

# Week 2: Part 1

---

Recursion, Recurrences & Running time

# Chapter 4

---

- Divide and conquer VS iterative algorithms
- Recursion
- Solving Recurrences
- Binary Search
- Merge Sort
- Towers of Hanoi

# Recall from Week 1

---

- Asymptotic Analysis:  $O$ ,  $\Omega$ ,  $\Theta$
- Used to compare functions that represent the running times of different algorithms that can be used to solve a problem.
- How did we get the functions

# Iterative Algorithm Analysis

---

<code>for (i=1; i&lt;=n*n; i++)</code>	Executed $n*n$ times
<code>for (j=0; j&lt;i; j++)</code>	Executed $\leq n*n$ times
<code>sum++;</code>	$O(1)$

Exact # of times `sum++` is executed:

$$\begin{aligned}\sum_{i=1}^{n^2} i &= \frac{n^2(n^2 + 1)}{2} \\ &= \frac{n^4 + n^2}{2} \\ &\in \Theta(n^4)\end{aligned}$$

# The Divide and Conquer Approach

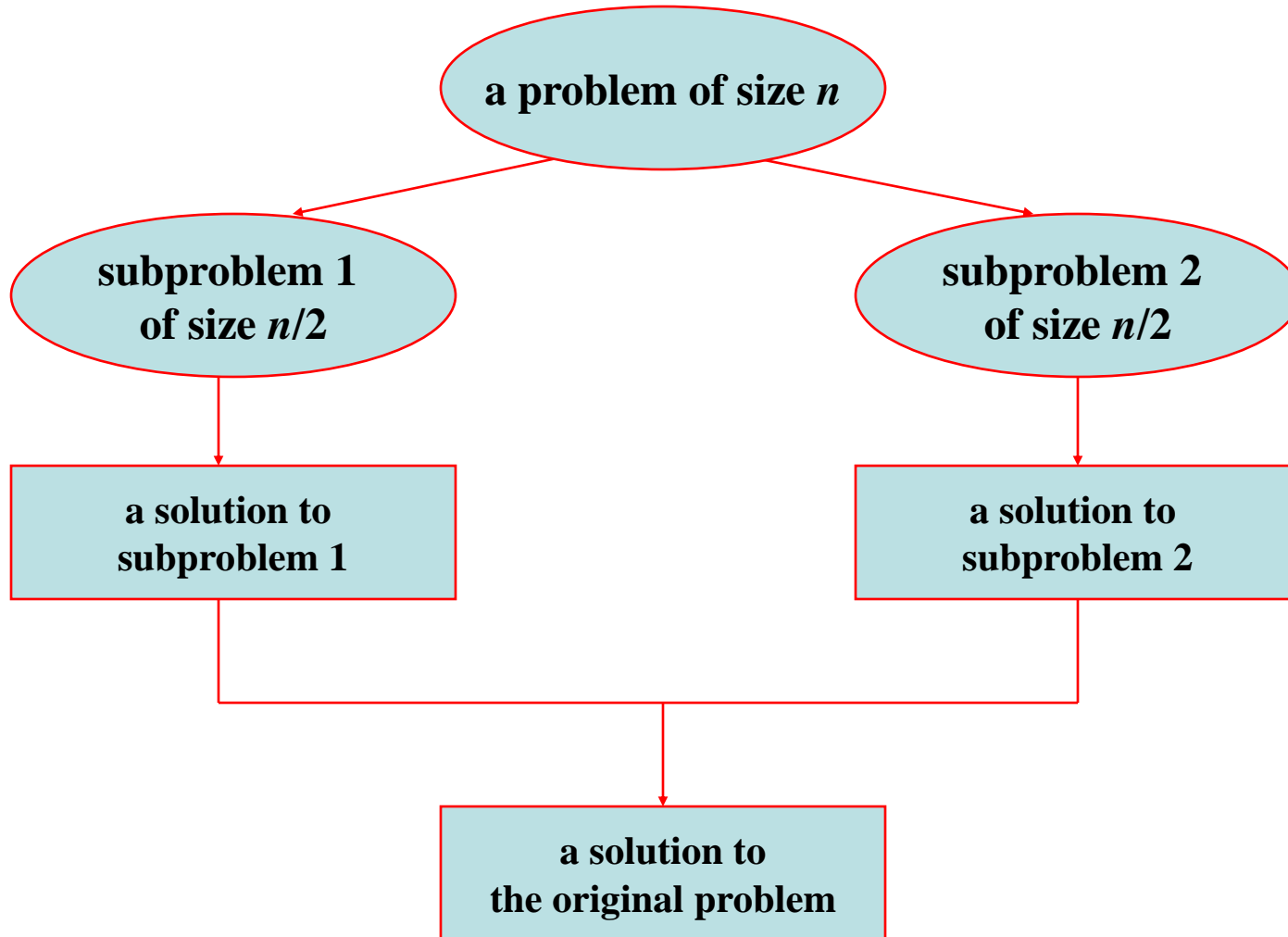
---

The most well known algorithm design strategy:

1. **Divide** the problem into two or more smaller subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solutions for the original problem.

# A Typical Divide and Conquer Case

---



# Recurrences and Running Time

---

- An equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = T\left(\frac{n}{4}\right) + 1$$

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves  $n$

# Merge-Sort Example

---

- Merge-sort on an input sequence  $S$  with  $n$  elements consists of three steps:
  - Divide: partition  $S$  into two sequences  $S_1$  and  $S_2$  of about  $n/2$  elements each
  - conquer: recursively sort  $S_1$  and  $S_2$
  - combine: merge  $S_1$  and  $S_2$  into a unique sorted sequence

**Algorithm** *mergeSort*( $S, c$ )

**Input** sequence  $S$  with  $n$  elements, comparator  $c$

**Output** sequence  $S$  sorted according to  $c$

**if**  $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

*mergeSort*( $S_1, c$ )

*mergeSort*( $S_2, c$ )

$S \leftarrow merge(S_1, S_2)$



# Recurrence Equation

---

- The combine step of merge-sort consists of merging two sorted sequences, each with  $n/2$  elements takes at most  $cn$  steps, for some constant  $c$ .
- Likewise, the basis case ( $n < 2$ ) will take at most  $b$  steps.
