Algorithms (CS-204)

performance analysis

performance analysis

- Why to worry about performance?
- *Given two algorithms for a task, how do we find out which one is better?*
- *Naïve way is to run both the algorithms with a set of inputs*

- Depends on input
  - It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
- Depends on machine architecture
  - It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.
- What is the alternative?

- Doing Asymptotic Analysis rather than comparing actual time of execution in a machine.

- Asymptotic analysis of an algorithm refers to defining the mathematical framing of its run-time performance. Here, we evaluate the performance of an algorithm in terms of input size. We calculate, how does the time (or space) taken by an algorithm changes with the input size.

Worst Case Analysis (Usually Done)

- In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.
  - For Linear Search, the worst case happens when the element to be searched is not present in the array. When specific element is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(n).

Average Case Analysis (Sometimes done)

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.
  - For the linear search problem, let us assume that all cases are uniformly distributed (including the case of element not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

    $(1+2+3+\ldots+n+n)/(n+1) = O(n)$

Best Case Analysis(Rarely done)

- In best case analysis, we take the best possible input and calculate computing time for that input.
  - For the linear search problem, best case will be the case when the item we are looking for is found in the very first element of the array. It will take constant time irrespective of the input size. O(1)

```
INSERTION-SORT(A)
1 for j = 2 to A.length
2       key = A[j]
3       // Insert A[j]  into the sorted sequence A[1.. j-1]
4       i = j - 1
5       while i > 0 and A[i] > key
6                   A[i + 1] = A[i]
7                   i = i - 1
8       A[i + 1] = key
```

		cost	times
INSERTION-SORT(A)			
1	**for** j = 2 **to** A.*length*	*c1*	*n*
2	*key* = A[j]	c2	n-1
3	// Insert A[j] into the sorted sequence A[1.. j-1]		
4	i = j – 1	c3	n-1
5	**while** i > 0 and A[i] > *key*	*c4*	$\sum_{j=2}^{n} t_j$
6	A[i + 1] = A[i]	*c5*	$\sum_{j=2}^{n}(t_j\text{-}1)$
7	i = i – 1	*c6*	$\sum_{j=2}^{n}(tj\text{-}1)$
8	A[i + 1] = *key*	*c7*	*n-1*

Best Case

- T (n) = c1n + c2(n – 1) + c3(n – 1) + c4$\sum_{j=2}^{n} t_j$ + c5 $\sum_{j=2}^{n}(t_j$-1) + c6 $\sum_{j=2}^{n}(tj$-1) + c7(n-1)
- For best case t_j = 1 for all possible values of j.
  - T(n) = c1n + c2(n – 1) + c3(n – 1) + c4(n – 1) + c7(n – 1)
  - T(n)= (c1+c2+c3+c4+c7)n +(-c2-c3-c4-c7)
  - T(n)=an+b

Worst Case

- For worst case t_j=j for all possible values of j
- $T(n)= an^2+bn+c$

Average Case

- The “average case” is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in sub-array A[1.. j – 1] to insert element A[j]. On average, half the elements in A[1.. j – 1] are less than A[j] , and half the elements are greater. On average, therefore, we check half of the sub-array A[1.. j – 1], and so t_j is about j/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Order of growth

- Abstraction for ease of analysis
  - ignored the actual cost of each statement, using the constants c_i
  - Then we represent cost in terms of an^2+bn+c where a, b and c are constants that depend on constants c_i
  - We can further simplify the abstraction ignoring all other terms except leading term an^2
  - We can also ignore constant term (a) as it is less significant compared to the term n^2

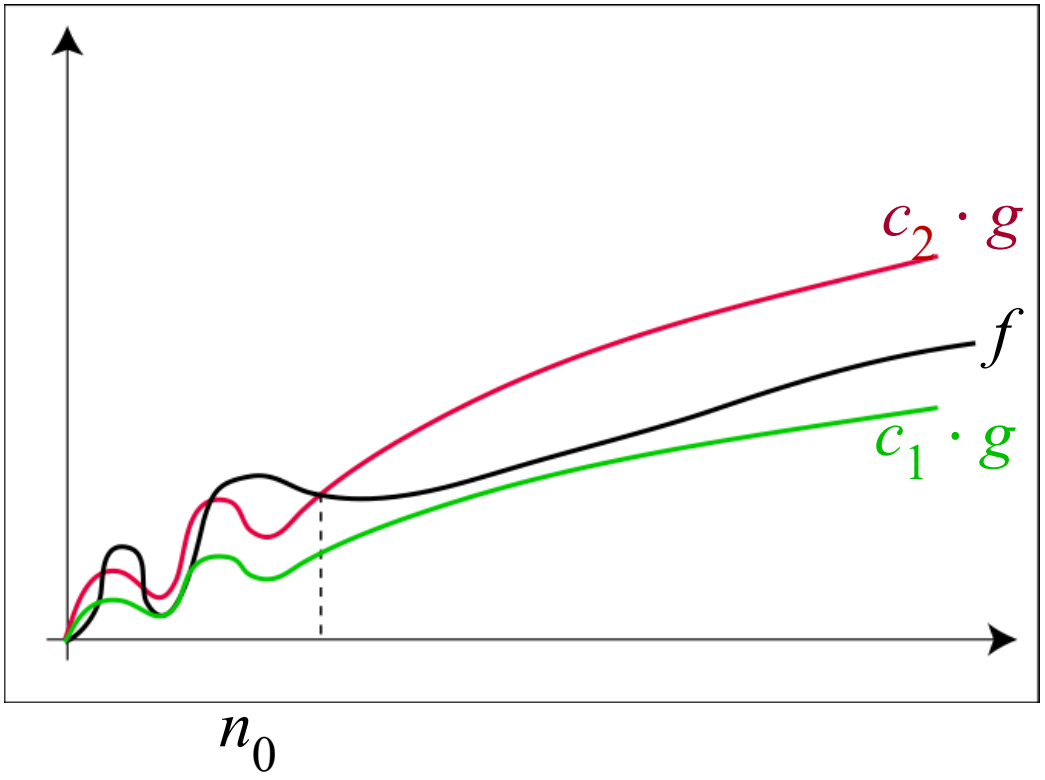
Asymptotic Analysis

- we are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases.

- An algorithm will be asymptotically more efficient if it’s growth is less compared to other algorithm with increase of input size.

The Θ-Notation

$$\Theta(g(n)) = \{ f(n) : \exists c_1, c_2 > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0: \\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ \}$$



A function f (n) belongs to the set θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n. In other words, for all n >= n_0, the function f (n) is equal to g(n) to within a constant factor. we usually write f (n) = θ(g(n)) to express the same notion instead of f(n) ∈ θ(g(n)).

Can we show $\frac{1}{2} n^2 - 3n = \Theta(n^2)$?

- Can we show ½ $n^2 - 3n = \Theta(n^2)$

$$c_1 n^2 \le \frac{1}{2} n^2 - 3n \le c_2 n^2$$

for all $n \ge n_0$. Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$

- We can make the right-hand inequality hold for any value of n >=1 by choosing any constant c_2 >=1/2. Likewise, we can make the left-hand inequality hold for any value of n >=7 by choosing any constant c_1 <=1/14. Thus, by choosing c_1 = 1/14, c_2 = 1/2, and n_0 = 7, we can verify that ½ $n^2 - 3n = \Theta(n^2)$

