Discrete Mathematics and Its Applications 2 (CS147)

Lecture 6: Master theorem

Fanghui Liu

Department of Computer Science, University of Warwick, UK

Our Goal

We want to solve the following recurrence relation.

$$T(n) = a \cdot T(\lceil n/b \rceil) + \Theta(n^d).$$

$$T(c) = \Theta(1) \text{ for any constant } c > 0.$$

→ $a > 0, b > 1, d \geq 0$ are some constants.

Our Goal

We want to solve the following recurrence relation.

$$T(n) = a \cdot T(\lceil n/b \rceil) + \Theta(n^d).$$

$$T(c) = \Theta(1) \text{ for any constant } c > 0.$$

$a > 0, b > 1, d \geq 0$ are some constants.

n: the problem size
a: #subproblems
n/b: the subproblem size
$\Theta(n^d)$: time cost on problem split and merge

For MERGE-SORT, we had $T(n) = 2 \cdot T(\lceil n/2 \rceil) + \Theta(n)$.

Our Goal

We want to solve the following recurrence relation.

$$T(n) = a \cdot T(\lceil n/b \rceil) + \Theta(n^d).$$

$a > 0, b > 1, d \geq 0$ are some constants.

$$T(c) = \Theta(1) \text{ for any constant } c > 0.$$

For MERGE-SORT, we had $T(n) = 2 \cdot T(\lceil n/2 \rceil) + \Theta(n)$.

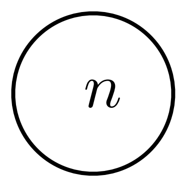
While solving the recurrence, we will typically ignore the floors and ceilings.

The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

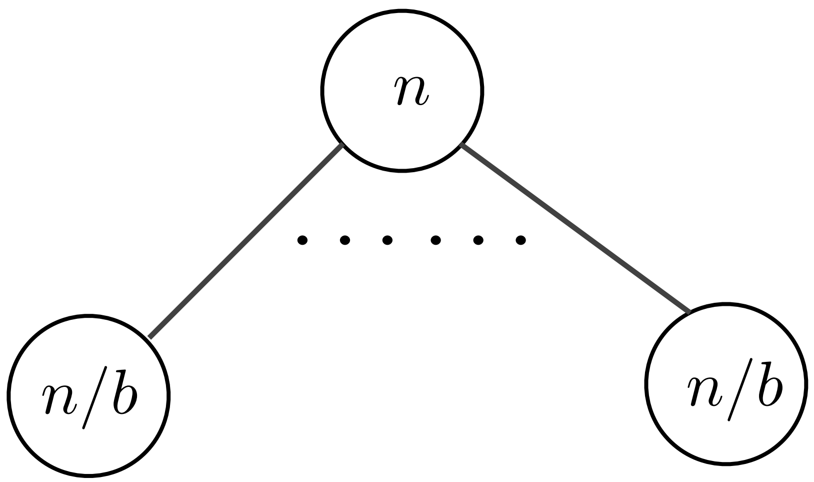
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



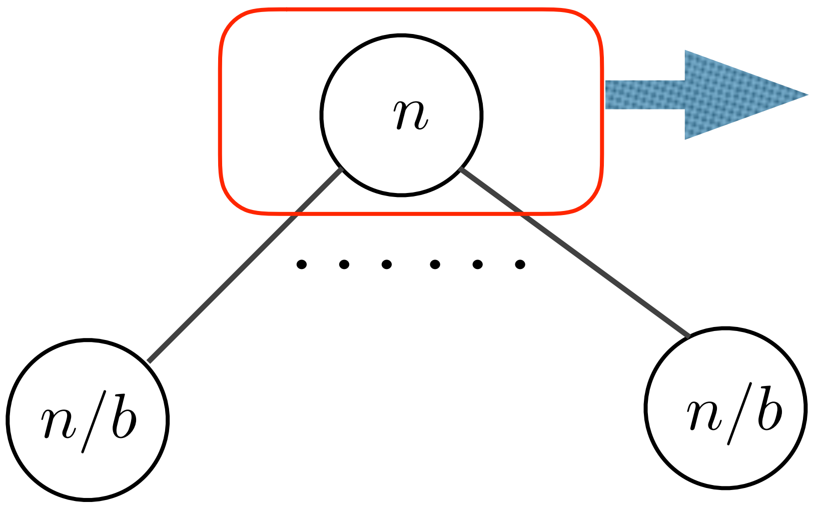
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



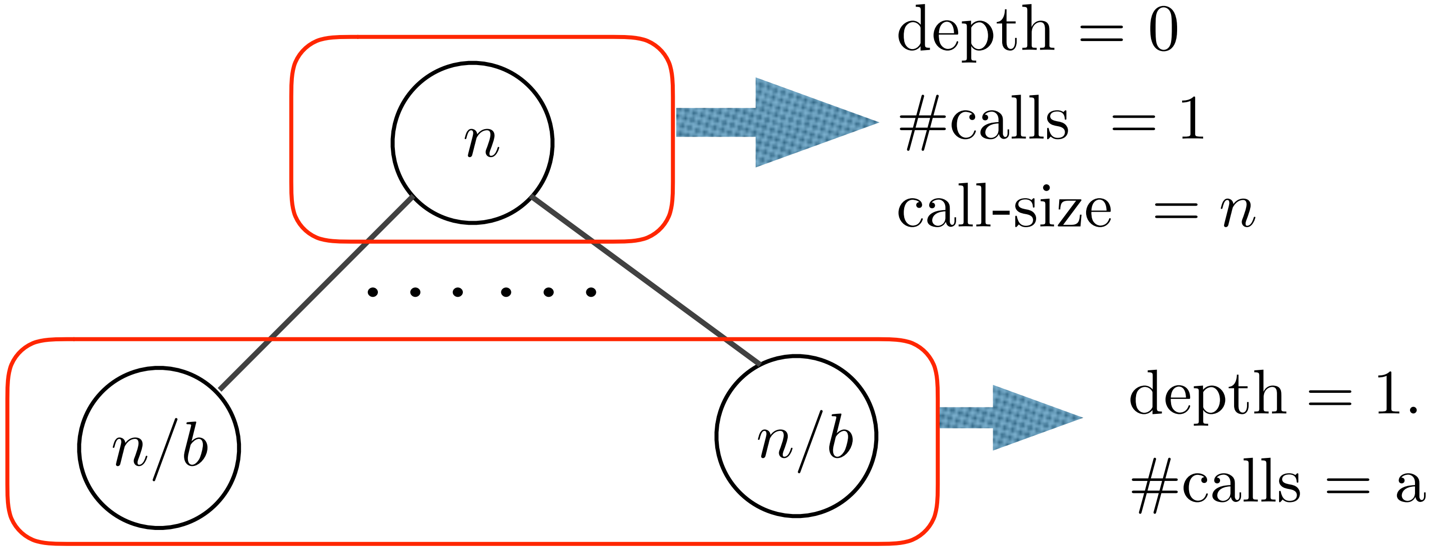
depth $= 0$

#calls $= 1$

call-size $= n$

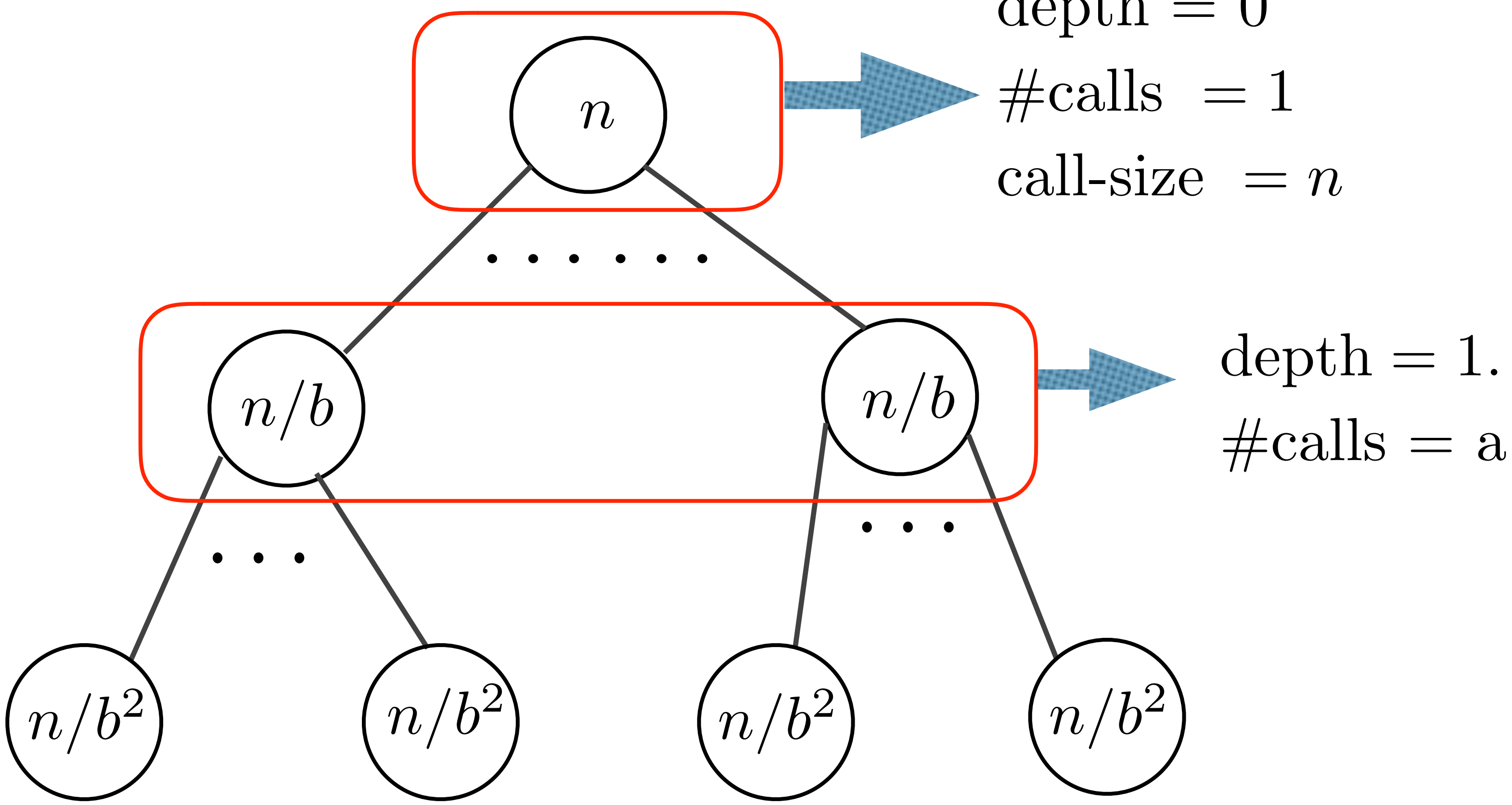
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



