Design and analysis of Algorithms

Chapter-3

Mathematical analysis of Algorithms

Revision on the analysis of Algorithms

- **The analysis framework**
  Time Complexity
  Space Complexity
- **Measuring input size**
  mostly straight forward
  e.g the maximum number in an array ,
  number of elements in an array.
- **Choose of parameters sometimes matter.**
  e.g nxn matrix
  n->order of matrix
  N-> total number of elements in the matrix
- **Units of measuring Running Time**
  its a metrics that does not depend on some extraneous factors.
  Through counting basic operations
  (the operation that takes max time).

Non-recursive Algorithms

A non-recursive algorithm **does the operations all at once**, without calling itself. E.g Bubble sort.

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules.

Example 1: The Largest Element

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

T(n) = $\sum_{1 \le i \le n-1} (1) = n-1 = \Theta(n)$ comparisons

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

$$\sum_{i=l}^{u} 1 = u - l + 1$$

$$\sum_{i=0}^{n-2} n+1-i = (n-1)+(n-2)+\ldots..+1 =$$

$$= \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 = \Theta(n^2)$$

T(n) = *Σ0≤i≤n-2 (Σi+1≤j≤n-1 (1))*

= *Σ0≤i≤n-2 (n-i+1)* =

= Θ(n^2) comparisons

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

T(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$ = $\Theta(n^3)$ multiplications

Ignored addition for simplicity

Example 4: Counting binary digits

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

It cannot be investigated the way the previous examples are.

The halving game: Find integer i such that $n/2^i \leq 1$.

Answer:* *i ≤ log n. ***So, T(n) = Θ(log n)*** divisions.

Another solution: Using recurrence relations.

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input’s size.

- Identify the algorithm’s basic operation.

- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)

- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.

- Solve the recurrence (or, at the very least, establish its solution’s order of growth) by backward substitutions or another method.

Example 1 : Factorial: Recursive Algorithm

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

The stopping condition is n=0

A repetitive algorithm uses recursion whenever the algorithm appears within the definition itself.

Example 1: Recursive evaluation of n!

Definition: $n\,! = 1 \ast 2 \ast \ldots \ast (n\text{-}1) \ast n$ for $n \geq 1$ and $0! = 1$

Recursive definition of n!: $F(n) = F(n\text{-}1) \ast n$ for $n \geq 1$ and
$F(0) = 1$

ALGORITHM $F(n)$

//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if $n = 0$ **return** 1
else return $F(n-1) * n$

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{bmatrix}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

Size: n

Basic operation: multiplication

Recurrence relation: M(n) = M(n-1) (for F(n-1)) + 1 (for the n * F(n-1))

M(0) = 0

Solving the recurrence for M(n)

$M(n) = M(n-1) + 1, \quad M(0) = 0$ (no multiplication when n=0)

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= (M(n-2) + 1) + 1 = M(n-2) + 2 \\ &= (M(n-3) + 1) + 2 = M(n-3) + 3 \\ &\ldots \\ &= M(n-i) + i \\ &= M(0) + n \\ &= n \end{aligned}$$

The method is called backward substitution.