- Show that $3n^3 + 2n^2+5n+3 = \Theta(n^3)$

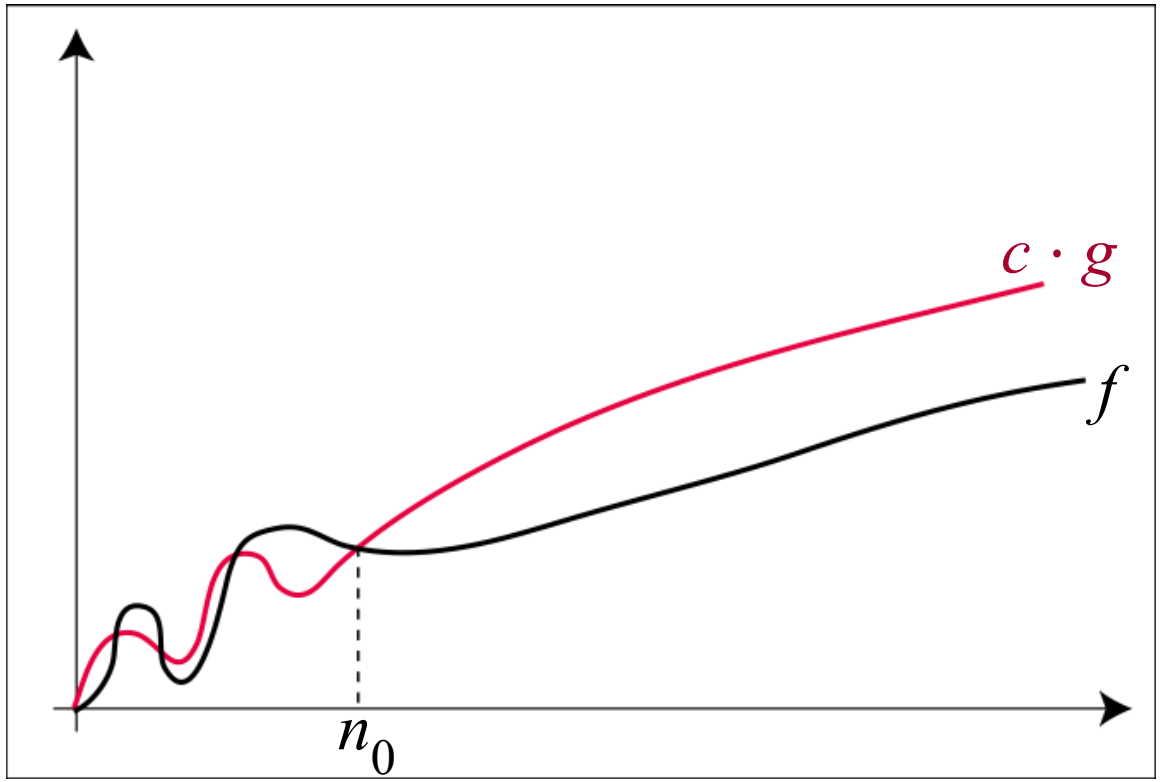
- $C_1n^3 <= 3n^3 + 2n^2+5n+3 <= C_2n^3$
- $C_1 <= 3+2/n+5/n^2+3/n^3 <= C_2$
- If $0< C_1 <=3$ for $C_2 >=4$ it is valid
- Assuming $C_1 =3$ and $C_2 =4$ and $n_0=4$ it is always true

- Also try $4n^2 + 2n +5 != \Theta(n^3)$

The *O*-Notation

$$O(g(n)) = \{ f(n) : \exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \}$$

The ,Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation.



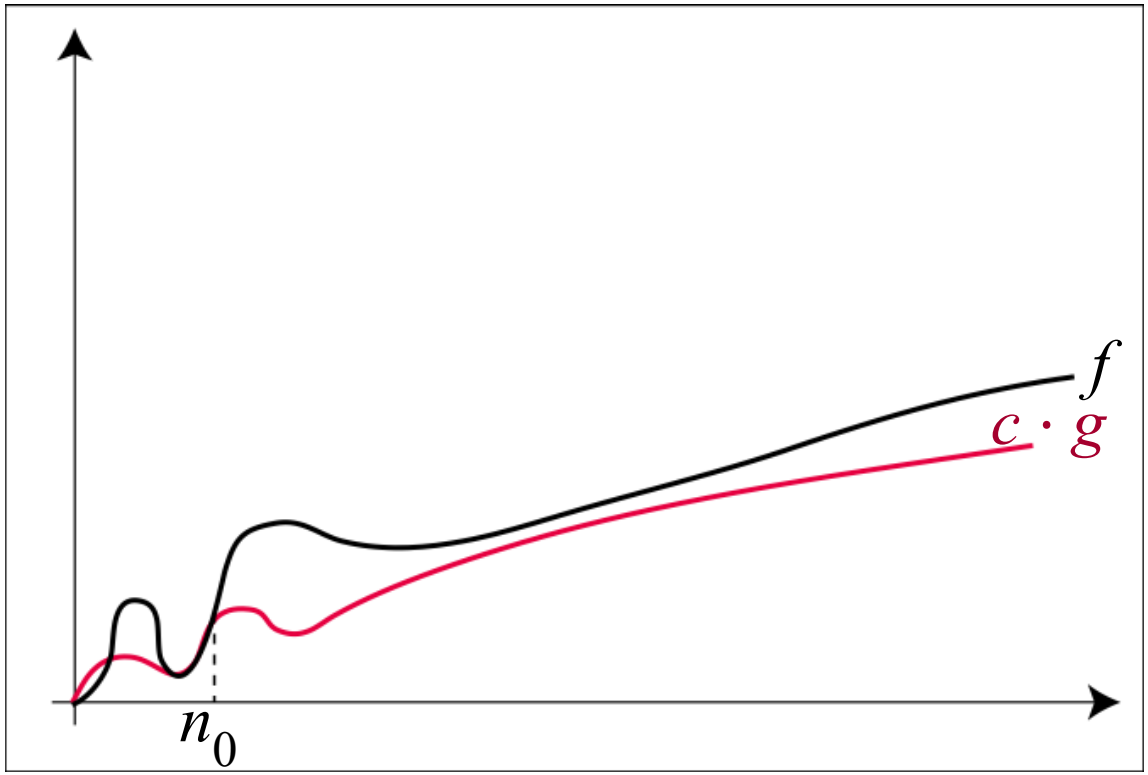
We use O-notation to give an upper bound on a function, to within a constant factor. We write f(n) = O(g(n)) to indicate that a function f (n) is a member of the set O(g(n)).

Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$

We may also write an+b = $O(n^2)$, here the bound is not tight

The Ω-Notation

$$\mathbf{\Omega(g(n))} = \{ f(n) : \exists c > 0, n_0 > 0 \text{ s.t. } \forall n \geq n_0 : f(n) \geq c \cdot g(n) \geq 0 \}$$

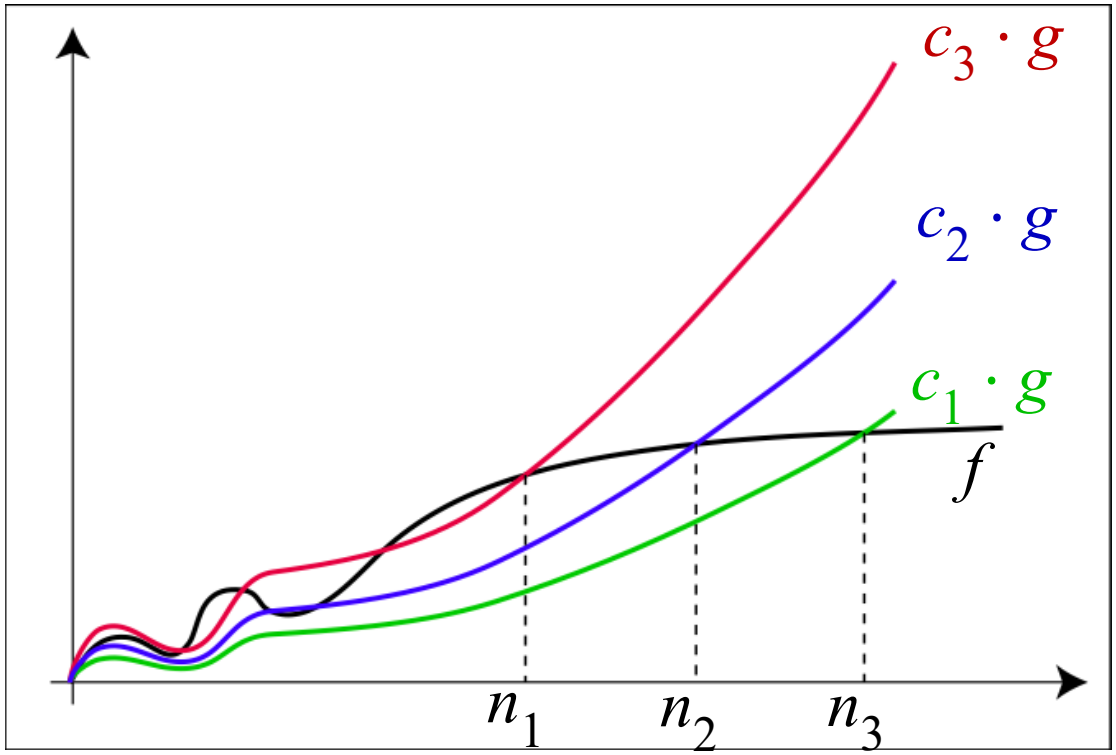


The *o*-Notation

$$o(g(n)) = \{ f(n) : \forall c > 0 \; \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : f(n) < c \cdot g(n) \}$$