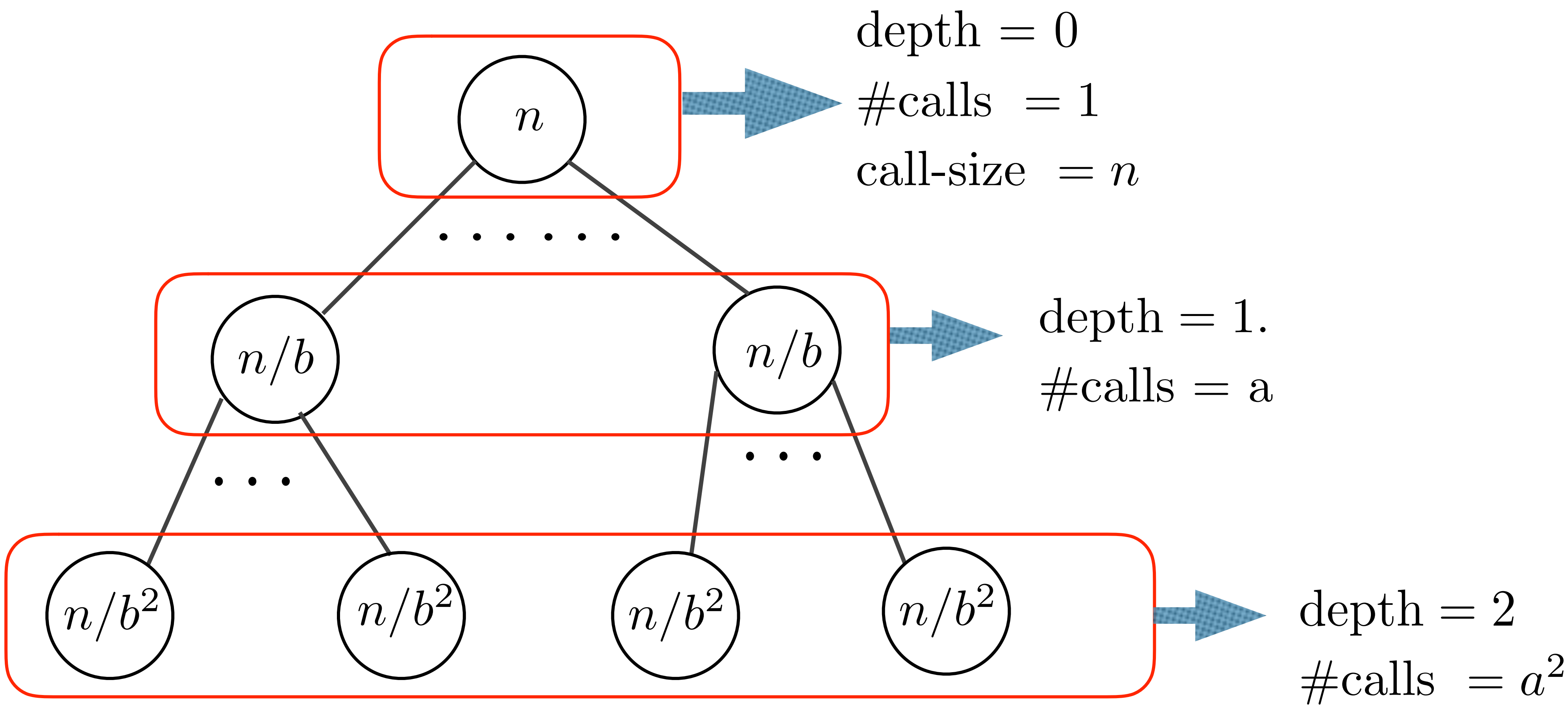
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



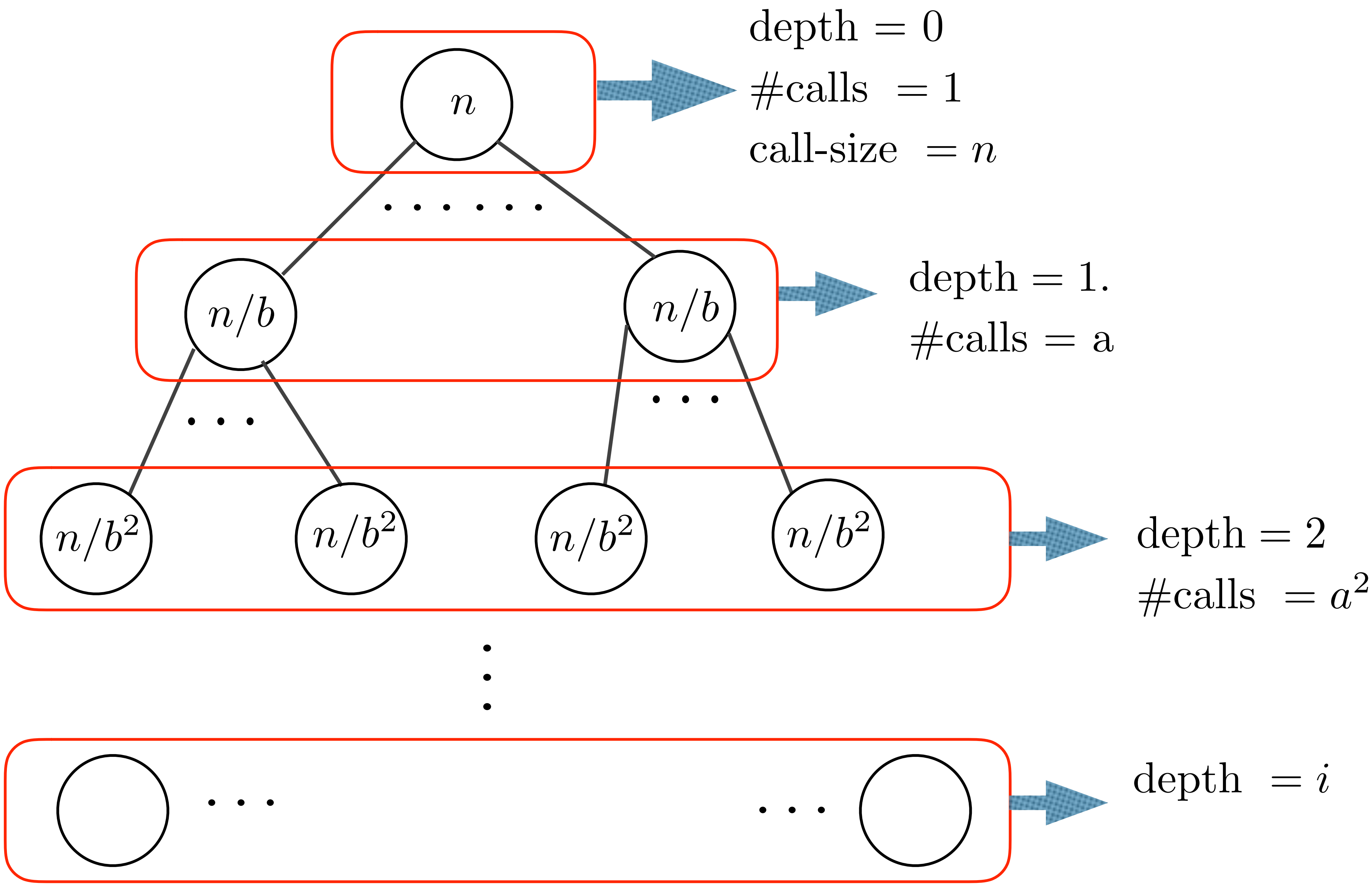
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