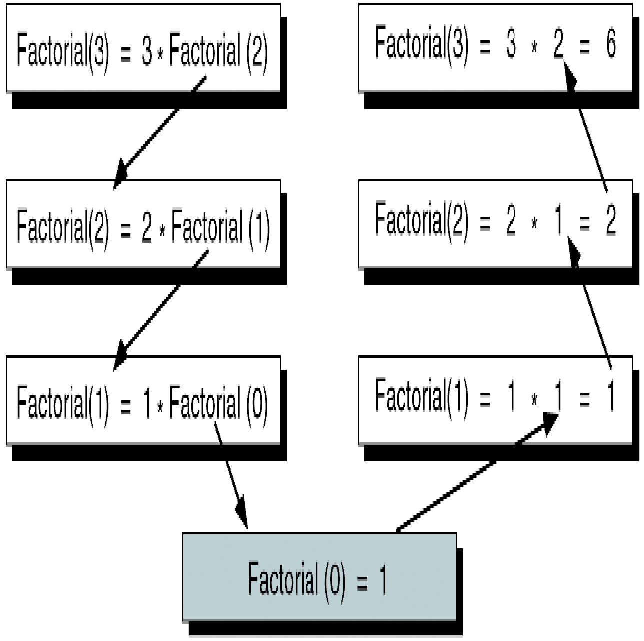


FIGURE 2-3 Factorial (3) Recursively

Example 2: The Tower of Hanoi Puzzle

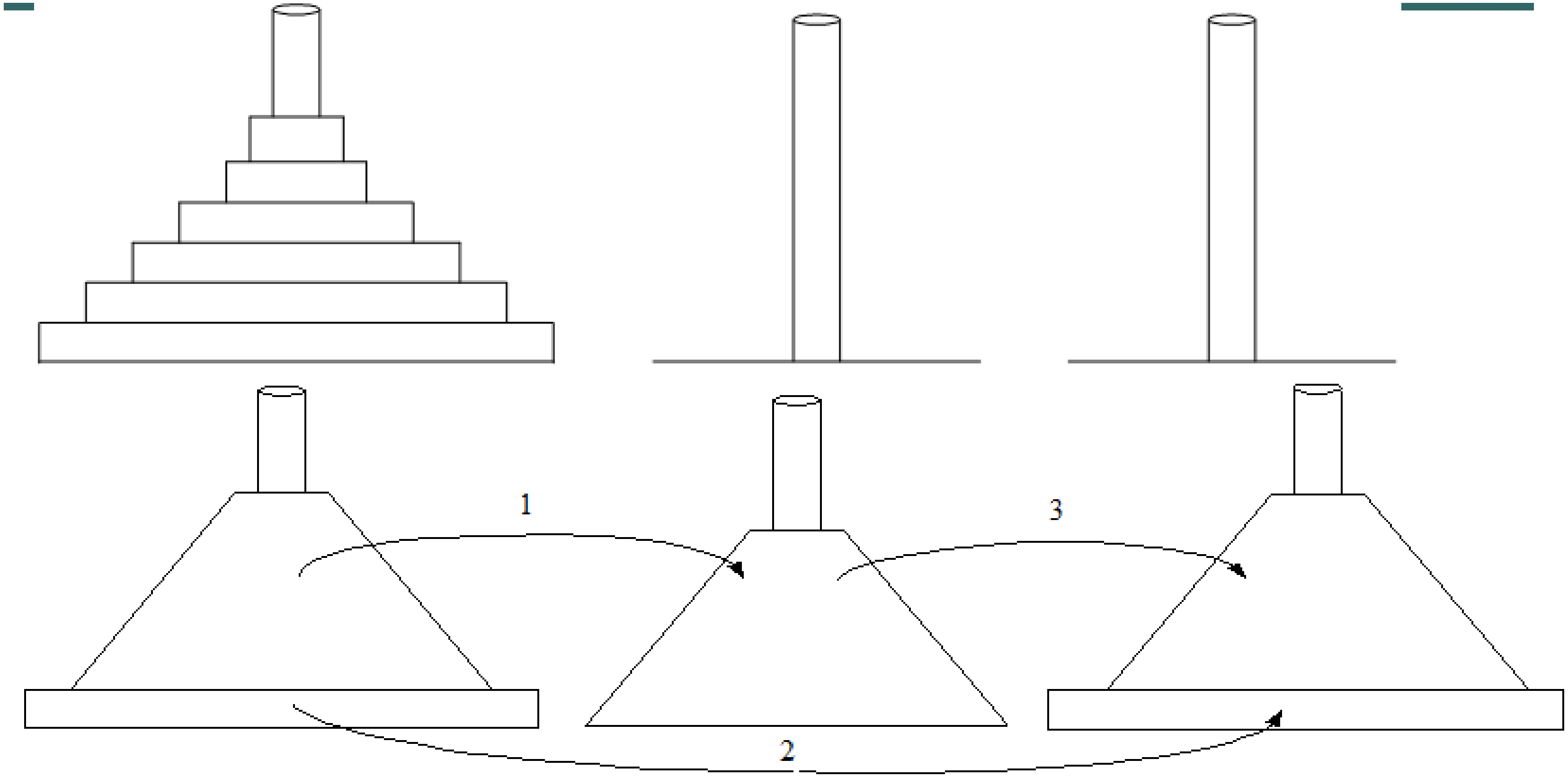
Towers of Hanoi (aka Tower of Hanoi) is a mathematical puzzle invented by a French Mathematician Edouard Lucas in 1983.

•Initially the game has few discs arranged in the increasing order of size in one of the tower.