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

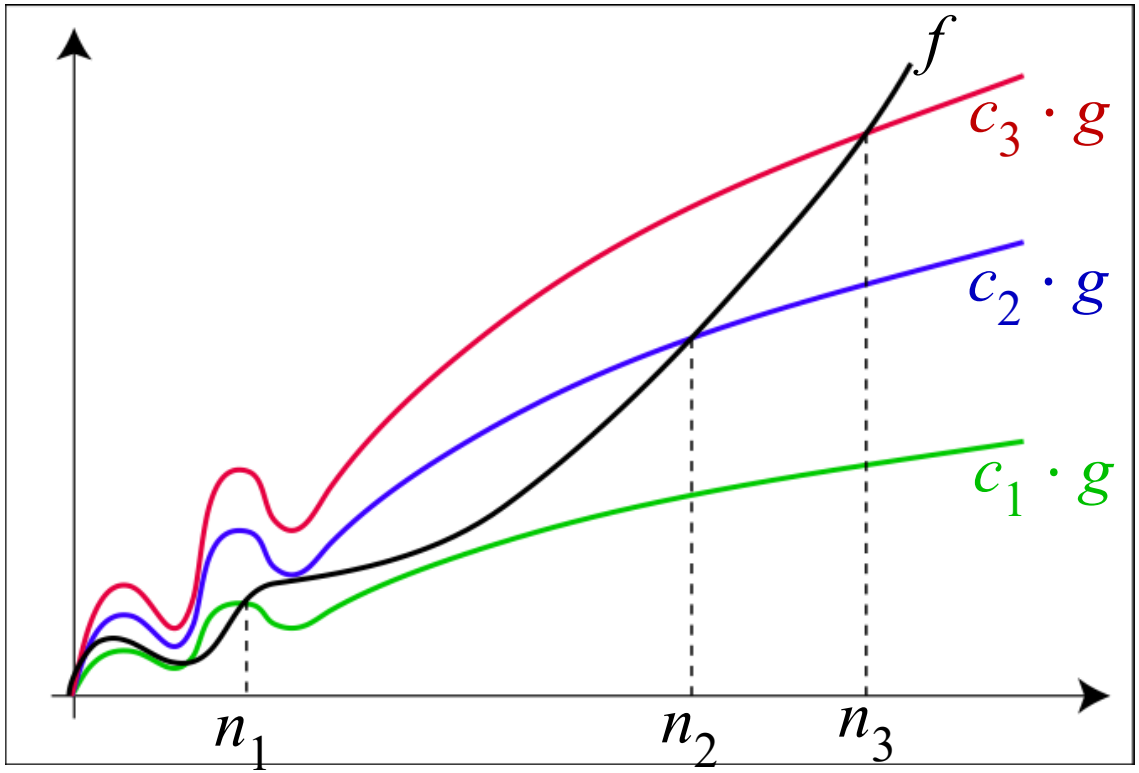
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 .$$



The ω-Notation

$$\omega(g(n)) = \{ f(n) : \forall c > 0 \; \exists n_0 > 0 \text{ s.t. } \forall n \geq n_0 : f(n) > c \cdot g(n) \}$$

For example, $2n^2 = \omega(n)$, but $2n^2 \neq \omega(n^2)$. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$



Asymptotic notation in equations

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that $2n^2 + 3n + 1 = 2n^2 + f(n)$, where f (n) is some function in the set $\Theta(n)$. In this case, we let $f(n) = 3n + 1$, which indeed is in $\Theta(n)$.
- In this way merge sort running time can be expressed as
- $T(n) = 2T(n/2) + \Theta(n)$.

Comparison of Functions

- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$ *Transitivity*
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

- $f(n) = O(f(n))$
  $f(n) = \Omega(f(n))$
  $f(n) = \Theta(f(n))$

Reflexivity

Comparison of Functions

- $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$ *Symmetry*

- $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ *Transpose Symmetry*
- $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

- $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$

Does Asymptotic Analysis always work?

- Asymptotic Analysis is not perfect
- For example, say there are two sorting algorithms that take 1000nlogn and 2nlogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nlogn).

- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Recurrences

- A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- For example $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 , \end{cases}$$

Solving Recurrences

- Substitution method
- Recursion Tree method
- Master method

Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.

Example: $T(n) = 4T(n/2) + 100n$

- [Assume that $T(1) = \Theta(1)$.]
- Guess $O(n^3)$. (Prove O and Ω separately.)
- Assume that $T(k) \le ck^3$ for $k < n$.
- Prove $T(n) \le cn^3$ by induction.

Example of substitution

$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4c(n/2)^3 + 100n \\
&= (c/2)n^3 + 100n \\
&= cn^3 - ((c/2)n^3 - 100n) \qquad \leftarrow \textit{desired} - \textit{residual} \\
&\le cn^3 \qquad \leftarrow \textit{desired}
\end{aligned}
$$

whenever $(c/2)n^3 - 100n \ge 0$, for example, if $c \ge 200$ and $n \ge 1$.

residual

A tighter upper bound?

We shall prove that $T(n) = O(n^2)$.

Assume that $T(k) \le ck^2$ for $k < n$:

$$\begin{aligned} T(n) &= 4T(n/2) + 100n \\ &\le cn^2 + 100n \\ &\le cn^2 \end{aligned}$$

for ***no*** choice of $c > 0$. Lose!

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

- ***Subtract*** a low-order term.

Inductive hypothesis: $T(k) \le c_1 k^2 - c_2 k$ for $k < n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + 100n \\
&\le 4(c_1(n/2)^2 - c_2(n/2)) + 100n \\
&= c_1 n^2 - 2c_2 n + 100n \\
&= c_1 n^2 - c_2 n - (c_2 n - 100n) \\
&\le c_1 n^2 - c_2 n \qquad \text{if } c_2 > 100.
\end{aligned}
$$

Wrong Application of induction

Given recurrence: $T(n) = 2T(n/2) + 1$

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

Claim : ∃ some constant c and n_0 , such that $T(n) <= cn$, ∀ $n >= n_0$.

Proof: Suppose the claim is true for all values <= n/2 then

$T(n)=2T(n/2)+1$ (given)

$<= 2.c.(n/2) +1$ (by induction hypothesis)

$= cn+1$

So, does that mean that the claim we initially made that $T(n)=O(n)$ was wrong ?

Given Recurrence: $T(n) = 2T(n/2) + 1$

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

Claim : ∃ some constant c and n_0 , such that $T(n) <= cn - b$, ∀ $n >= n_0$

Proof: Suppose the claim is true for all values $<= n/2$

then

$$T(n)=2T(n/2)+1$$
$$<=2[c(n/2)-b]+1$$
$$<= cn-2b+1$$
$$<= cn-b+(1-b)$$
$$<= cn-b \text{ , } \forall\ b>=1$$

Thus, $T(n/2) <= cn/2 – b$

$=> T(n) <= cn – b$, ∀ $b>=1$

Hence, by induction $T(n) = O(n)$, i.e. Our claim was true.

Hence proved.

Changing variables

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before.

$T(n) = 2T(\sqrt{n}) + \lg n$
$m=\lg n$
$T(2^m)=2T(2^{m/2})+m$
$S(m)=2S(m/2)+m$
$S(m) = O(m \lg m)$.
$T(n)=T(2^m)=S(m) = O(m \lg m)=O(\lg n \lg \lg n)$

Home Work

Show that the solution of

$T(n) = T(n-1) + n$ is $O(n^2)$.

$T(n) = 4T(n/2) + n^2$ is $T(n) = \Theta(n^2)$.

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method promotes intuition, however.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

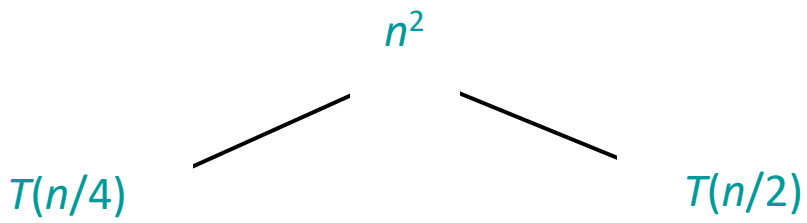
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

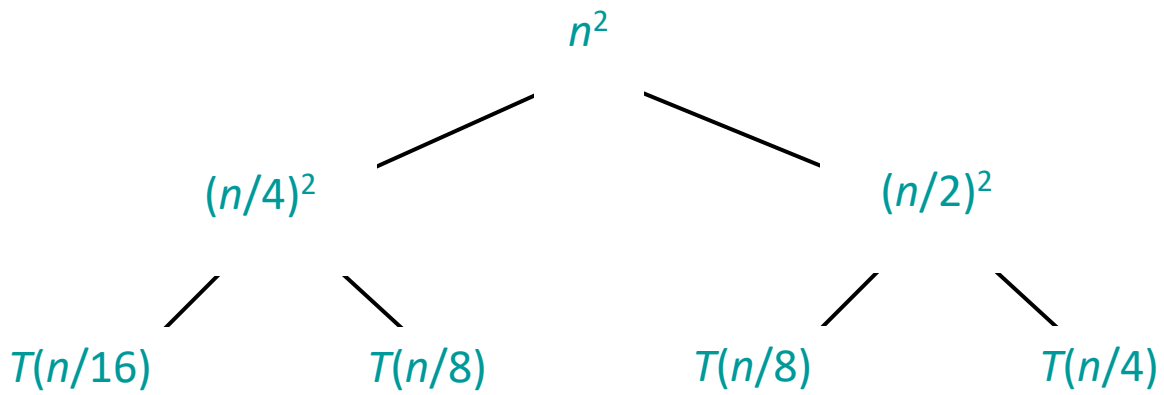
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