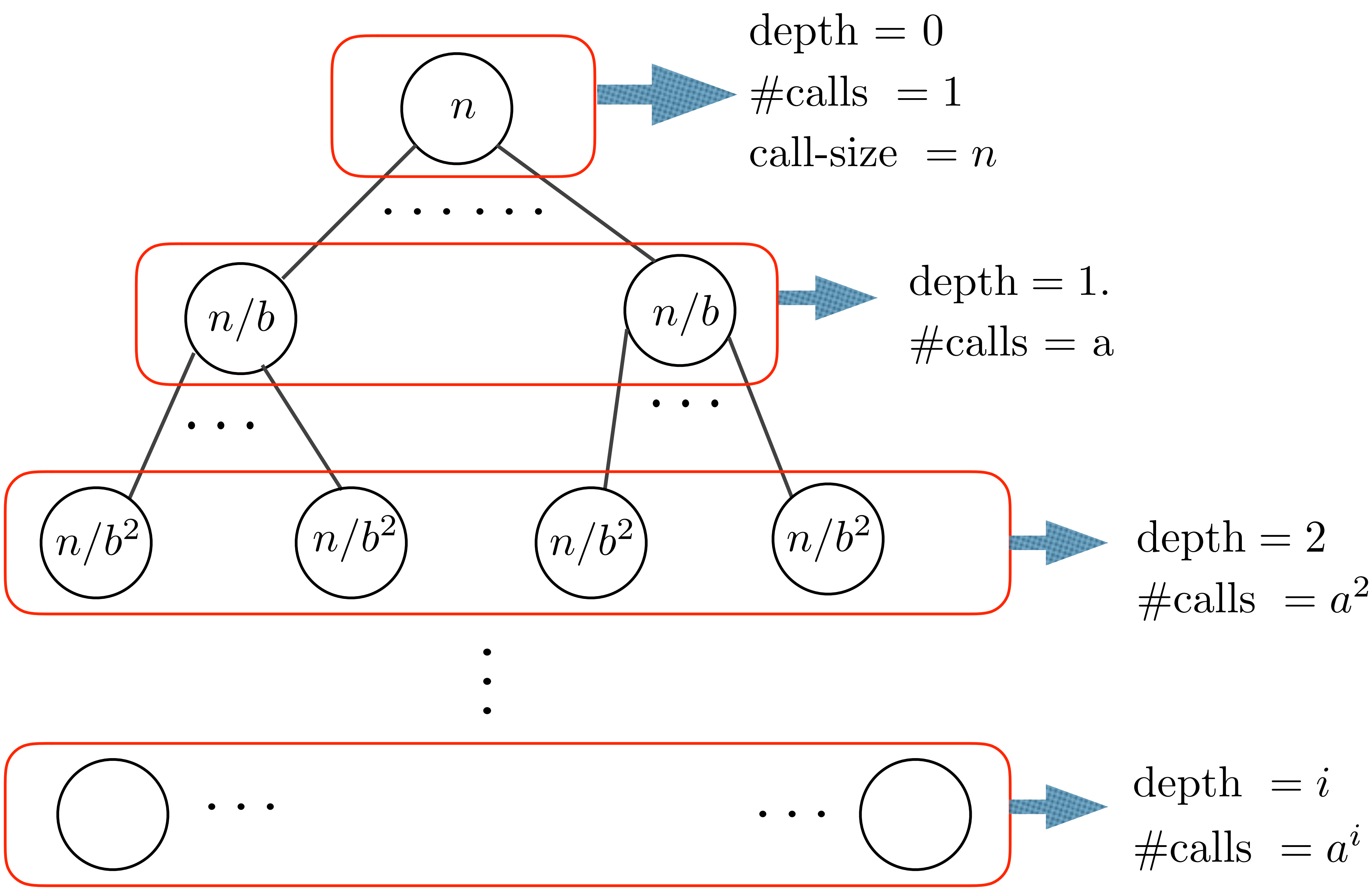
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



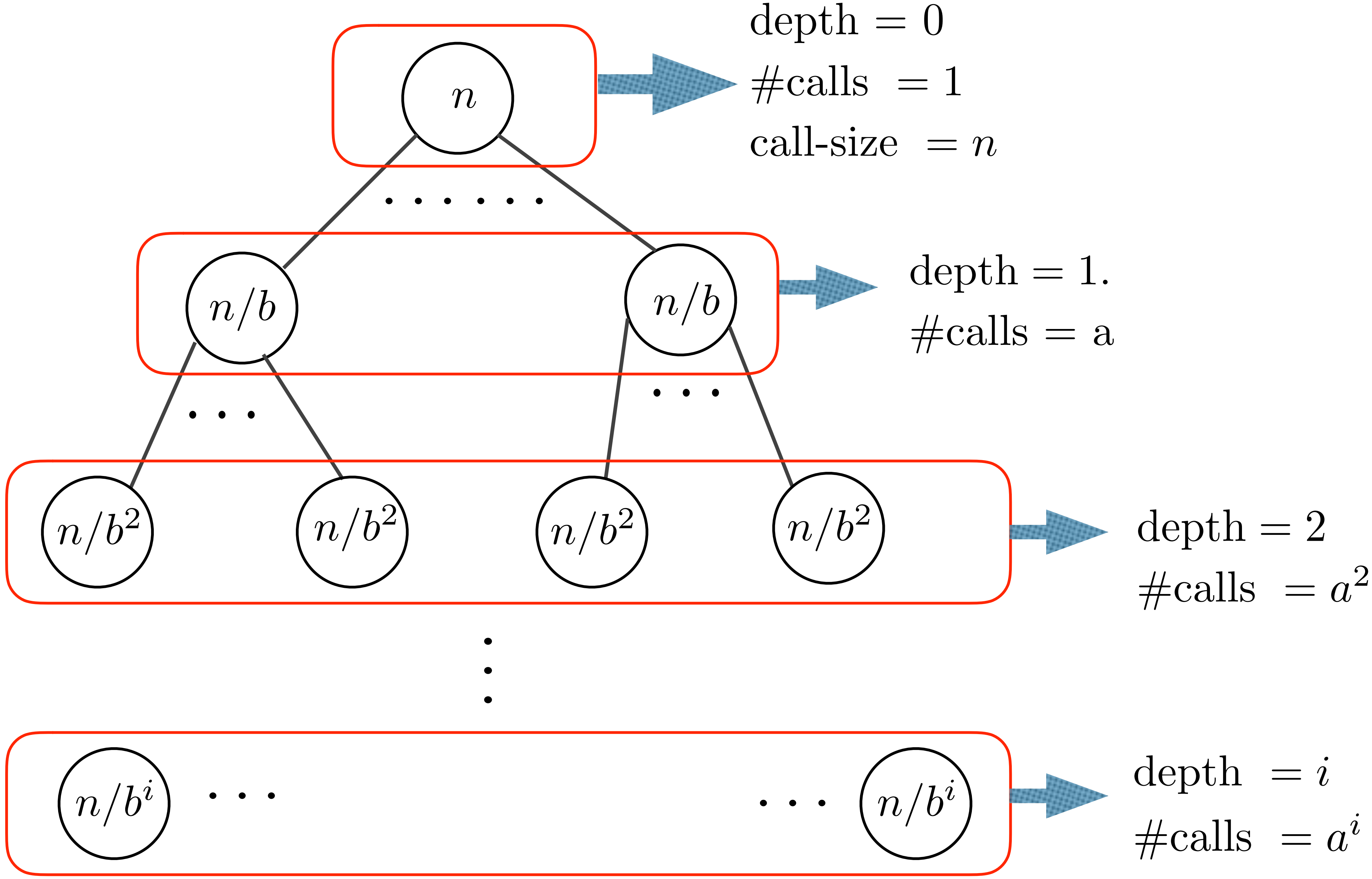
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



The Recursion Tree

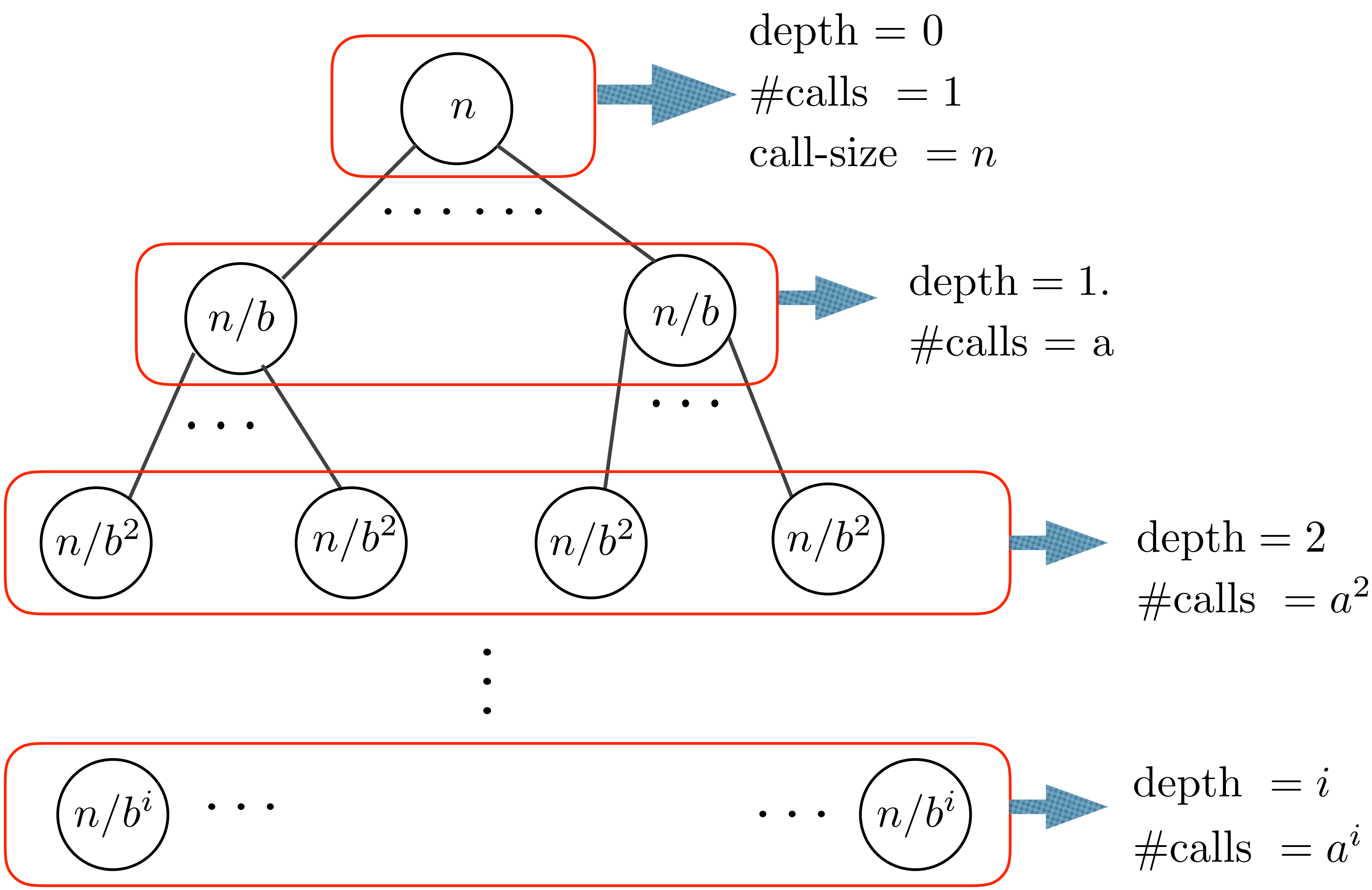
$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

$h =$ maximum depth.