• The number of discs can vary, but there are only three towers.

•The goal is to transfer the discs from one tower another tower. However you can move only one disk at a time and you can never place a bigger disc over a smaller disk. It is also understood that you can only take the top most disc from any given tower.

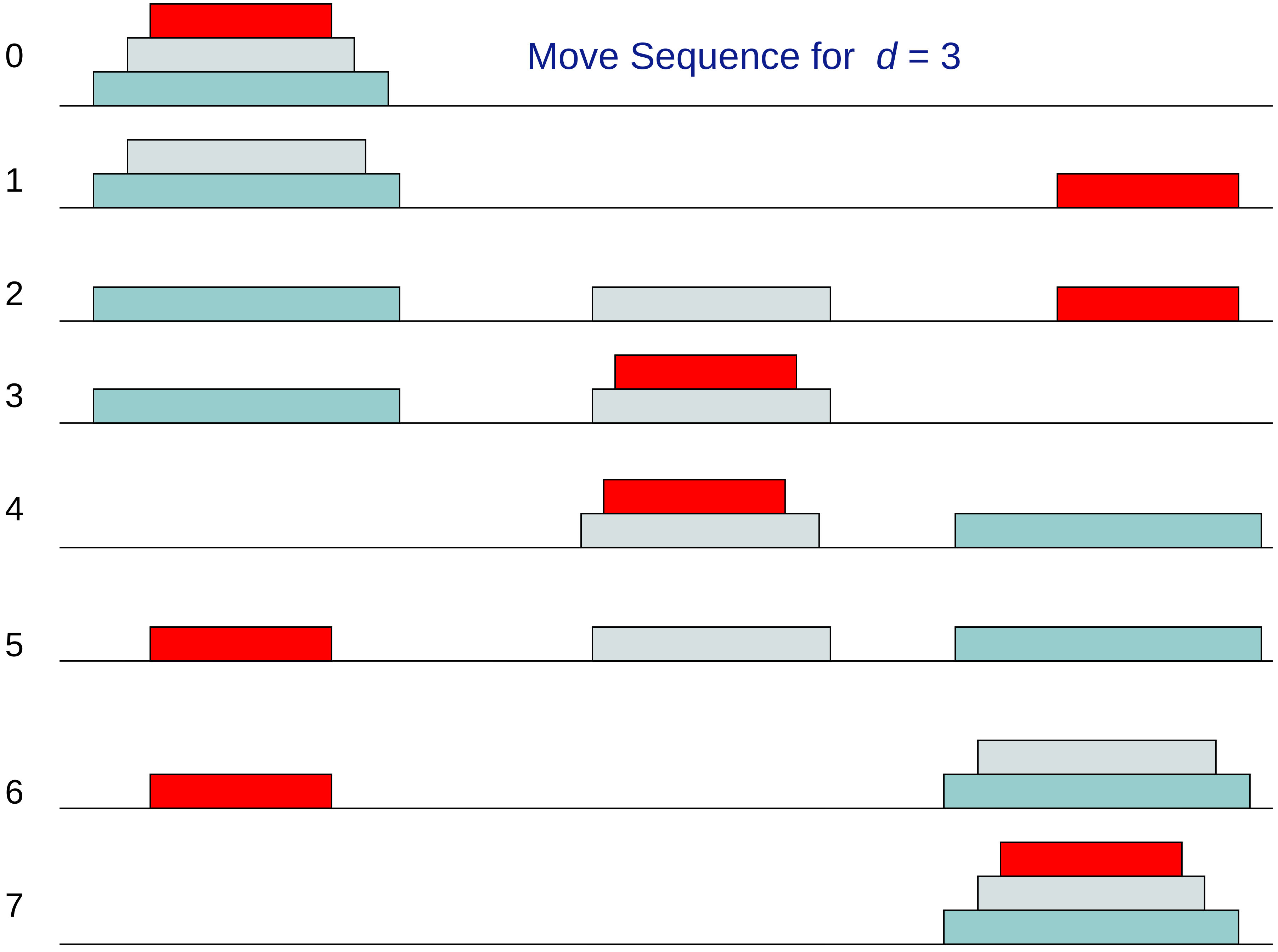
Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