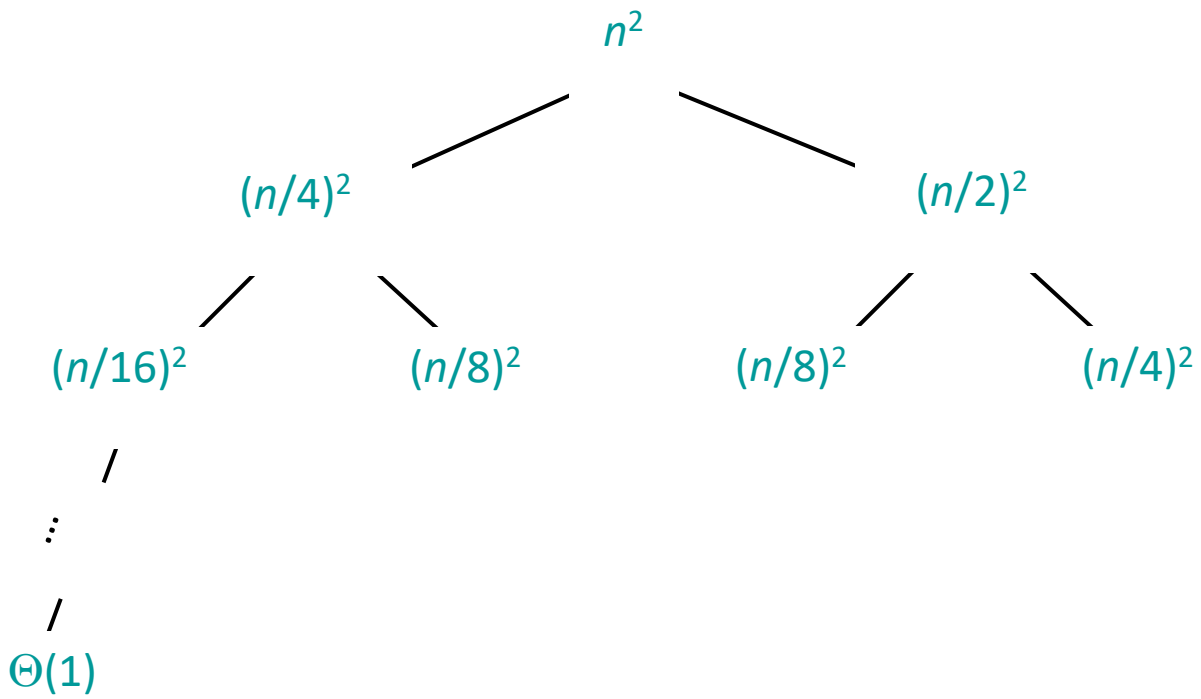
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



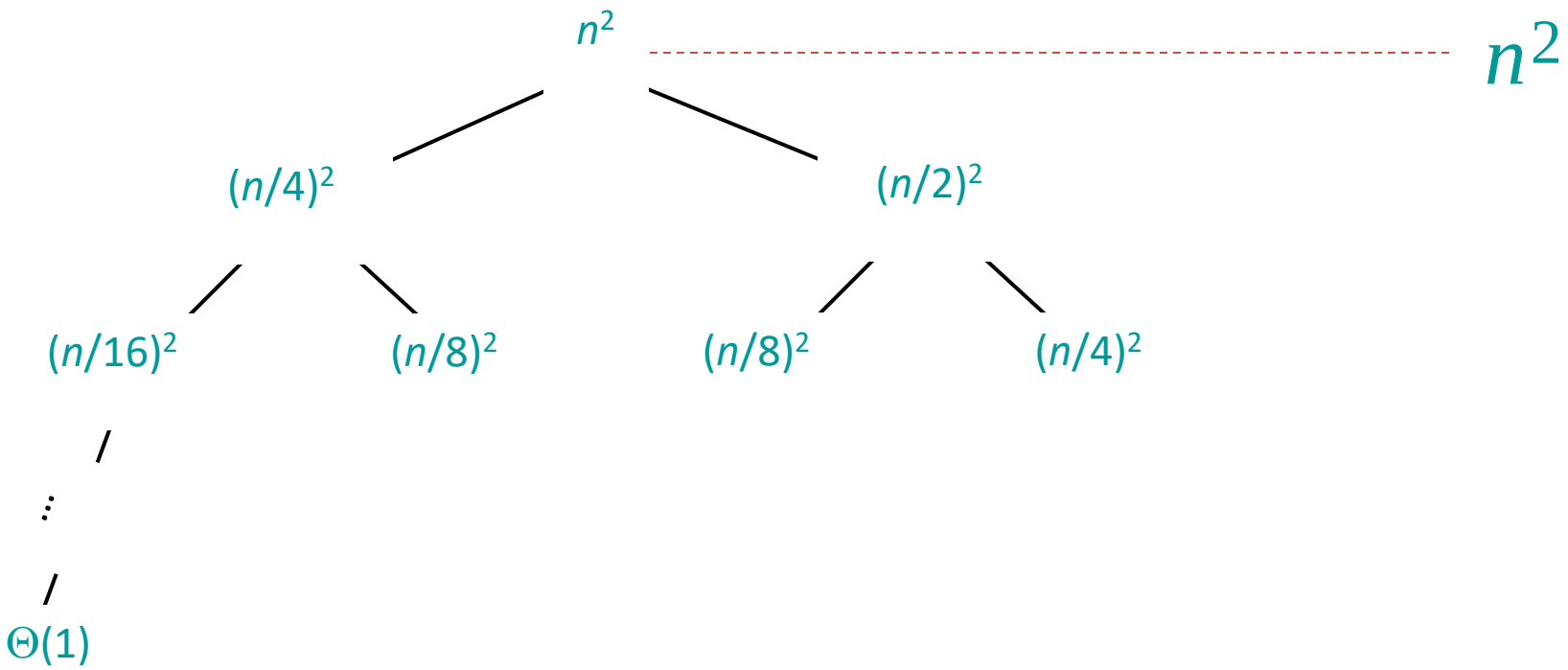
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



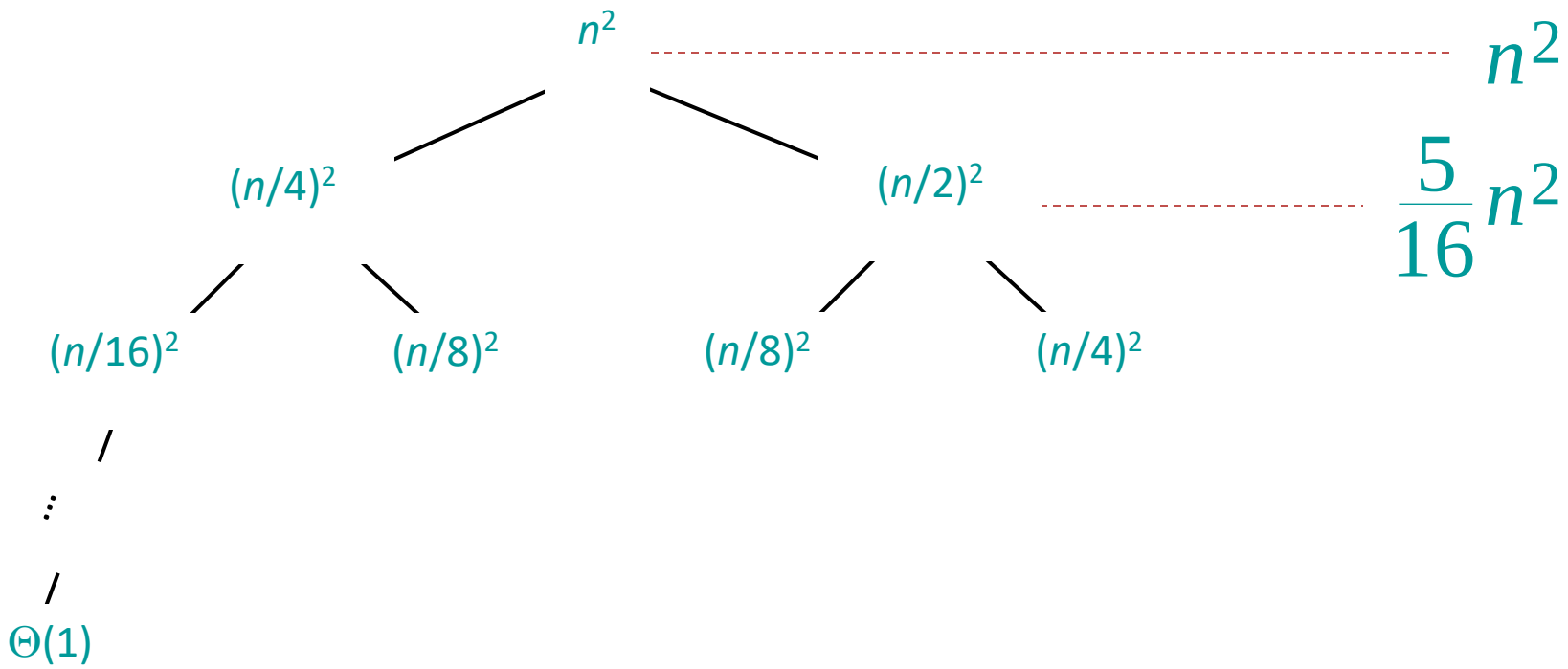
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