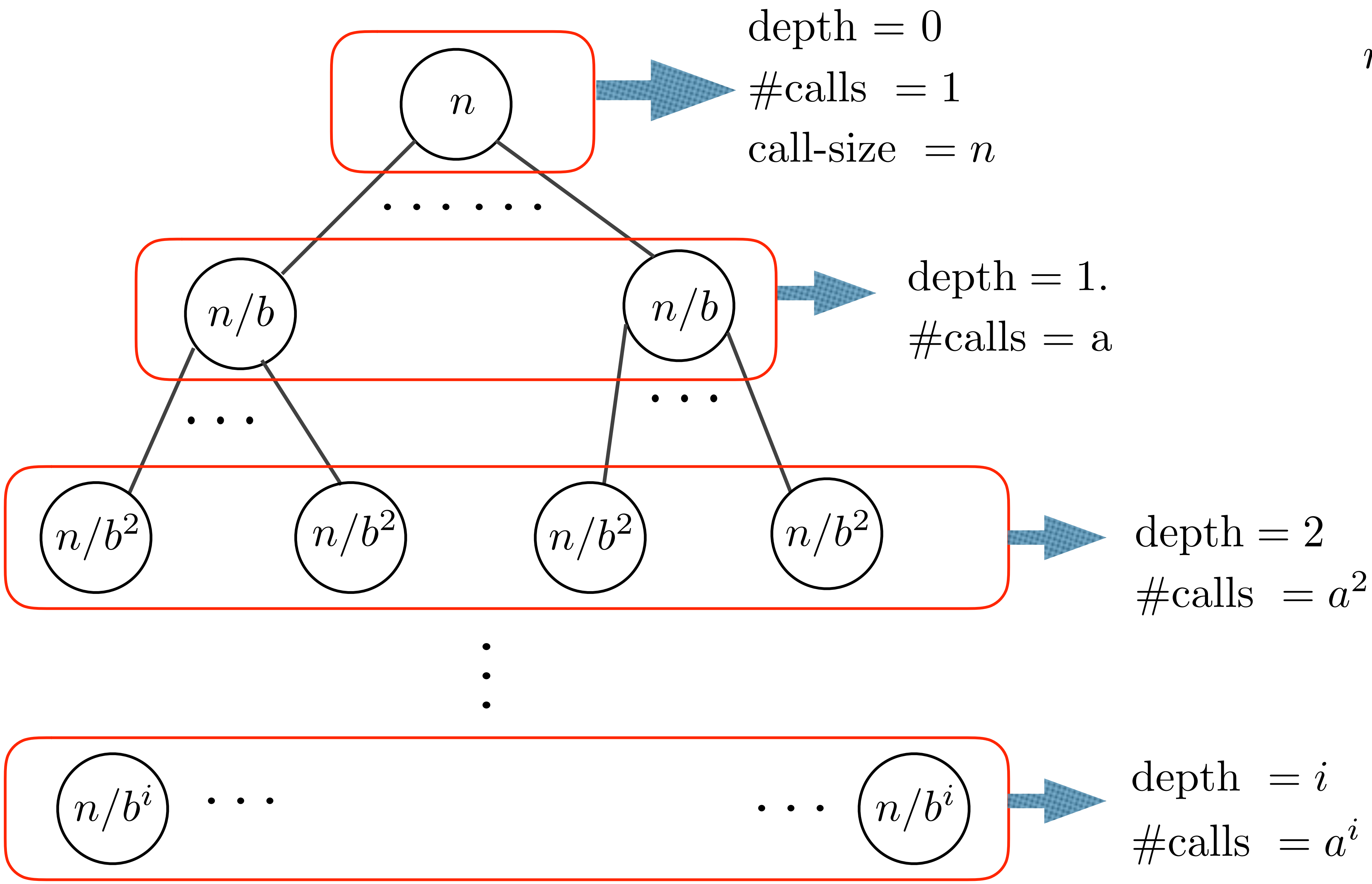


The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

h = maximum depth.

$n/b^h = 1$



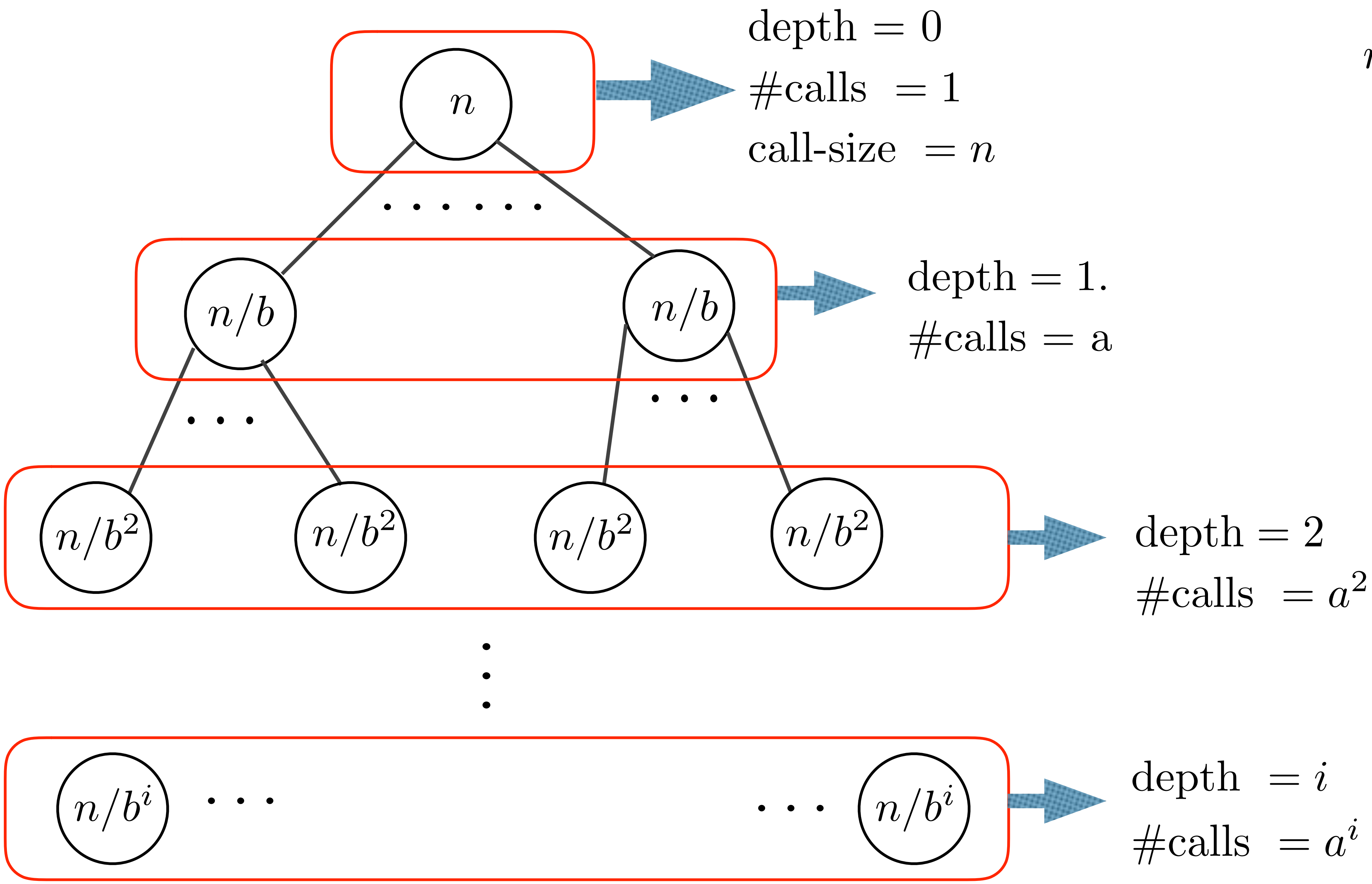
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

h = maximum depth.

