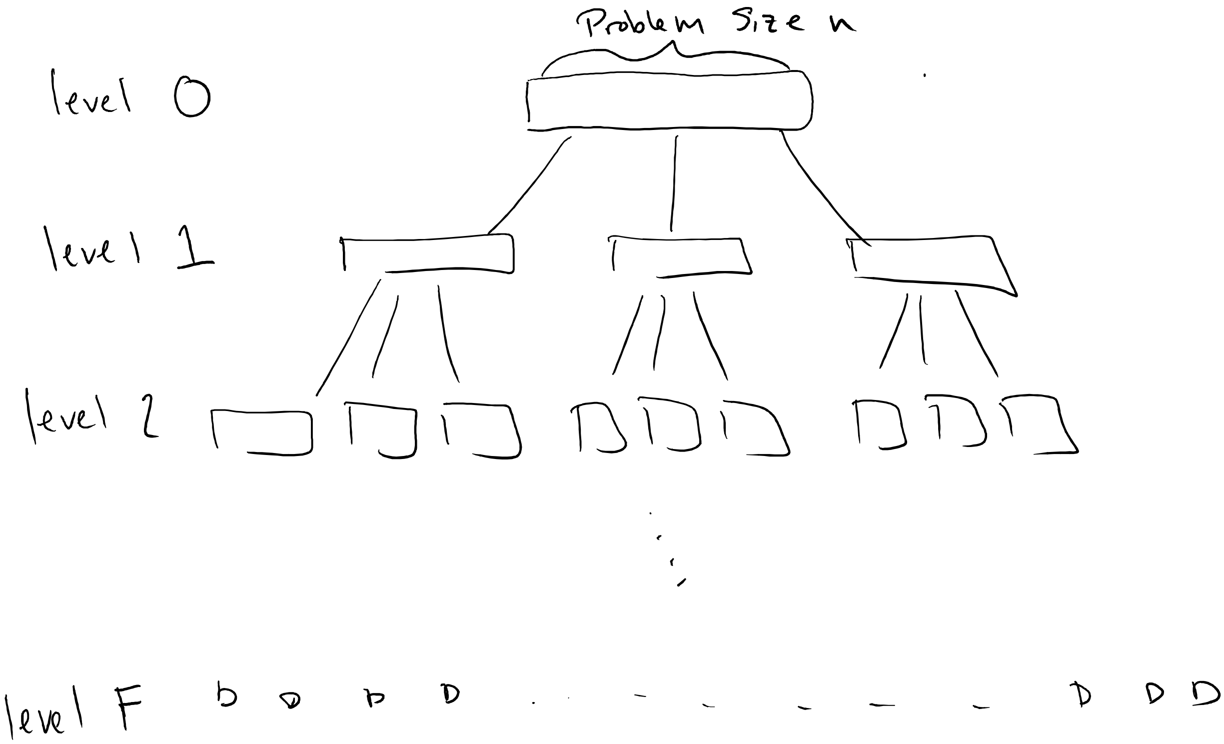
level F — constant $= \frac{n}{b^F}$

Q. What is F (in terms of a, b, d)?

A) $O(\log_b n)$ B) $O(\log_d n)$ C) $O(n^{\log_b d})$ D) $O(b^{\log_d n})$

S.KIMMEL

Proof of Tree Method



Problem size

n

$\frac{n}{b}$

$\frac{n}{b^2}$

$\vdots$

constant $= \frac{n}{b^F}$

Q. What is F (in terms of a, b, d)?

A) $O(\log_b n)$ B) $O(\log_d n)$ C) $O(n^{\log_b d})$ D) $O(b^{\log_d n})$

⇑

Because at each level, problem size is divided by b. $\log_b n$ is number of times n can be divided by b before reaching a constant.

$c = \frac{n}{b^F}$, so $b^F = \frac{n}{c}$, so $F = \log_b\left(\frac{n}{c}\right)$

$= \log_b n - \log_b c$ (constant)

$= O(\log_b n)$

S.KIMMEL

Q. What is the ~~ttl~~ total work (operations) done just at level k, not at other levels?

- $\boxed{a^k}$ subproblems at level k.
- level k subproblem size: $\boxed{\frac{n}{b^k}}$
- Work outside of recursive call required to solve 1 subproblem : $\boxed{\left(\frac{n}{b^k}\right)^d}$

$\Rightarrow$ Total work $\quad a^k\left(\frac{n}{b^k}\right)^d = \boxed{\left(\frac{a}{b^d}\right)^k n^d}$