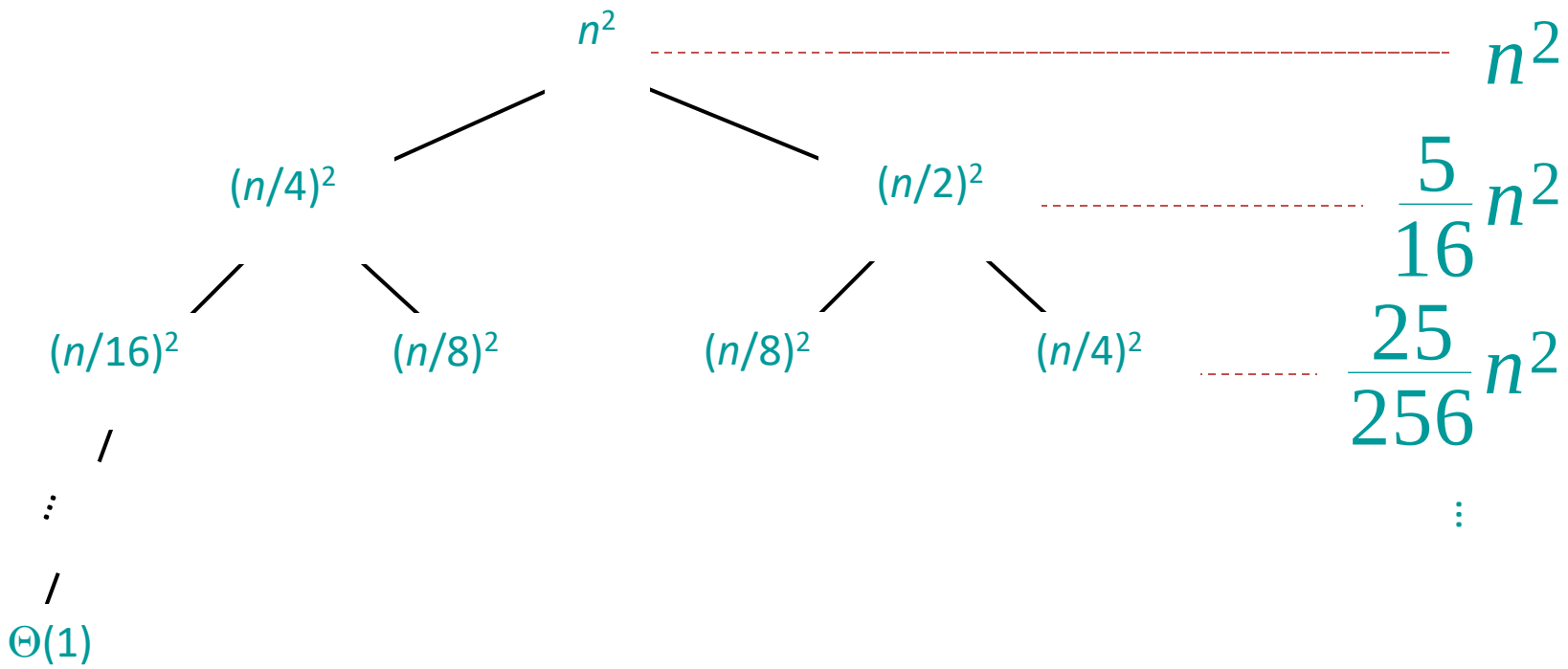
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



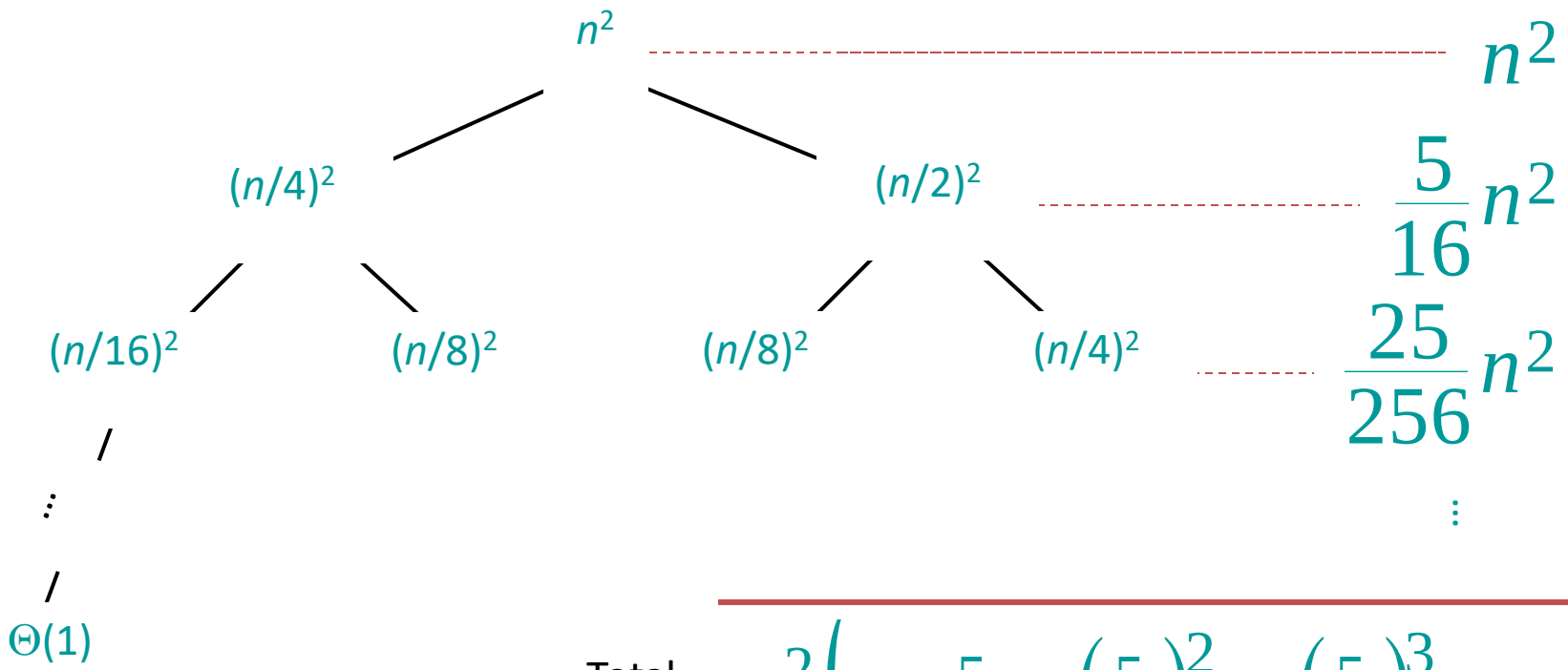
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Total $= n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$