$n/b^h = 1$

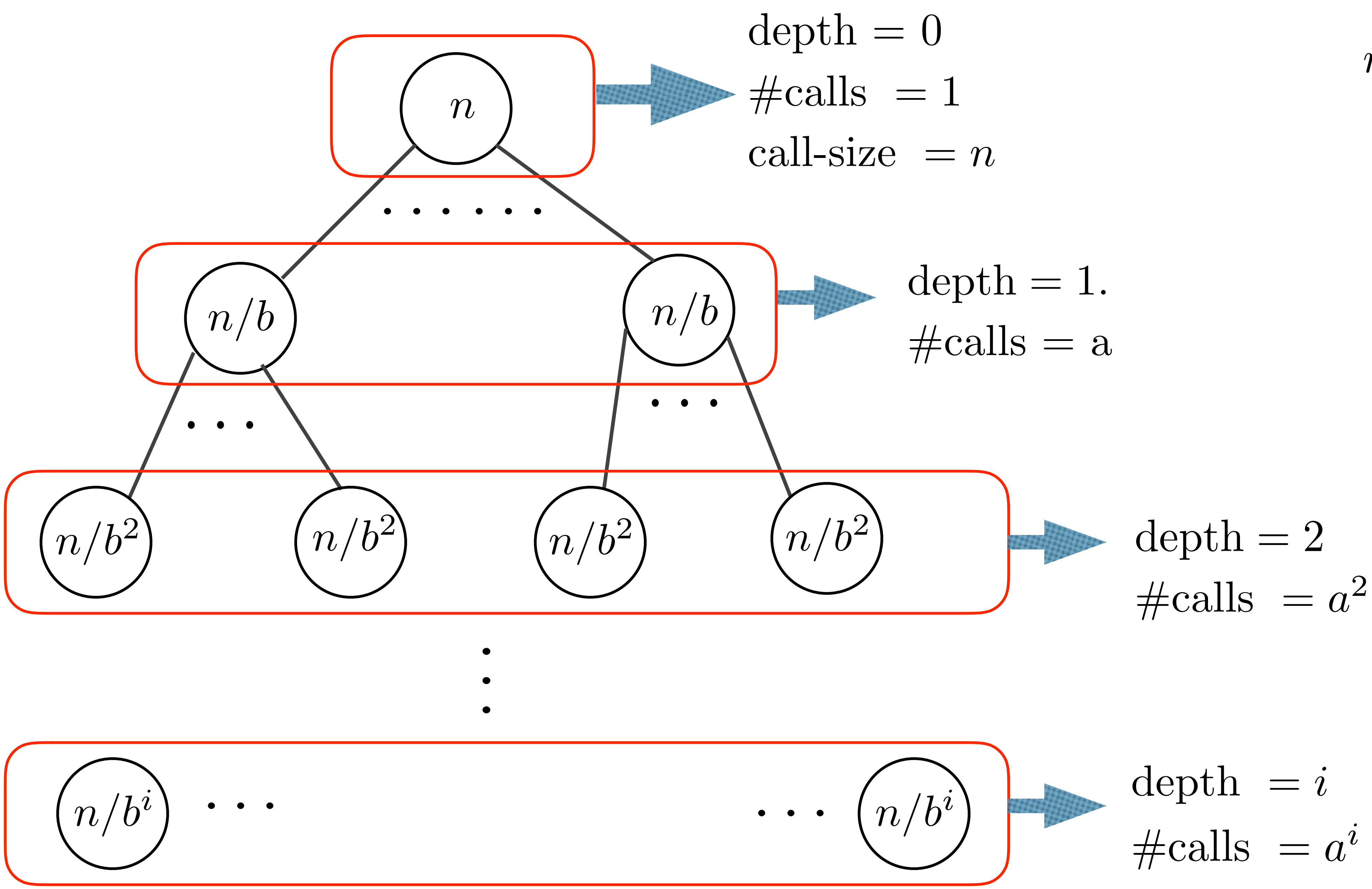
$b^h = n$



The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

h = maximum depth.



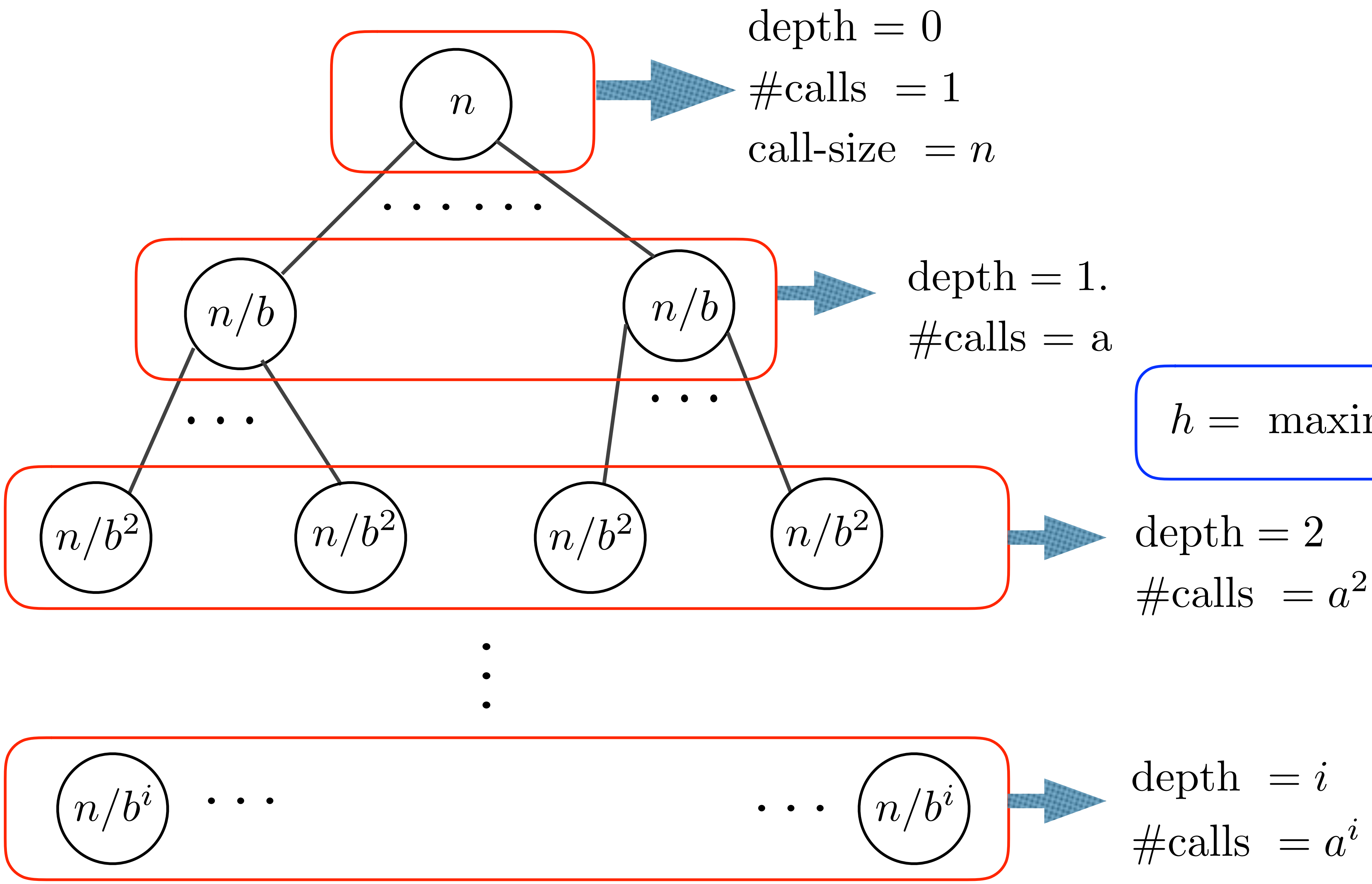
$$n/b^h = 1$$

$$b^h = n$$

$$h = \log_b n.$$

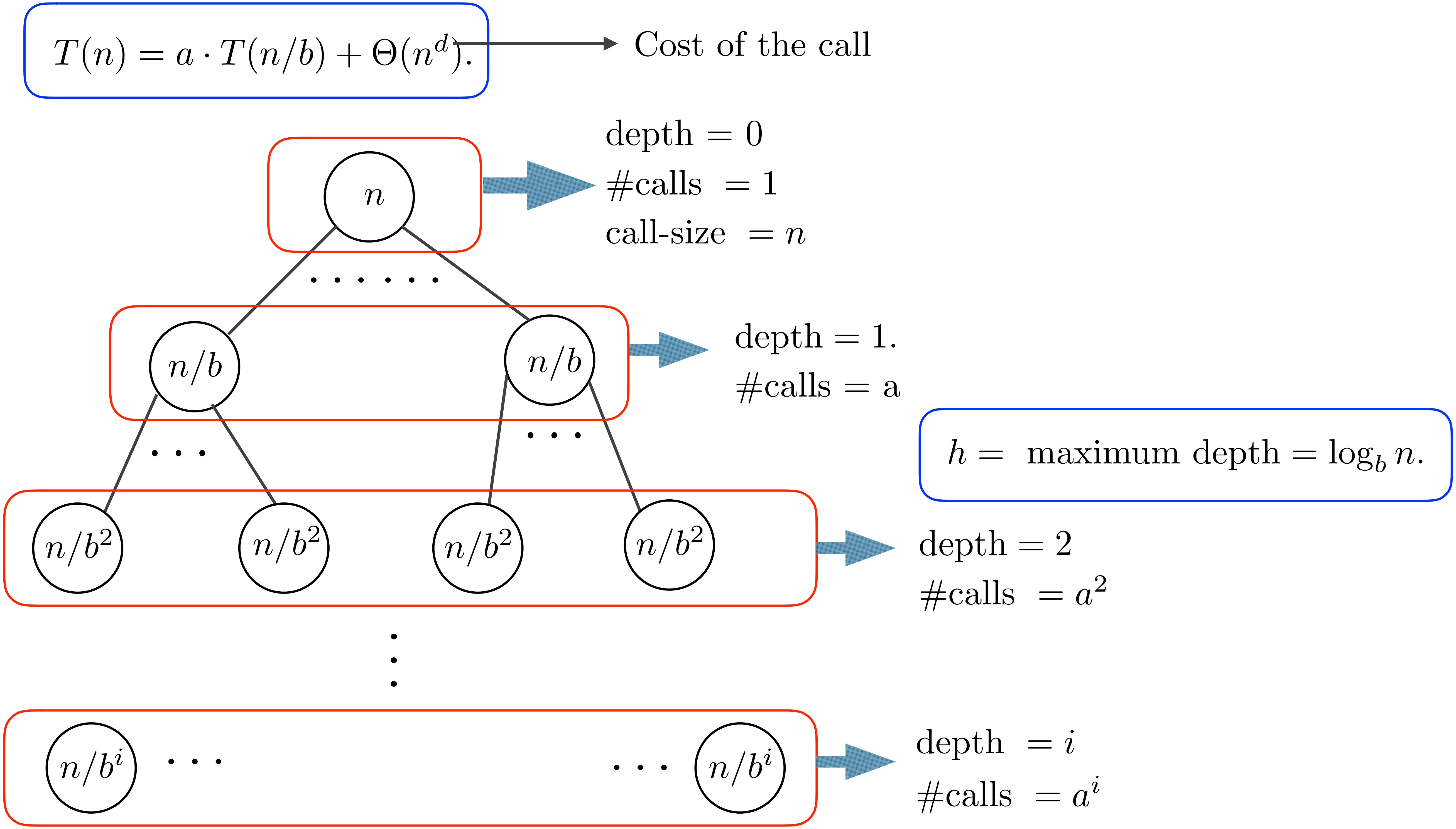
The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$