Now we add up work done at all levels:

$$\sum_{k=0}^{\log_b n}\left(\frac{a}{b^d}\right)^k n^d$$

$$T(n) = n^d\left(\sum_{k=0}^{\log_b n}\left(\frac{a}{b^d}\right)^k\right)$$

Mutliplicative Distributive property (factor out n^d)

Geometric Series:

$$\sum_{k=0}^{F} r^k = \begin{cases} F+1 & \text{if } r = 1 \\ \dfrac{1-r^{F+1}}{1-r} & \text{otherwise} \end{cases}$$

Geometric Series:

$$\sum_{k=0}^{F} r^k = \begin{cases} F+1 & \text{if } r = 1 \\ \dfrac{1-r^{F+1}}{1-r} & \text{otherwise} \end{cases}$$

S. KIMMEL

PSET:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

↑

This is usually called "master method"
"master theorem"

Master has pretty unpleasant connotations. Also it is not descriptive

My term: "Tree method"

ex: Binary Search:

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

$T(1) = O(1)$

$a = 1$
$b = 2$
$d = 0$

$\Rightarrow \boxed{a = b^d}$

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

$= O(n^0 lo$

S. KIMMEL

We'll do one of the 3 cases here:

Case: $a < b^d$

$$T(n) = n^d \left(\sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k \right) = n^d \left(\frac{\boxed{1 - \left(\frac{a}{b^d}\right)^{\log_b n + 1}}}{\boxed{1 - \left(\frac{a}{b^d}\right)}} \right)$$

constant. Can ignore for big-O

$1 - r^{\log_b n + 1}$ where $r = \frac{a}{b^d} < 1$

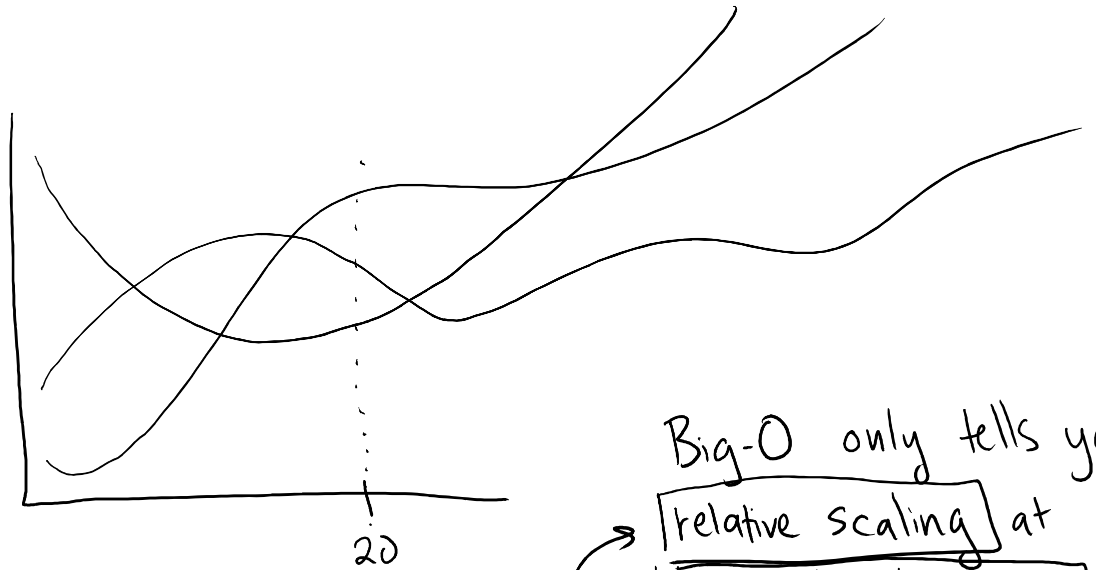
So as n gets big $r^{\log_b n+1} \to 0$

ex: $\left(\frac{1}{3}\right)^{10} = \frac{1}{3^{10}} \approx 0$

So $1 - r^{\log_b n+1} \longrightarrow 1$ for large n.

$= O(n^d)$

S.KIMMEL



Big-O only tells you about $\boxed{\text{relative scaling}}$ at $\boxed{\text{large input sizes}}$

C

K

Even if know C, K. Still don't know anything.
Infinitely many C, K work! One pair is not useful.

To know how algorithm will do on specific input, need to know actual time complexity function. Not just big-O or big-Θ, or big-Ω.