$= \Theta(n^2)$ *geometric series*

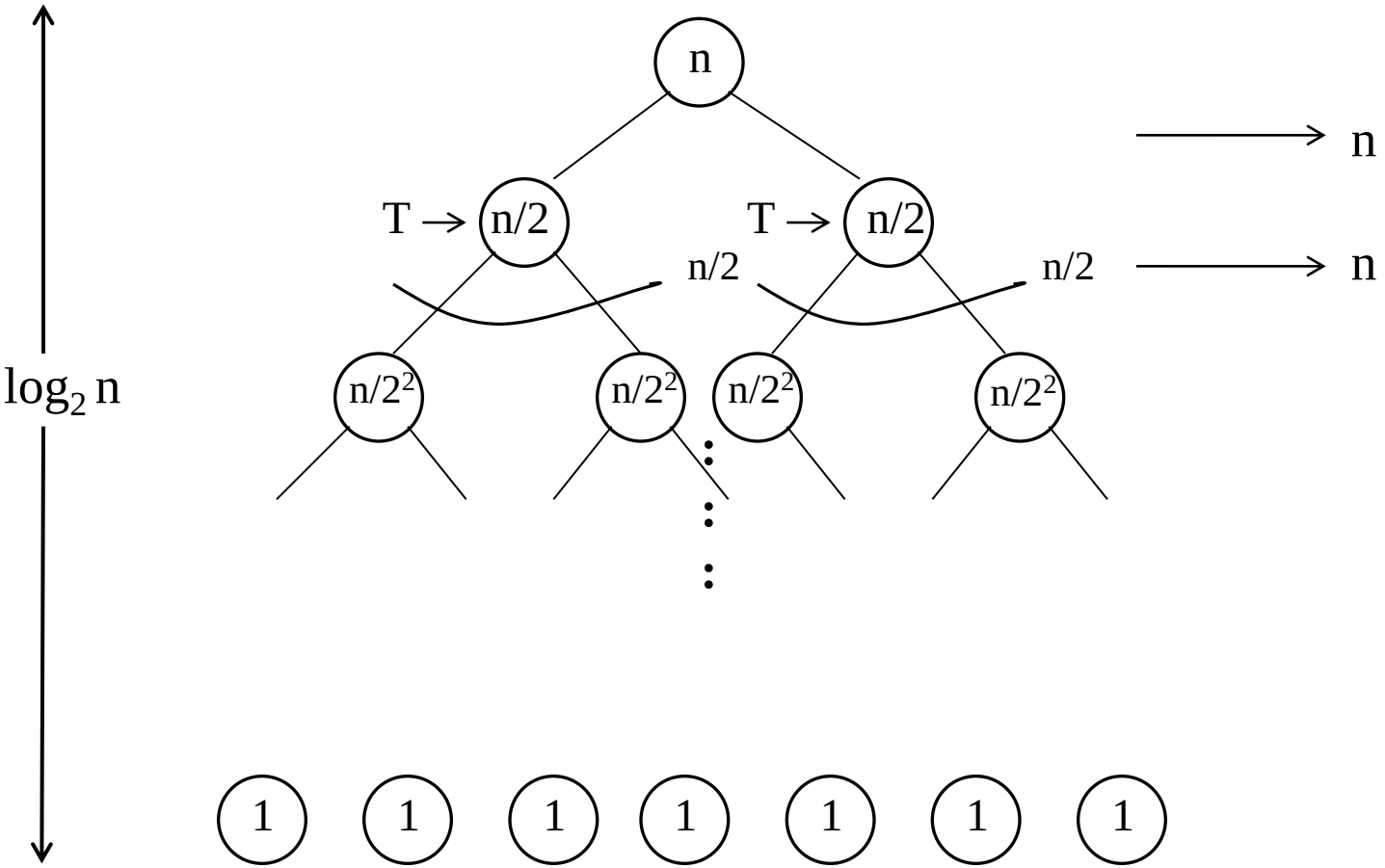
Appendix: geometric series

$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

1) **T(n) = 2T(n/2) + n**

The recursion tree for this recurrence is :



When we add the values across the levels of the recursion tree, we get a value of n for every level.

We have $n + n + n + \ldots\ldots$ log n times

$= n (1 + 1 + 1 + \ldots\ldots$ log n times)

$= n (\log_2 n)$

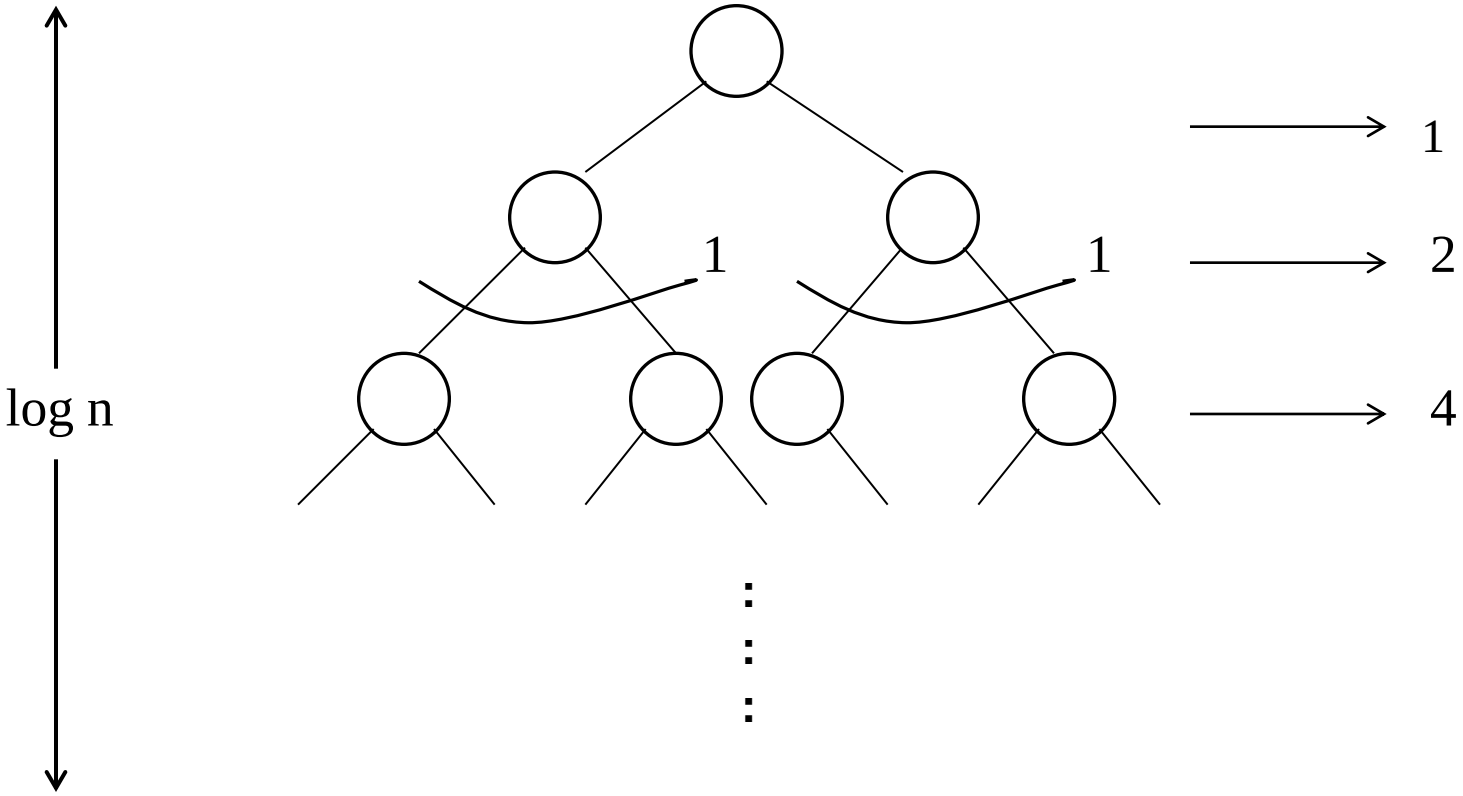
$= \Theta (n \log n)$

$T(n) = \Theta (n \log n)$

II.

Given : **T(n) = 2T(n/2) + 1**

Solution : The recursion tree for the above recurrence is



Now we add up the costs over all levels of the recursion tree, to determine the cost for the entire tree :

We get series like

$1 + 2 + 2^2 + 2^3 + \ldots\ldots$ log n times which is a G.P.

[So, using the formula for sum of terms in a G.P. :

$$a + ar + ar^2 + ar^3 + \ldots\ldots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1} \quad]$$

$$= \frac{1\,(2^{\log n} - 1)}{2 - 1}$$

$$= n - 1$$

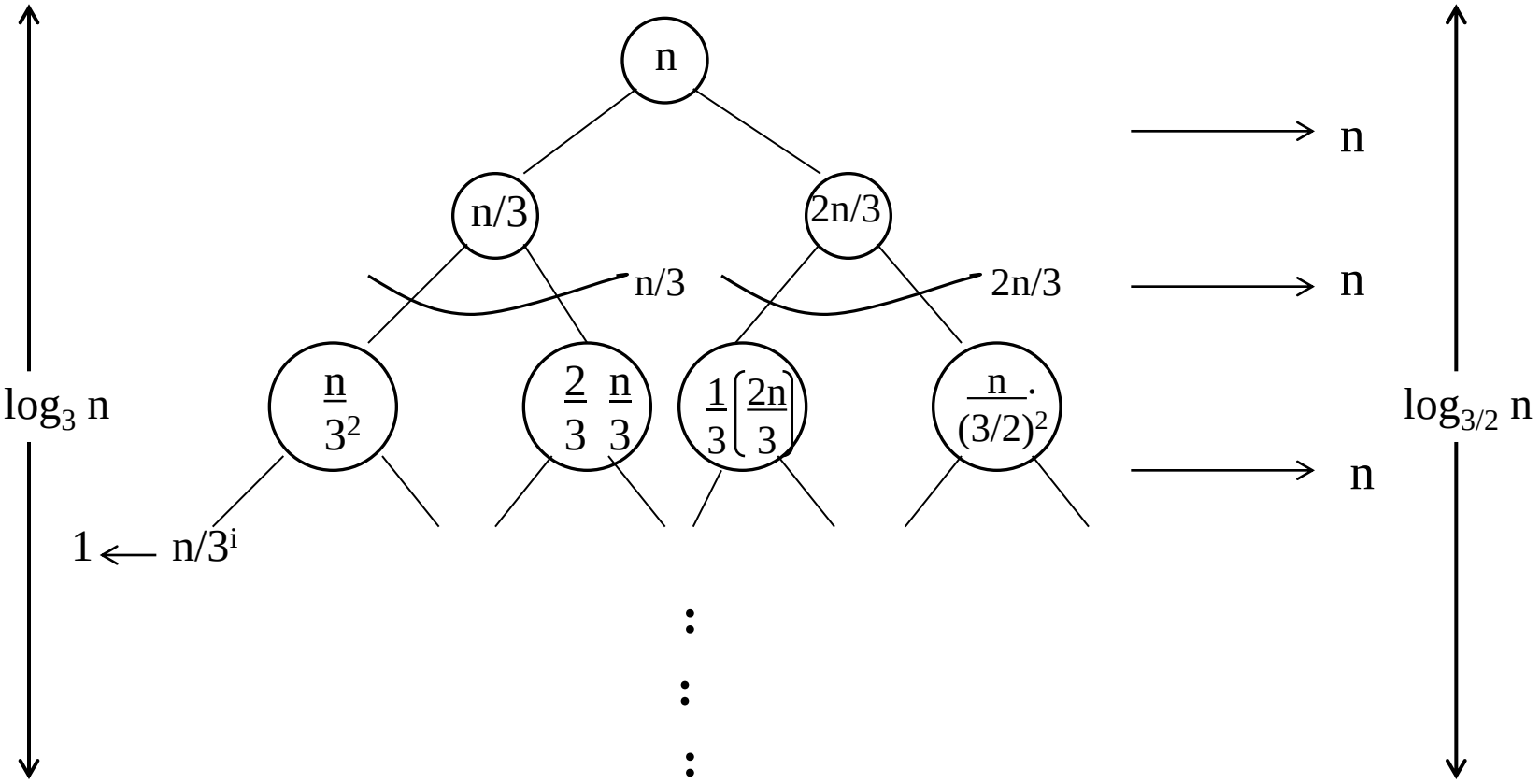
$$= \Theta\,(n - 1) \qquad \text{(neglecting the lower order terms)}$$