$$h = \text{ maximum depth} = \log_b n.$$

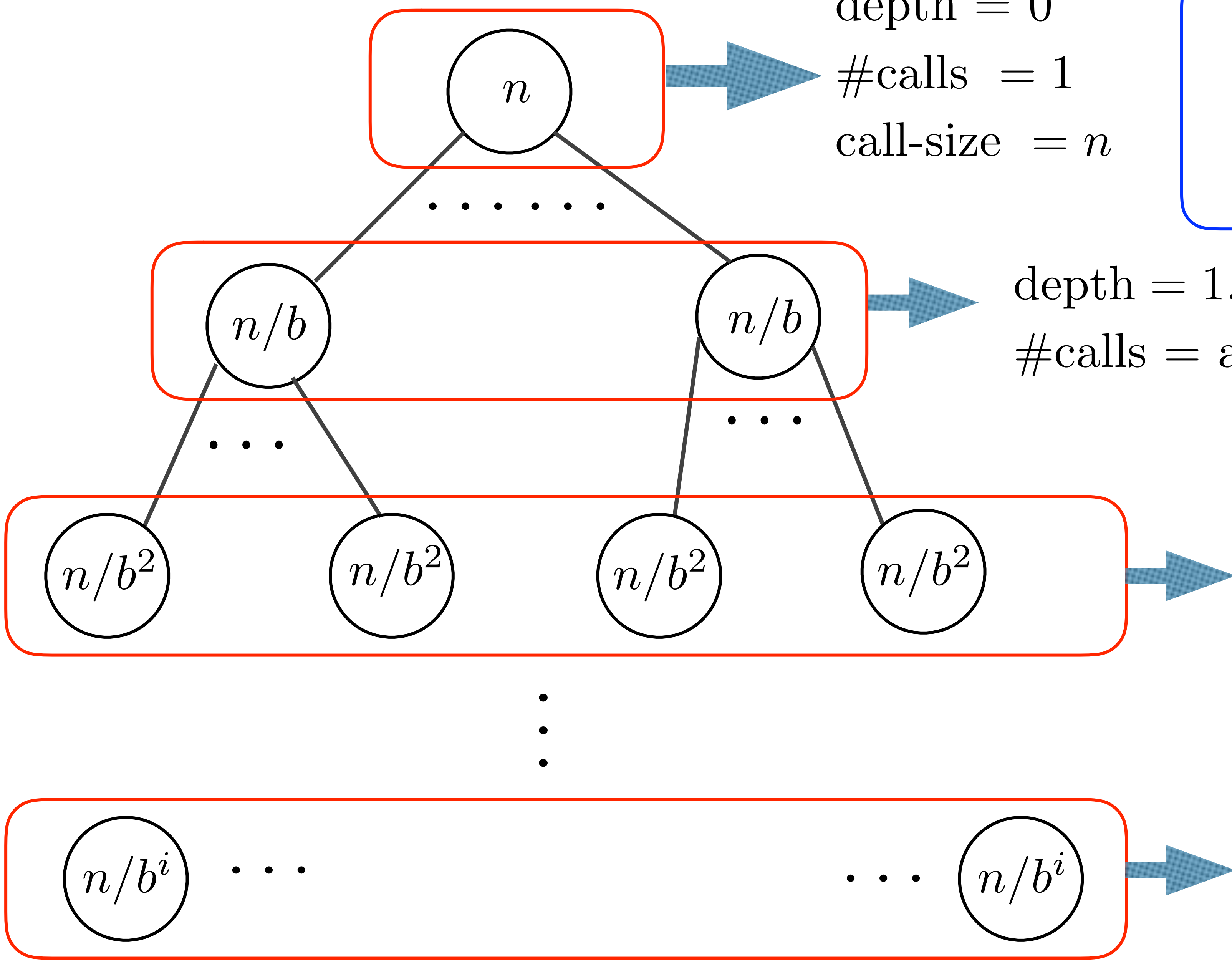
The Recursion Tree



The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

→ Cost of the call



depth $= 0$
#calls $= 1$
call-size $= n$

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$

depth $= 1.$
#calls $=$ a

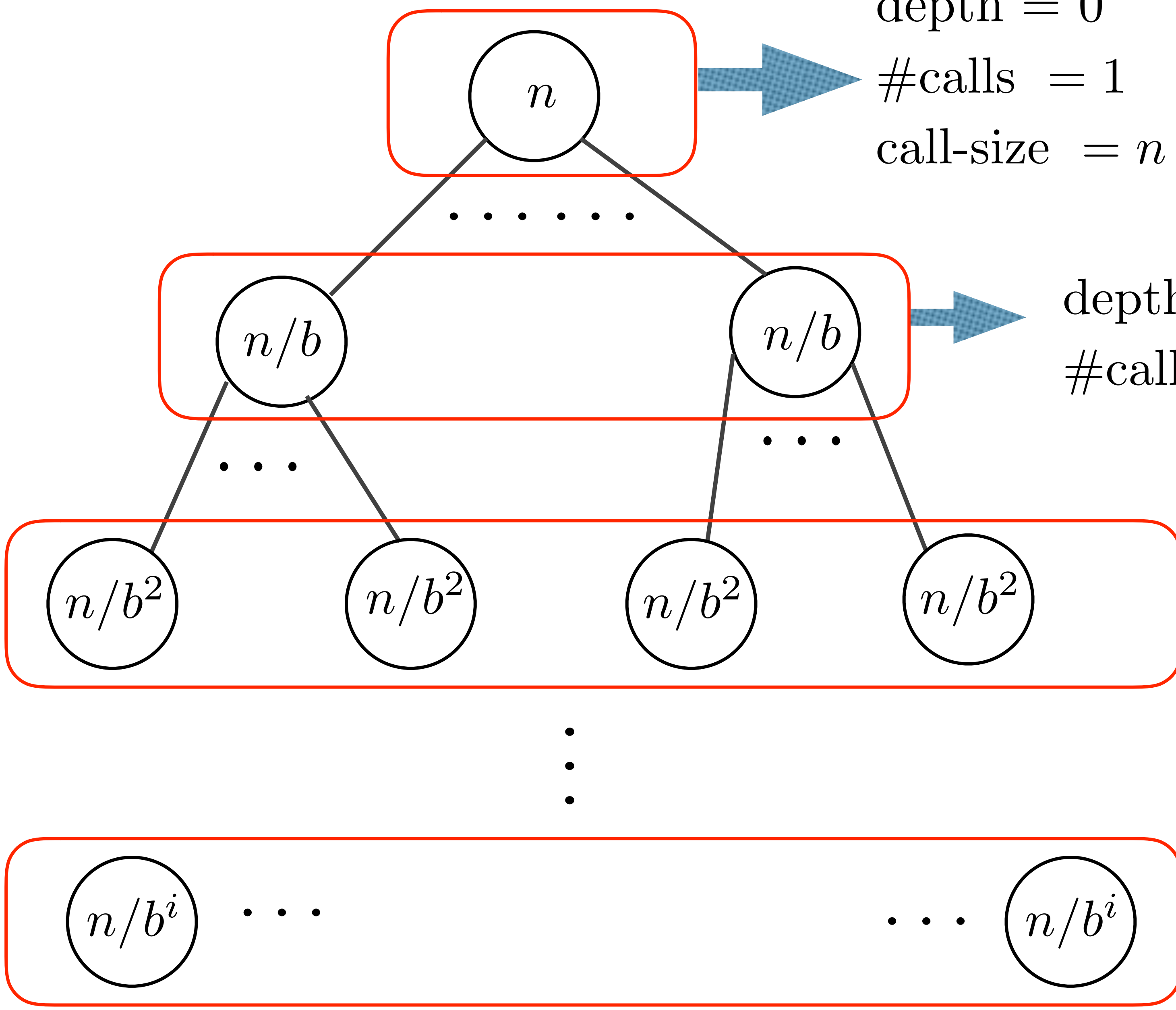
$h =$ maximum depth $= \log_b n.$

depth $= 2$
#calls $= a^2$

depth $= i$
#calls $= a^i$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call



depth $= 0$
#calls $= 1$
call-size $= n$

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$
$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