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

Move Sequence for d = 3



The Tower of Hanoi Puzzle

- Here's how to find the number of moves needed to transfer larger numbers of disks from post A to post C, when M = the number of moves needed to transfer n-1 disks from post A to post C:
- for **1 disk** it takes 1 move to transfer 1 disk from post A to post C;
- for **2 disks**, it will take 3 moves: 2M + 1 = 2(**1**) + 1 = **3**
- for **3 disks**, it will take 7 moves: 2M + 1 = 2(**3**) + 1 = **7**
- for **4 disks**, it will take 15 moves: 2M + 1 = 2(**7**) + 1 = **15**
- for **5 disks**, it will take 31 moves: 2M + 1 = 2(**15**) + 1 = **31**
- for **6 disks**... ?

Explicit Pattern

- Number of Disks Number of Moves

Number of Disks	Number of Moves
1	1
2	3
3	7
4	15
5	31
6	63

Powers of two help reveal the pattern:

- **Number of Disks (n)** **Number of Moves**

Number of Disks (n)	Number of Moves
1	$2^1 - 1 = 2 - 1 = 1$
2	$2^2 - 1 = 4 - 1 = 3$
3	$2^3 - 1 = 8 - 1 = 7$
4	$2^4 - 1 = 16 - 1 = 15$
5	$2^5 - 1 = 32 - 1 = 31$
6	$2^6 - 1 = 64 - 1 = 63$

Fascinating fact

So the formula for finding the number of steps it takes to transfer *n* disks from post *A* to post *C* is:

$$2^n - 1$$

Solving recurrence for number of moves

M(*n*) = 2M(*n*-1) + 1, M(1) = 1

M(n) = 2M(n-1) + 1

= 2(2M(n-2) + 1) + 1 = 2^2*M(n-2) + 2^1 + 2^0

= 2^2*(2M(n-3) + 1) + 2^1 + 2^0

= 2^3*M(n-3) + 2^2 + 2^1 + 2^0

= …

= 2^(n-1)*M(1) + 2^(n-2) + … + 2^1 + 2^0

= 2^(n-1) + 2^(n-2) + … + 2^1 + 2^0

= 2^n - 1

Mathimatical analysis of Recursive algorithms

I.Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

Substitution method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

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

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

Example of substitution

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

$$\leq 4c(n/2)^3 + n$$

$$= (c/2)n^3 + n$$

$$= cn^3 - ((c/2)n^3 - n)$$ *desired – residual*

$$\leq cn^3$$ *desired*

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

residual

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

Example (continued)

- We must also handle the initial conditions, that is, ground the induction with base cases.
- ***Base:*** $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
- For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.

This bound is not tight!

A tighter upper bound?

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

A tighter upper bound?

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

Assume that $T(k) \le ck^2$ for $k <$ n: $T(n) = 4T(n/2) + n$

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

A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \le ck^2$ for $k < n$: $T(n) = 4T(n/2) + n$
  - $\le 4c(n/2)^2 + n$
  - $= \cancel{cn^2} + n$
  - $= O(n^2)$ ***Wrong!***

We must prove the I.H.

A tighter upper bound?

- We shall prove that $T(n) = O(n^2)$.
- Assume that $T(k) \leq ck^2$ for $k < n$: $T(n) = 4T(n/2) + n$
  - $\leq 4c(n/2)^2 + n$
  - $= \cancel{cn}^2 + n$
  - $= O(n^2)$ ***Wrong!*** We must prove the I.H.
  - $= cn^2 - (-n)$ [desired – residual]

    $\leq cn^2$ for ***no*** choice of $c > 0$. Lose!

A tighter upper bound!

IDEA: Strengthen the inductive hypothesis.

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

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

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 $k < n$.

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &= 4\big(c_1(n/2)^2 - c_2(n/2)\big) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &= c_1 n^2 - c_2 n - (c_2 n - n) \\ &\le c_1 n^2 - c_2 n \quad \text{if } c_2 \ge 1. \end{aligned}$$

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 $k < n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&= 4\big(c_1(n/2)^2 - c_2(n/2)\big) + n \\
&= c_1 n^2 - 2c_2 n + n \\
&= c_1 n^2 - c_2 n - (c_2 n - n) \\
&\le c_1 n^2 - c_2 n \quad \text{if } c_2 \ge 1.
\end{aligned}
$$

Pick c_1 big enough to handle the initial conditions.

II. Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method.

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

$$n^2$$

$$T(n/4) \qquad\qquad T(n/2)$$

Example of recursion tree

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

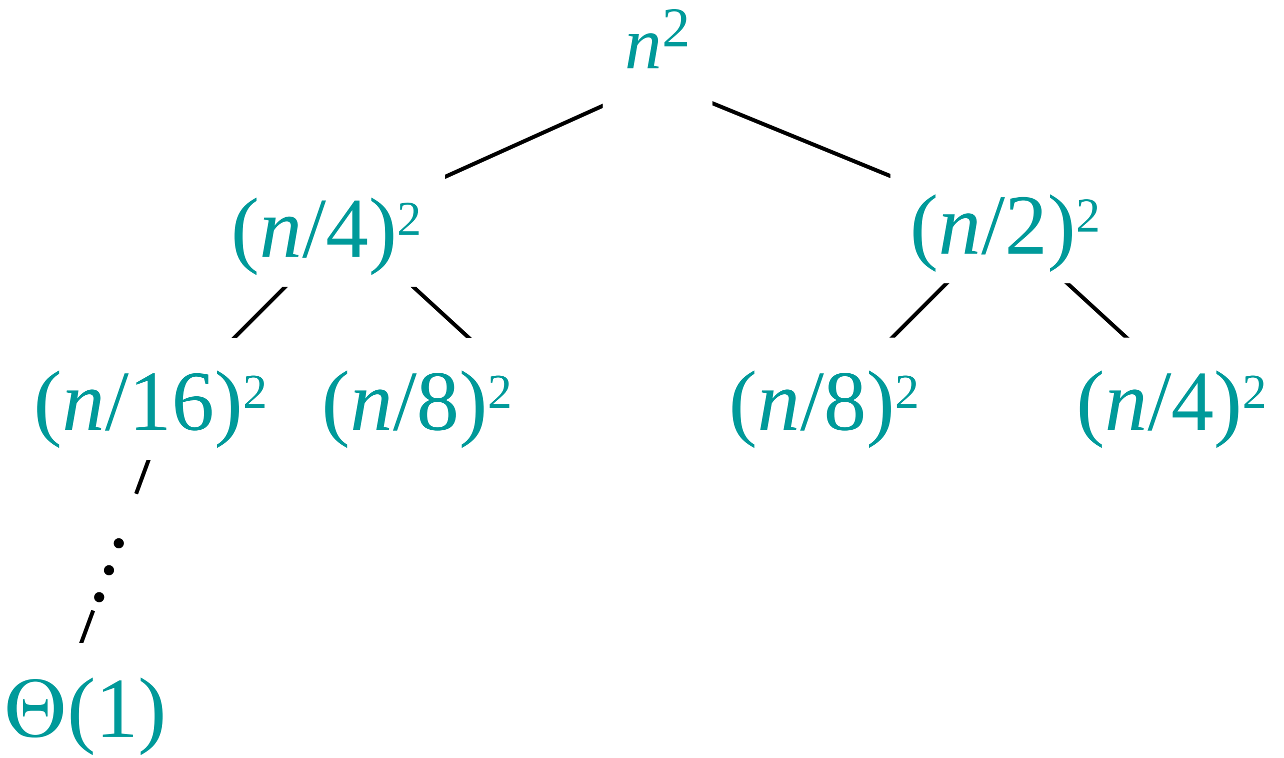
$$n^2$$

$$(n/4)^2 \qquad\qquad (n/2)^2$$

$$T(n/16) \quad T(n/8) \qquad\qquad T(n/8) \quad T(n/4)$$

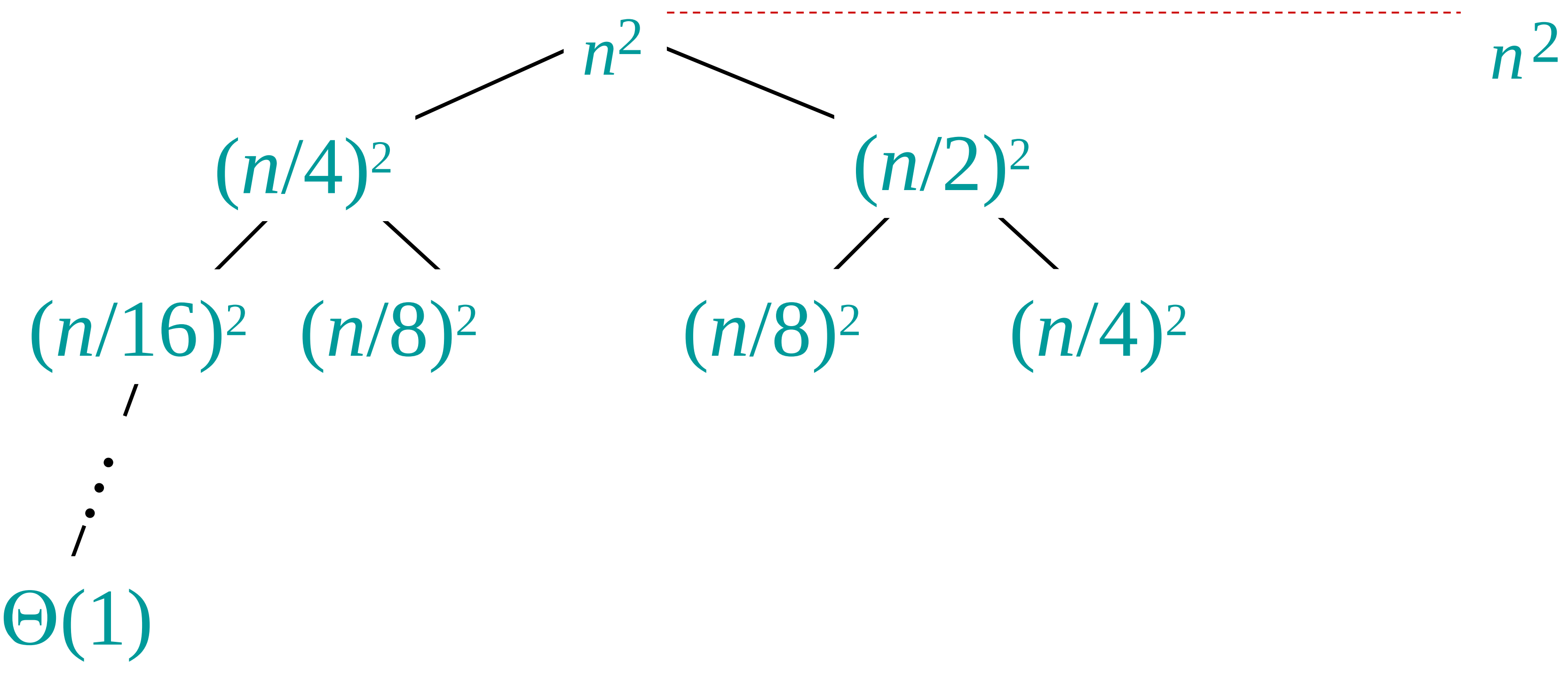
Example of recursion tree

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



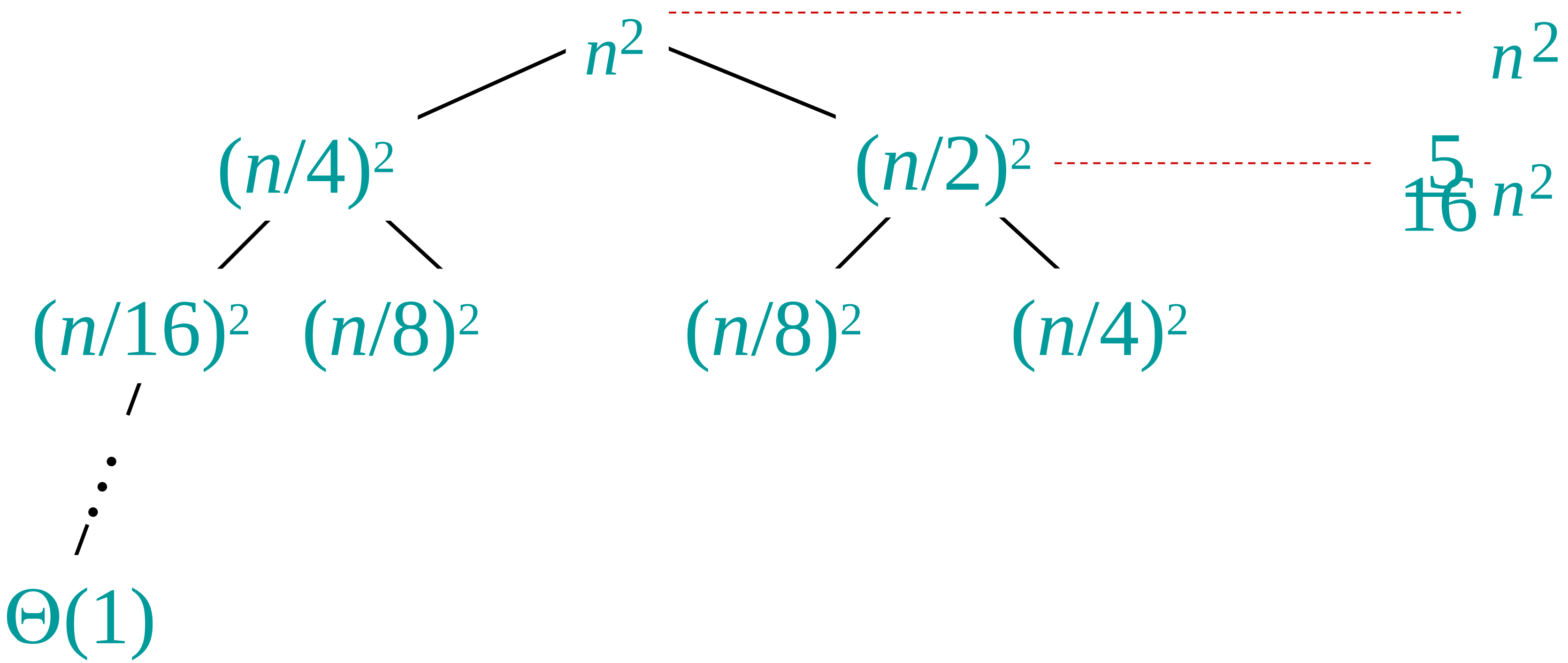
Example of recursion tree

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



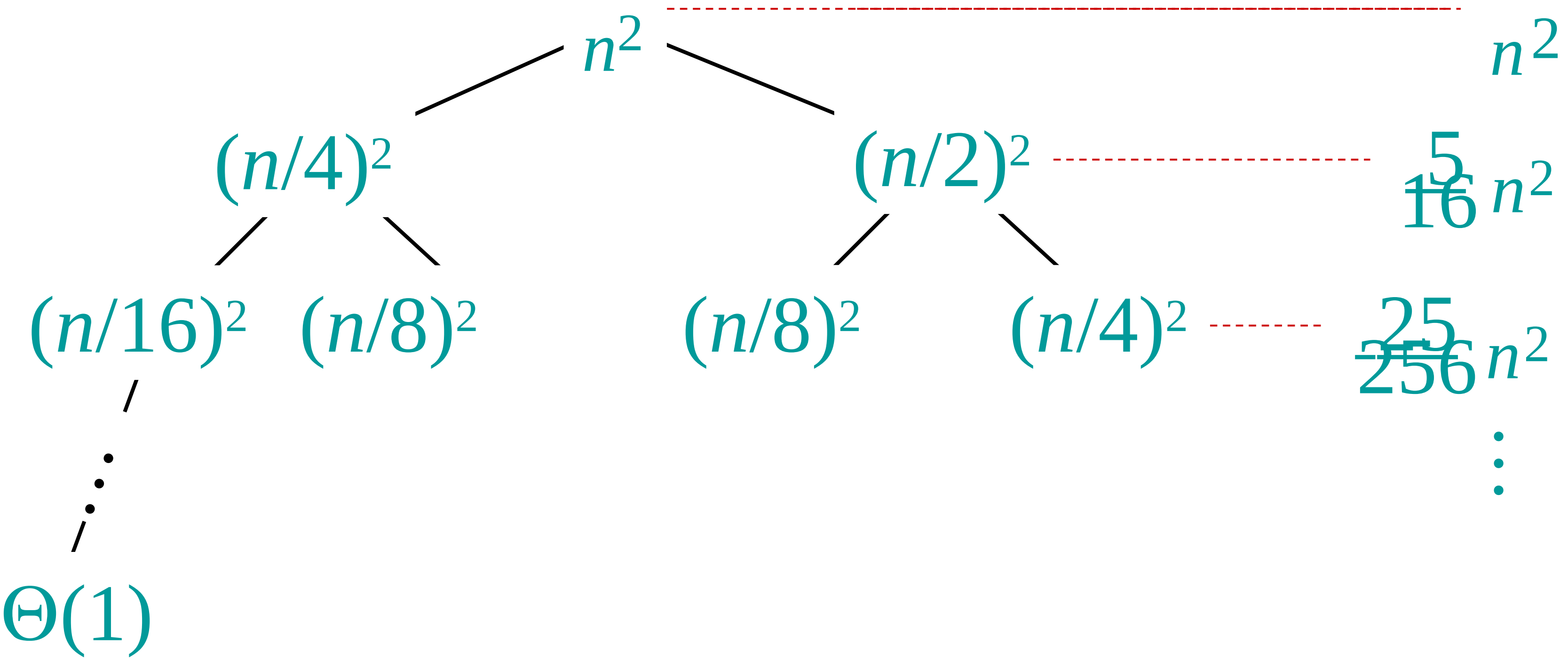
Example of recursion tree

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

n^2 — level sum: n^2

$(n/4)^2 \quad (n/2)^2$ — level sum: $\frac{5}{16} n^2$

$(n/16)^2 \quad (n/8)^2 \quad (n/8)^2 \quad (n/4)^2$ — level sum: $\frac{25}{256} n^2$

$\vdots$

$\Theta(1)$

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

$$= \Theta(n^2) \quad \textit{geometric series}$$

III. 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.

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

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

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

- $f(n)$ and $n^{\log_b a}$ grow

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 $af(n/b) \le cf(n)$ for some constant $c < 1$.

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

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 $\varepsilon = 1$.

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

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: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

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

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

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

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

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

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

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

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

Idea of master theorem

Recursion tree:

$$f(n)$$

$$a$$

$$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$$

$$a$$

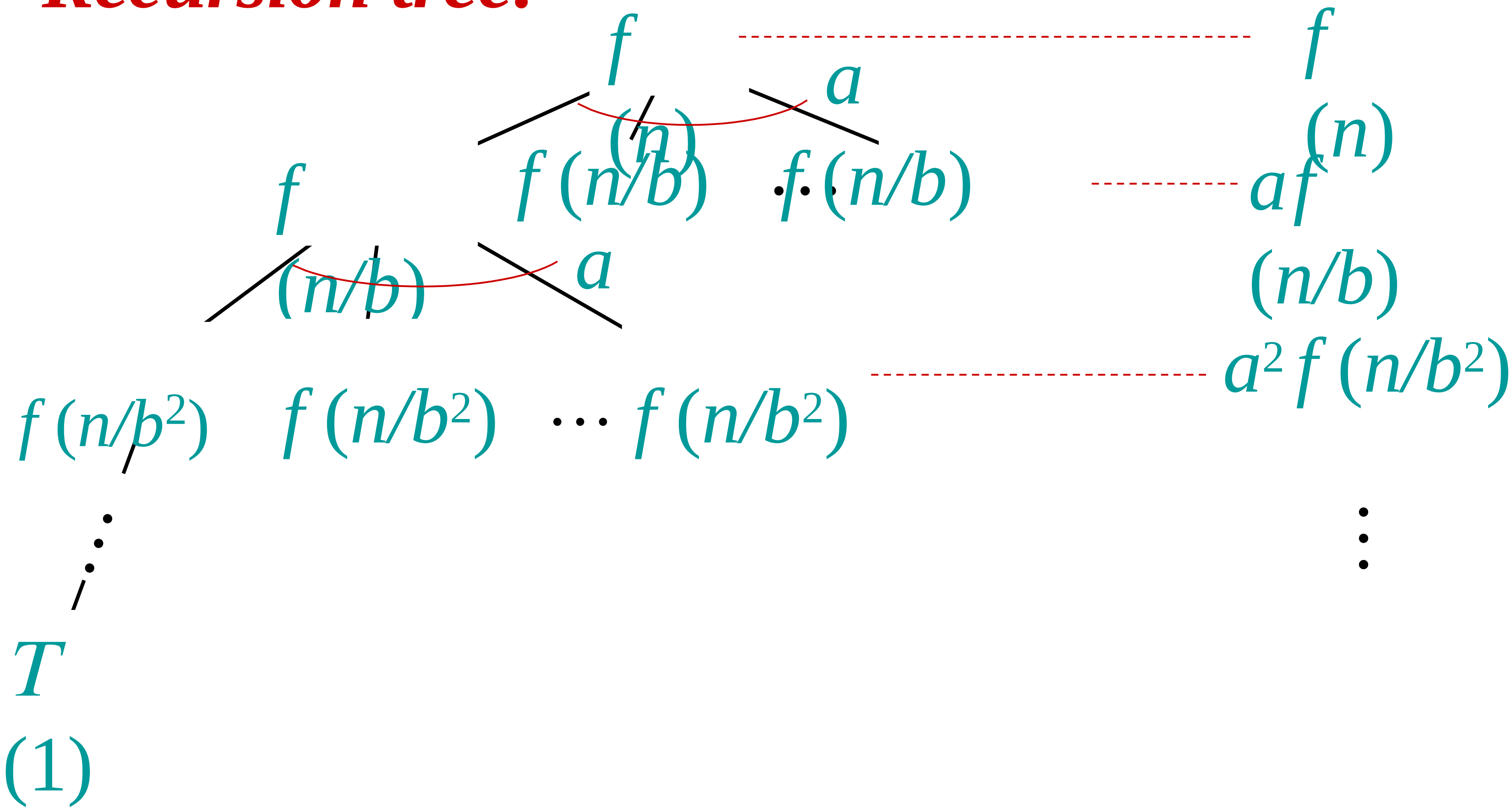
$$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$$

$$\vdots$$

$$T(1)$$

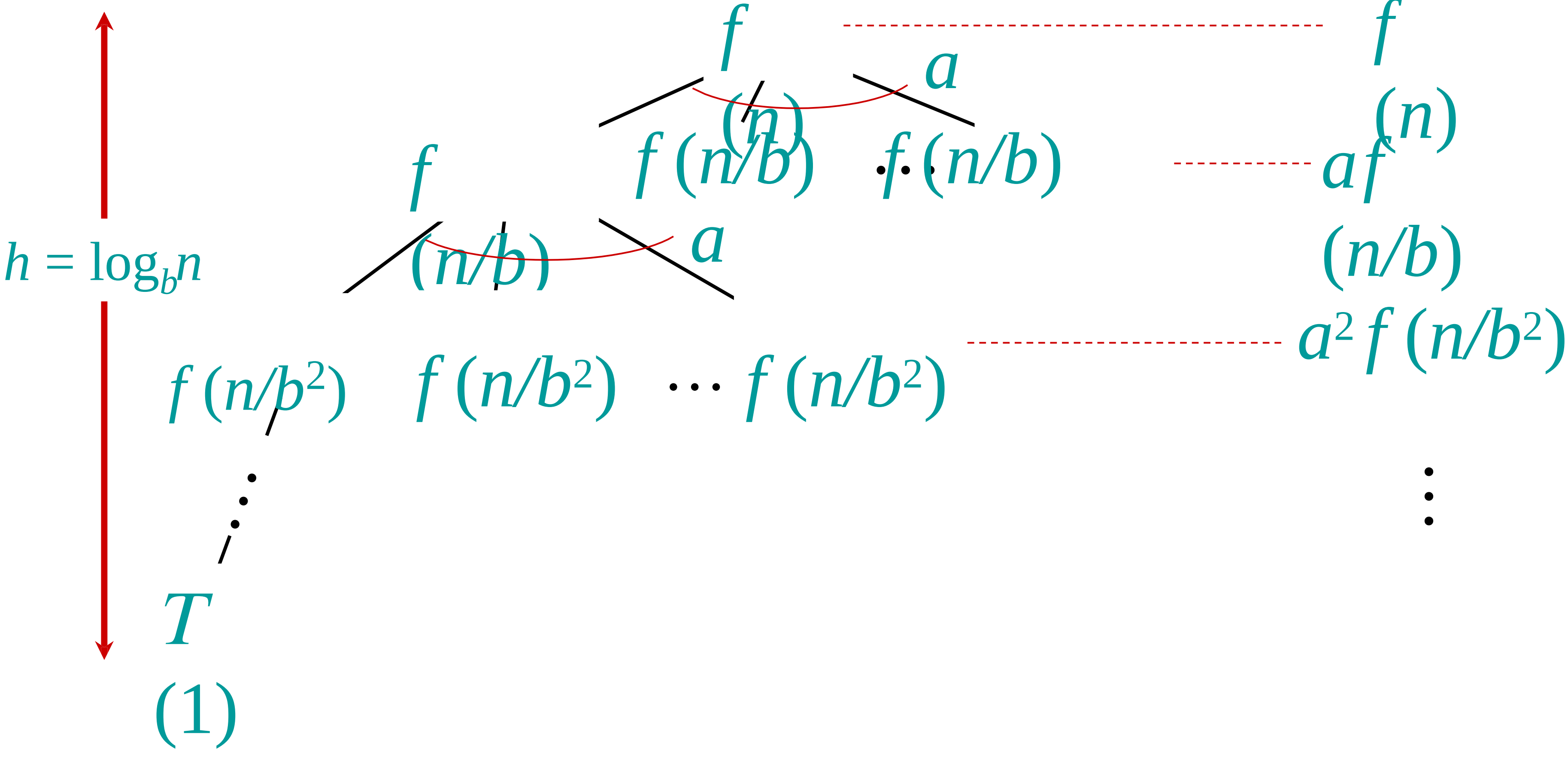
Idea of master theorem

Recursion tree:



Idea of master theorem

Recursion tree:



Idea of master theorem

Recursion tree:

$$f(n)$$

a

$$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$$

a

$$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$$

$\vdots$

$$T(1)$$

$h = \log_b n$

#leaves $= a^h$

$= a^{\log_b n}$

$= n^{\log_b a}$

$$f(n)$$

$$af(n/b)$$

$$a^2 f(n/b^2)$$

$$\vdots$$

$$n^{\log_b a} T(1)$$

Idea of master theorem

Recursion tree:

$f(n)$

a

$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$

a

$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$

$\vdots$

$T(1)$

$h = \log_b n$

$f(n)$

$af(n/b)$

$a^2 f(n/b^2)$

$\vdots$

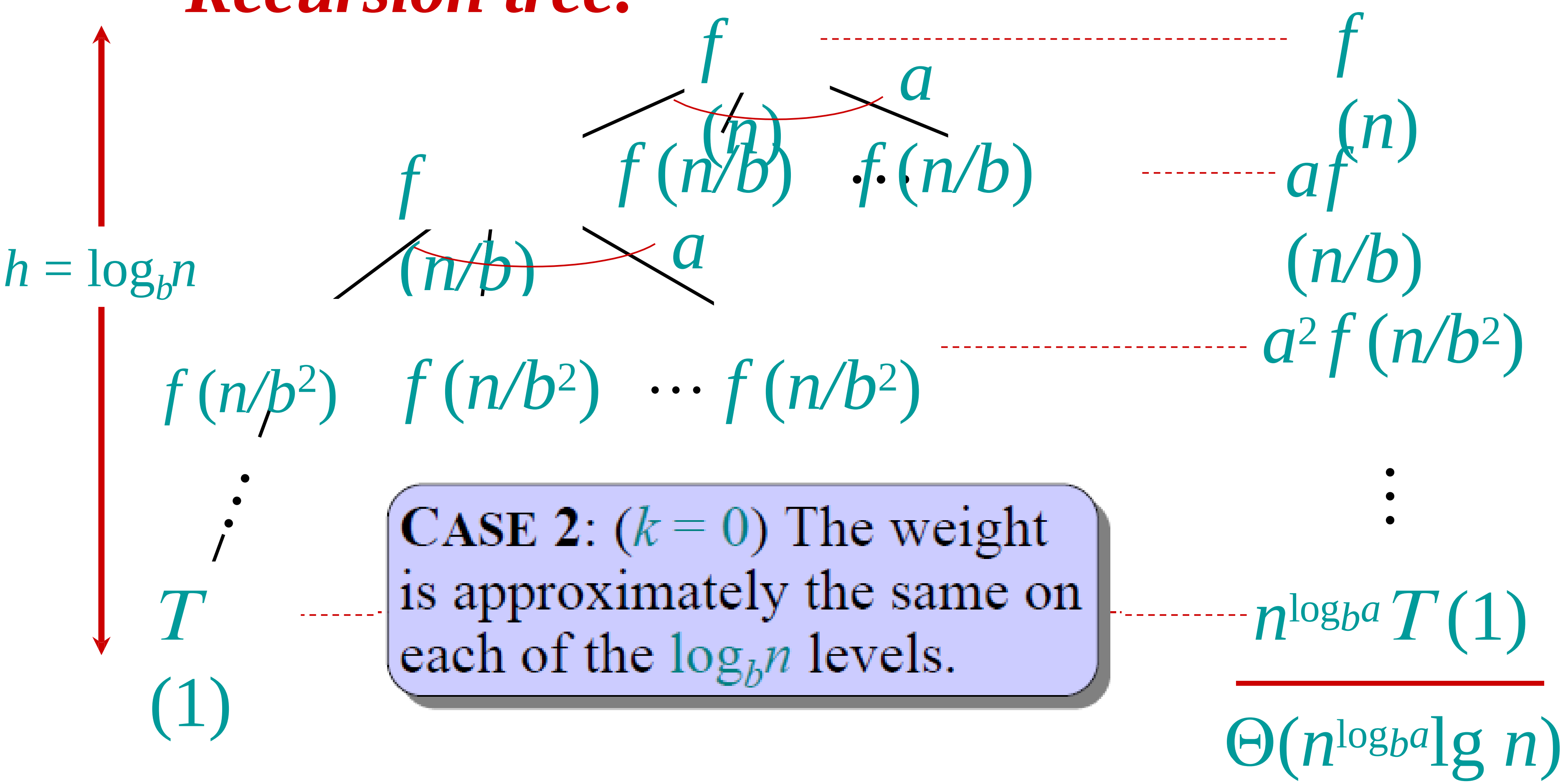
$n^{\log_b a} T(1)$

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

CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

Idea of master theorem

Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

Idea of master theorem

Recursion tree:

$f(n)$ — a

$f(n/b) \quad f(n/b) \quad \cdots \quad f(n/b)$ — a

$f(n/b^2) \quad f(n/b^2) \quad \cdots \quad f(n/b^2)$

$\vdots$

$T(1)$

$h = \log_b n$

$f(n)$

$a f(n/b)$

$a^2 f(n/b^2)$

$\vdots$

$n^{\log_b a} T(1)$

$\Theta(f(n))$

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

Appendix: geometric series

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

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

Return to last slide viewed.