$$= \Theta\,(n)$$

III.

Given : **T(n) = T(n/3) + T(2n/3) + n**

Solution : The recursion tree for the above recurrence is



Home Work

Show that the solution of

$$T(n) = 4T(n/2 + 2) + n$$

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

$$T(n) = T(n-1) + T(n/2) + n$$

The master method

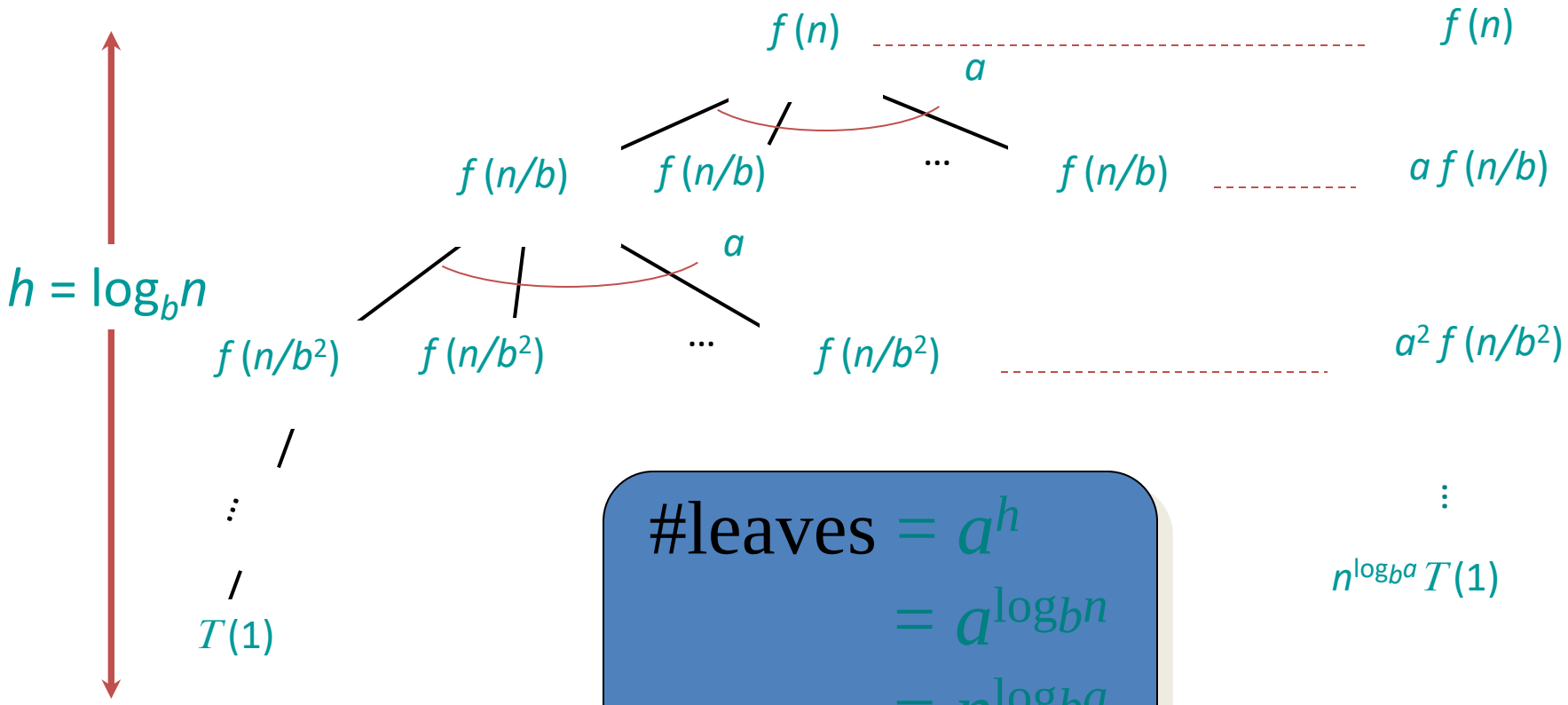
The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Idea of master theorem

Recursion tree:



$$\#\text{leaves} = a^h = a^{\log_b n} = n^{\log_b a}$$

Three common cases

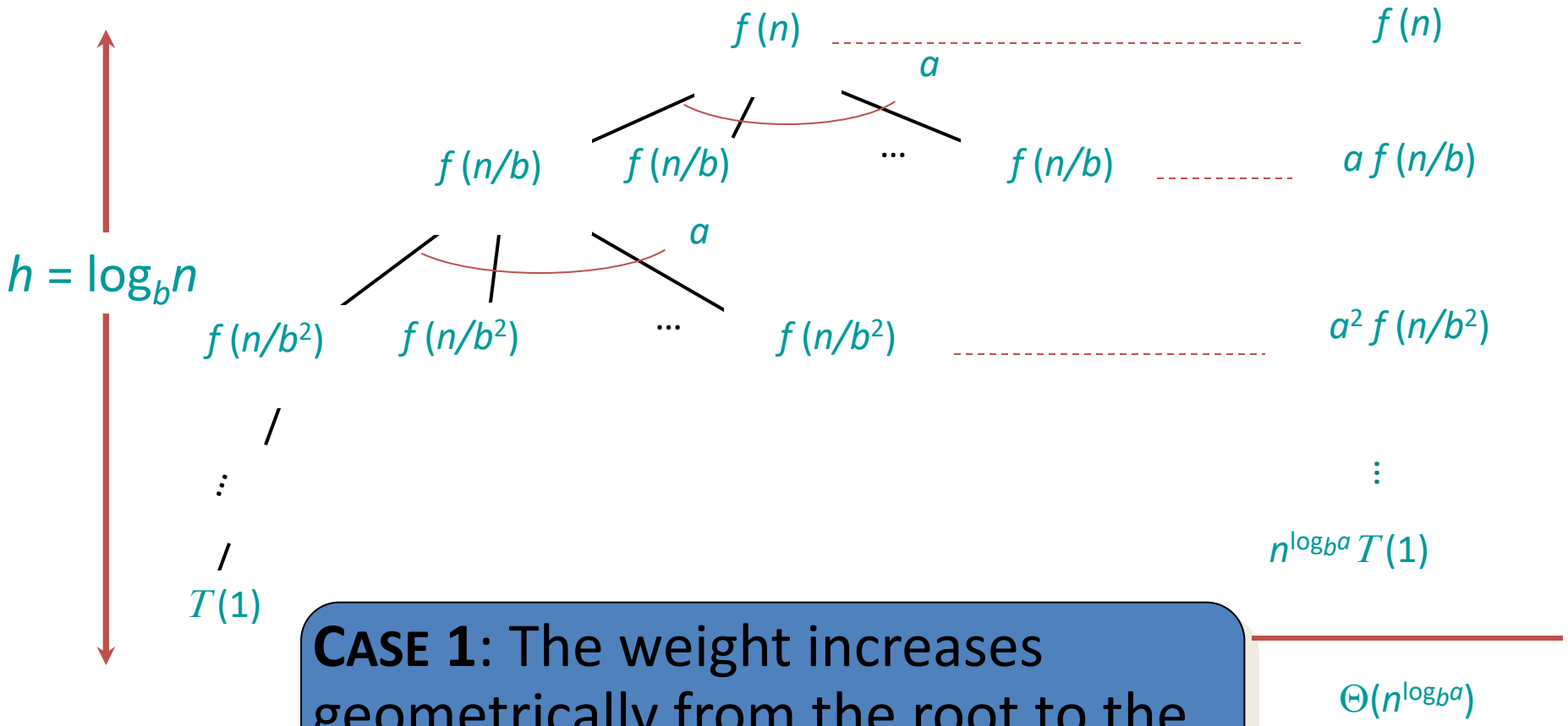
Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

Idea of master theorem

Recursion tree:



CASE 1: The weight increases geometrically from the root to the leaves.