depth $= 1.$
#calls $=$ a

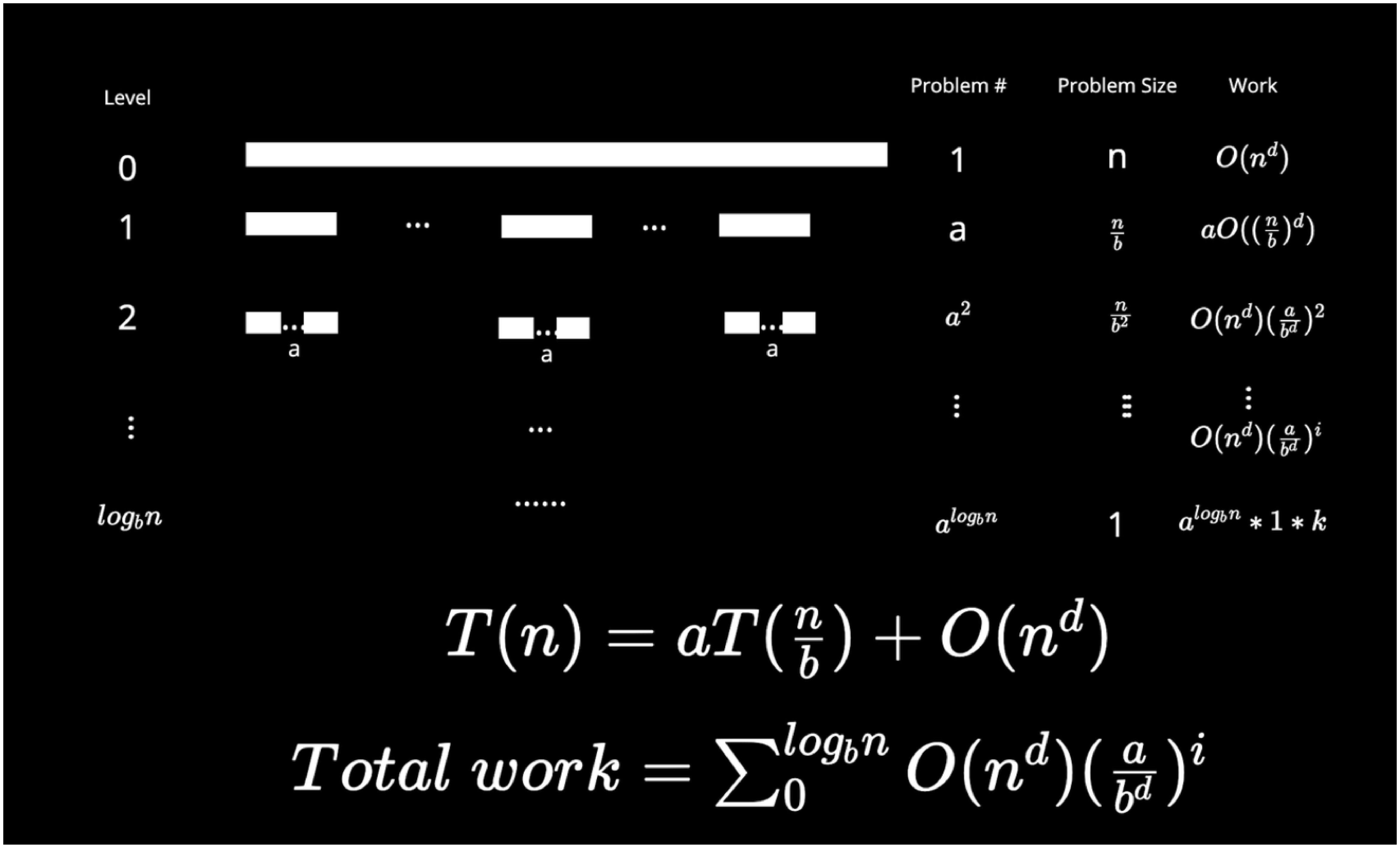
$h =$ maximum depth $= \log_b n.$

depth $= 2$
#calls $= a^2$

depth $= i$
#calls $= a^i$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call



The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call

$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$

$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$

$h =$ maximum depth $= \log_b n.$

Case I: $(a/b^d < 1)$

Case II: $(a/b^d = 1)$

Case III: $(a/b^d > 1)$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d)$. → Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$

$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d)$.

$h =$ maximum depth $= \log_b n$.

Case II: $(a/b^d = 1)$

Case III: $(a/b^d > 1)$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d)$. → Cost of the call

$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$

$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i$.

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d)$.

Case II: $(a/b^d = 1)$

$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n)$.

$h =$ maximum depth $= \log_b n$.

Case III: $(a/b^d > 1)$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d)$. → Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$

$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d)$.

Case II: $(a/b^d = 1)$

$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n)$.

$h =$ maximum depth $= \log_b n$.

Case III: $(a/b^d > 1)$

$T(n) = \Theta(n^d \cdot (a/b^d)^h)$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$
$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d).$

Case II: $(a/b^d = 1)$

$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$

$h =$ maximum depth $= \log_b n.$

Case III: $(a/b^d > 1)$

$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$

The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$

$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$$T(n) = \Theta(n^d).$$

Case II: $(a/b^d = 1)$

$$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$$

$h =$ maximum depth $= \log_b n$.

Case III: $(a/b^d > 1)$

$$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$$

$$= \Theta(n^d \cdot a^{\log_b n} / b^{d \cdot \log_b n})$$

The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

→ Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$

$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$$T(n) = \Theta(n^d).$$

$h =$ maximum depth $= \log_b n.$

Case II: $(a/b^d = 1)$

$$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$$

Case III: $(a/b^d > 1)$

$$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$$

$$= \Theta(n^d \cdot a^{\log_b n} / b^{d \cdot \log_b n}) = \Theta(n^d \cdot a^{\log_b n} / n^d)$$

The Recursion Tree

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$ → Cost of the call

$$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$$
$$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$$

Case I: $(a/b^d < 1)$

$$T(n) = \Theta(n^d).$$

$h =$ maximum depth $= \log_b n.$

Case II: $(a/b^d = 1)$

$$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$$

Case III: $(a/b^d > 1)$

$$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$$
$$= \Theta(n^d \cdot a^{\log_b n}/b^{d \cdot \log_b n}) = \Theta(n^d \cdot a^{\log_b n}/n^d)$$
$$= \Theta(a^{\log_b n})$$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call

$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$

$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d).$

$h =$ maximum depth $= \log_b n.$

Case II: $(a/b^d = 1)$

$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$

Case III: $(a/b^d > 1)$

$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$

$= \Theta(n^d \cdot a^{\log_b n} / b^{d \cdot \log_b n}) = \Theta(n^d \cdot a^{\log_b n} / n^d)$

$= \Theta(a^{\log_b n}) = \Theta(a^{\log_b a \log_a n})$

The Recursion Tree

$T(n) = a \cdot T(n/b) + \Theta(n^d).$ → Cost of the call

$T(n) = \sum_{i=0}^{h} a^i \cdot (n/b^i)^d$

$= n^d \cdot \sum_{i=0}^{h} (a/b^d)^i.$

Case I: $(a/b^d < 1)$

$T(n) = \Theta(n^d).$

$h =$ maximum depth $= \log_b n.$

Case II: $(a/b^d = 1)$

$T(n) = \Theta(n^d \cdot h) = \Theta(n^d \cdot \log_b n) = \Theta(n^d \cdot \log n).$

Case III: $(a/b^d > 1)$

$T(n) = \Theta(n^d \cdot (a/b^d)^h) = \Theta(n^d \cdot (a/b^d)^{\log_b n})$

$= \Theta(n^d \cdot a^{\log_b n}/b^{d \cdot \log_b n}) = \Theta(n^d \cdot a^{\log_b n}/n^d)$

$= \Theta(a^{\log_b n}) = \Theta(a^{\log_b a \log_a n}) = \Theta(n^{\log_b a})$

Master Theorem

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

Case I: $(a/b^d < 1)$

$$T(n) = \Theta(n^d).$$

Case II: $(a/b^d = 1)$

$$T(n) = \Theta(n^d \cdot \log n).$$

Case III: $(a/b^d > 1)$

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

An Application: Binary Search

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$, and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.

An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$, and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.

Trivial Algorithm: Scan through all the entries in the array, and for each entry check whether or not it is equal to α.

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,
and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.

Trivial Algorithm: Scan through all the entries in the array, and for each entry check whether or not it is equal to α. Takes $\Theta(n)$ time.

An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,
and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.

Binary-Search$(A[i \ldots j], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.

Binary-Search$(A[i\ldots j],\alpha)$

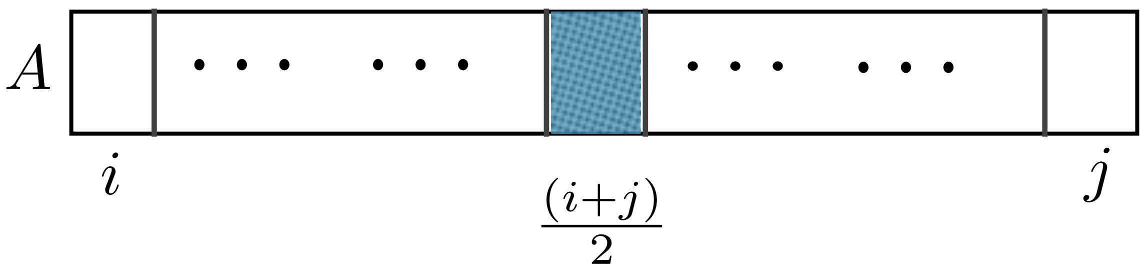
IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \ldots j], \alpha)$

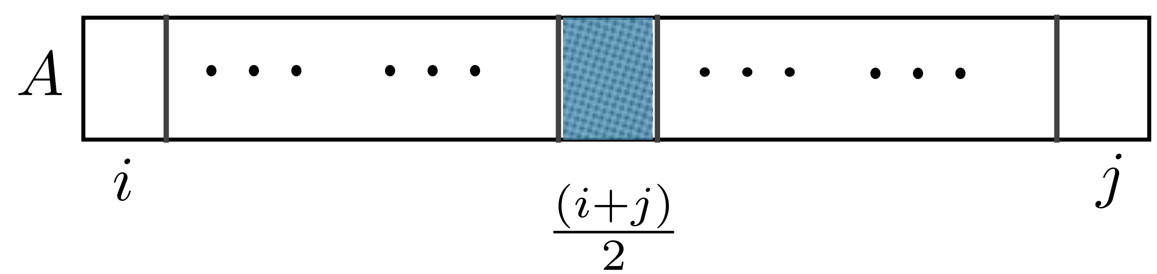
IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i + j)/2] = \alpha$, THEN RETURN $(i + j)/2$.

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$, and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \ldots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

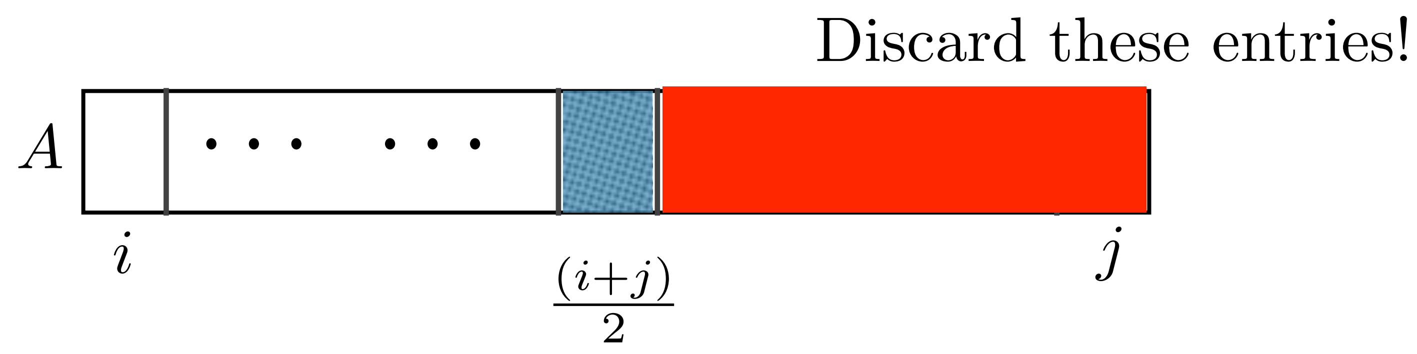
ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \ldots (i+j)/2], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \ldots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

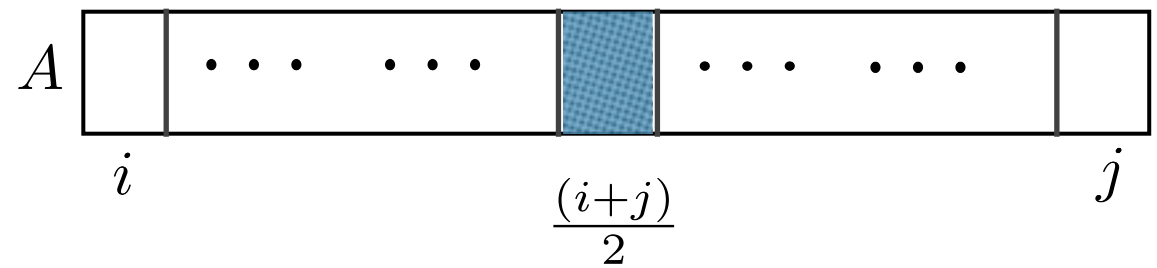
ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \ldots (i+j)/2], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1, \dots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \dots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

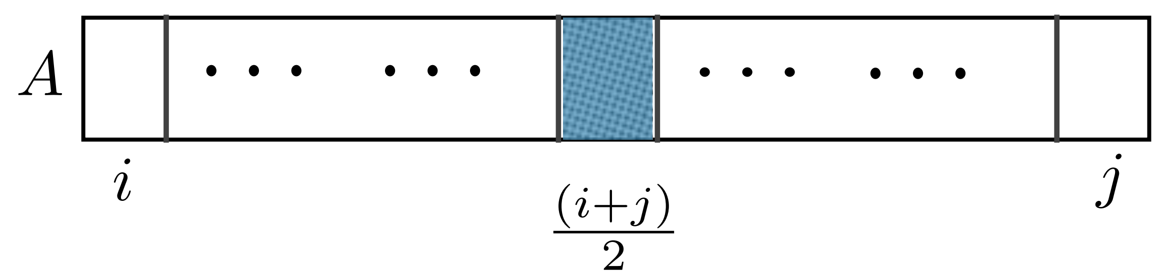
ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \dots (i+j)/2], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \ldots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \ldots (i+j)/2], \alpha)$

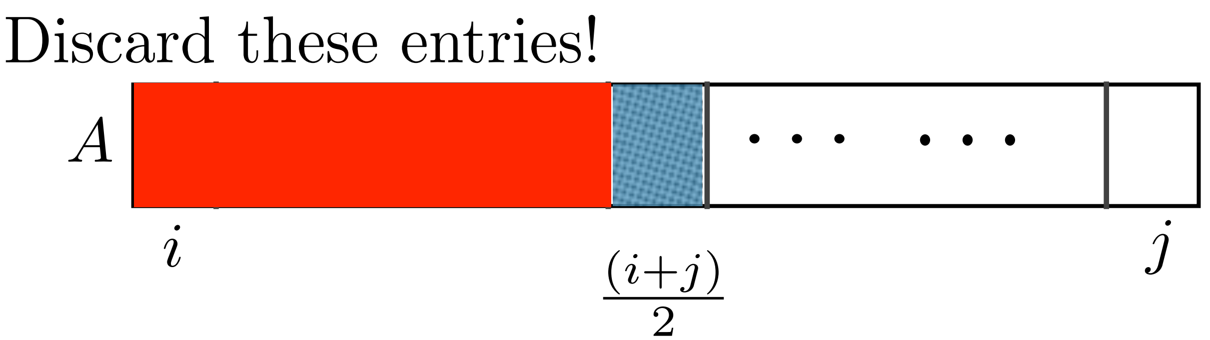
ELSE RETURN Binary-Search$(A[(i+j)/2 \ldots j], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.



Binary-Search($A[i \ldots j], \alpha$)

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search($A[i \ldots (i+j)/2], \alpha$)

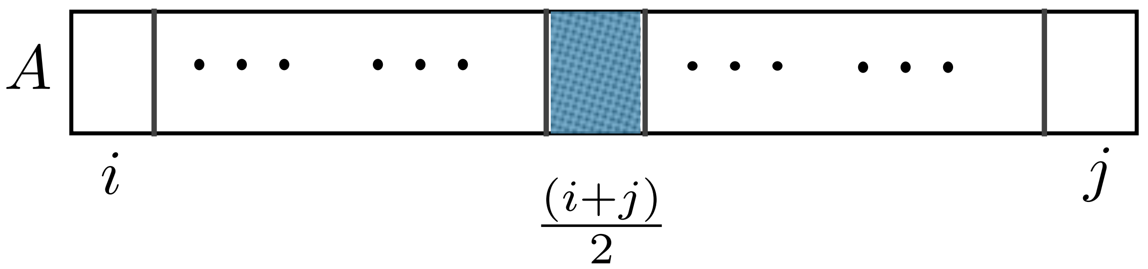
ELSE RETURN Binary-Search($A[(i+j)/2 \ldots j], \alpha$)

An Application: Binary Search

Input: A sorted array $A[1, \dots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.



Binary-Search$(A[i \dots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \dots (i+j)/2], \alpha)$

ELSE RETURN Binary-Search$(A[(i+j)/2 \dots j], \alpha)$

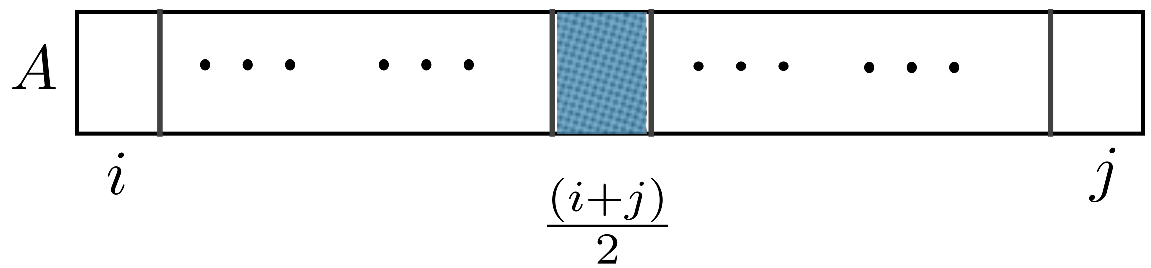
An Application: Binary Search

Input: A sorted array $A[1,\ldots,n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1,n]$ such that $A[i] = \alpha$.

$$T(n) = T(n/2) + \Theta(1)$$



Binary-Search$(A[i \ldots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \ldots (i+j)/2], \alpha)$

ELSE RETURN Binary-Search$(A[(i+j)/2 \ldots j], \alpha)$

An Application: Binary Search

Input: A sorted array $A[1, \ldots, n]$ where $A[1] \leq A[2] \leq \cdots \leq A[n]$,

and a real number α.

Goal: Check whether or not there is an index $i \in [1, n]$ such that $A[i] = \alpha$.

$T(n) = T(n/2) + \Theta(1)$ ⇒ $T(n) = \Theta(\log n)$.

Binary-Search$(A[i \ldots j], \alpha)$

IF $i = j$, THEN RETRUN i if $A[i] = \alpha$, and NO otherwise.

IF $A[(i+j)/2] = \alpha$, THEN RETURN $(i+j)/2$.

ELSE IF $A[(i+j)/2] > \alpha$, THEN RETURN Binary-Search$(A[i \ldots (i+j)/2], \alpha)$

ELSE RETURN Binary-Search$(A[(i+j)/2 \ldots j], \alpha)$

General functions in master theorem

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

How to handle beyond polynomial function? e.g., log function

General functions in master theorem

$$T(n) = a \cdot T(n/b) + \Theta(n^d).$$

How to handle beyond polynomial function? e.g., log function

$$\begin{aligned} T(n) &:= aT\left(\frac{n}{b}\right) + n \log n \\ &\leq \underbrace{aT\left(\frac{n}{b}\right) + \Theta(n^{1+\epsilon})}_{:=T_1(n)} \quad \text{for any } \epsilon > 0 \end{aligned}$$

$$\begin{aligned} T(n) &:= aT\left(\frac{n}{b}\right) + n \log n \\ &\geq \underbrace{aT\left(\frac{n}{b}\right) + \Theta(n)}_{:=T_2(n)} \end{aligned}$$

https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/mm-proof.pdf