Three common cases

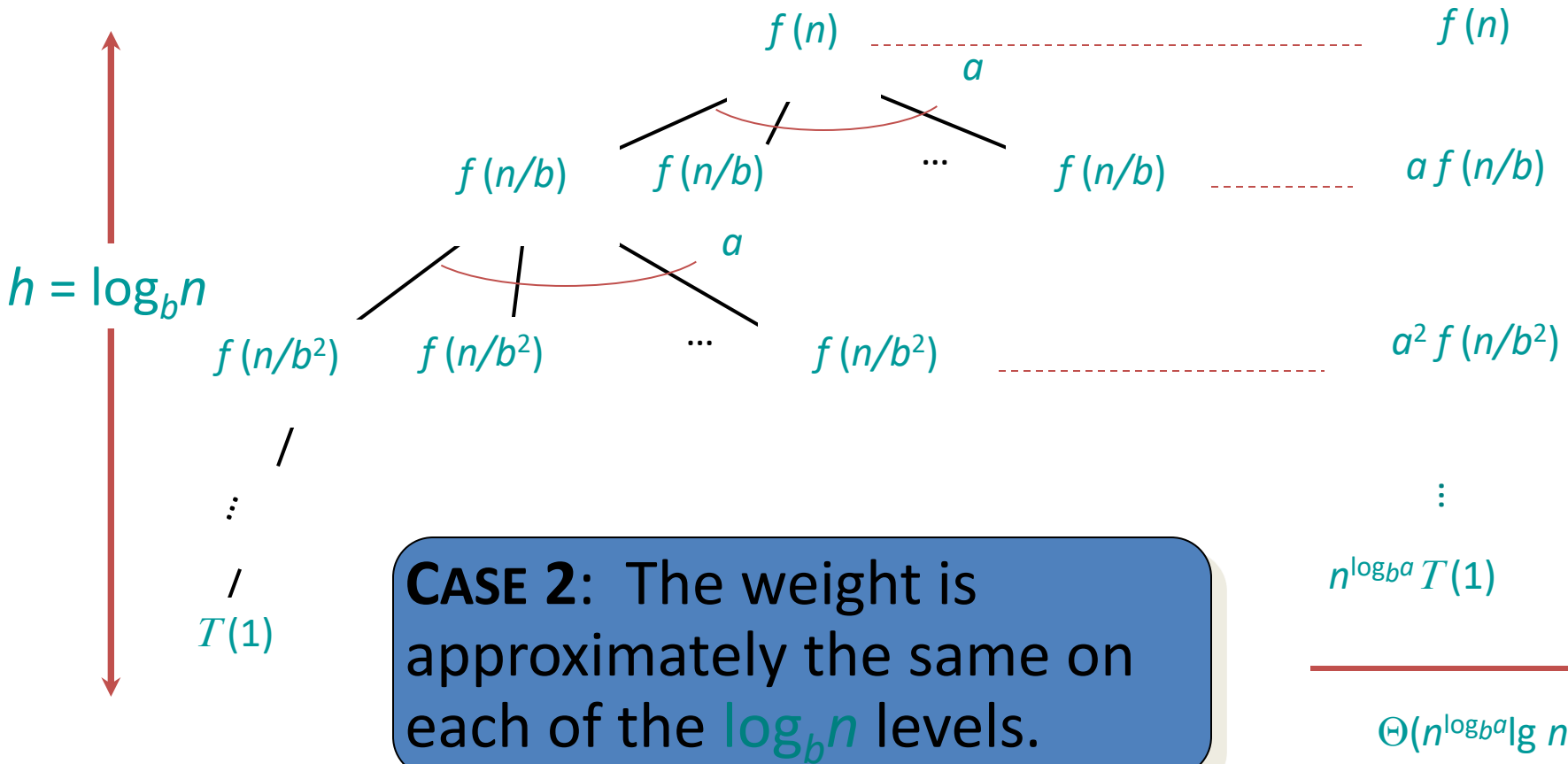
Compare $f(n)$ with $n^{\log_b a}$:

2. $f(n) = \Theta(n^{\log_b a})$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \log n)$.

Idea of master theorem

Recursion tree:



CASE 2: The weight is approximately the same on each of the $\log_b n$ levels.

Three common cases (cont.)

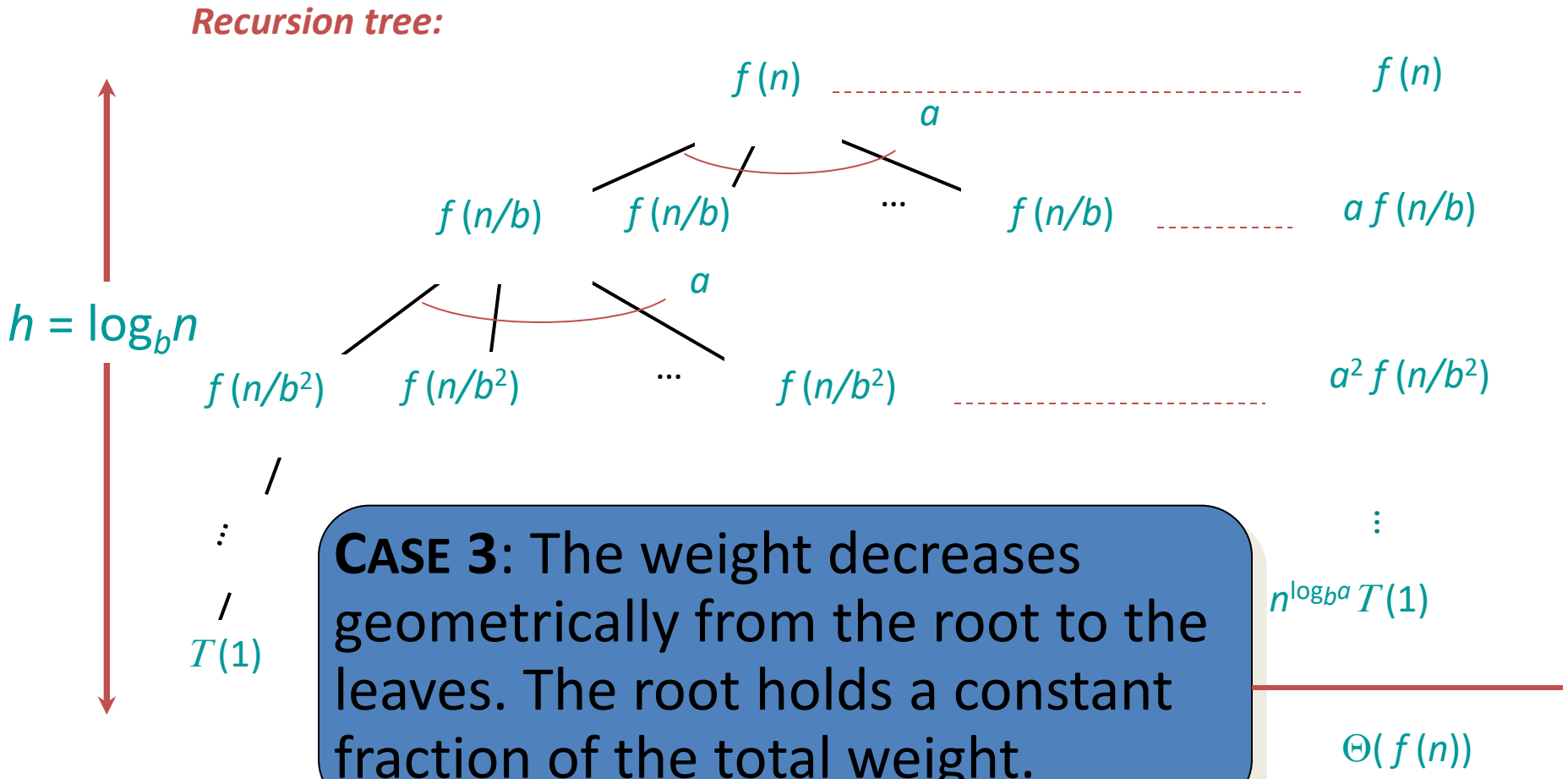
Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Idea of master theorem



CASE 3: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$

CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore T(n) = \Theta(n^2).$

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$

CASE 2: $f(n) = \Theta(n^2).$

$\therefore T(n) = \Theta(n^2 \lg n).$

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4$, $b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$.

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

Home Work of Master's Theorem

- $T(n) = 3T(n/5) + n$
- $T(n) = 2T(n/2) + n$
- $T(n) = 2T(n/2) + 1$
- $T(n) = T(n/2) + n$
- $T(n) = T(n/2) + 1$