- Therefore, if we let  $T(n)$  denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + cn & \text{if } n \geq 2 \end{cases}$$

- We can analyze the running time of merge-sort by finding a **closed form solution** to the above equation. *That is, a solution that has  $T(n)$  only on the left-hand side.*

# Merge-Sort

---

- 1. Divide:** Trivial.
- 2. Conquer:** Recursively sort 2 subarrays.
- 3. Combine:** Linear-time merge.

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

The diagram illustrates the recurrence relation  $T(n) = 2T(n/2) + \Theta(n)$ . The terms  $2$ ,  $T(n/2)$ , and  $\Theta(n)$  are highlighted in yellow. Three arrows point from descriptive text below to these terms: an arrow from "# subproblems" points to the coefficient 2; an arrow from "subproblem size" points to the  $T(n/2)$  term; and an arrow from "work dividing and combining" points to the  $\Theta(n)$  term.

Closed form:  $T(n) = \Theta(n \lg n)$

# Binary Search

---

Find an element in a sorted array:

- 1. Divide:* Check middle element.
- 2. Conquer:* Recursively search 1 subarray.
- 3. Combine:* Trivial.

# Binary Search

---

Find an element in a sorted array:

- 1. Divide:* Check middle element.
- 2. Conquer:* Recursively search 1 subarray.
- 3. Combine:* Trivial.

*Example:* Find 9

3    5    7    8    9    12    15

# Binary Search

---

Find an element in a sorted array:

- 1. *Divide*:** Check middle element.
- 2. *Conquer*:** Recursively search **1** subarray.
- 3. *Combine*:** Trivial.

***Example:*** Find **9**

3    5    7    8    9    12    15

# Binary Search

---

Find an element in a sorted array:

- 1. Divide:* Check middle element.
- 2. Conquer:* Recursively search 1 subarray.
- 3. Combine:* Trivial.

*Example:* Find 9

3

5

7

8

9

12

15

# Binary Search

---

Find an element in a sorted array:

- 1. Divide:* Check middle element.
- 2. Conquer:* Recursively search 1 subarray.
- 3. Combine:* Trivial.

*Example:* Find 9

3      5      7      8      9      12      15

# Binary Search

---

Find an element in a sorted array:

- 1. Divide:* Check middle element.
- 2. Conquer:* Recursively search 1 subarray.
- 3. Combine:* Trivial.

*Example:* Find 9

3    5    7    8    9    12    15

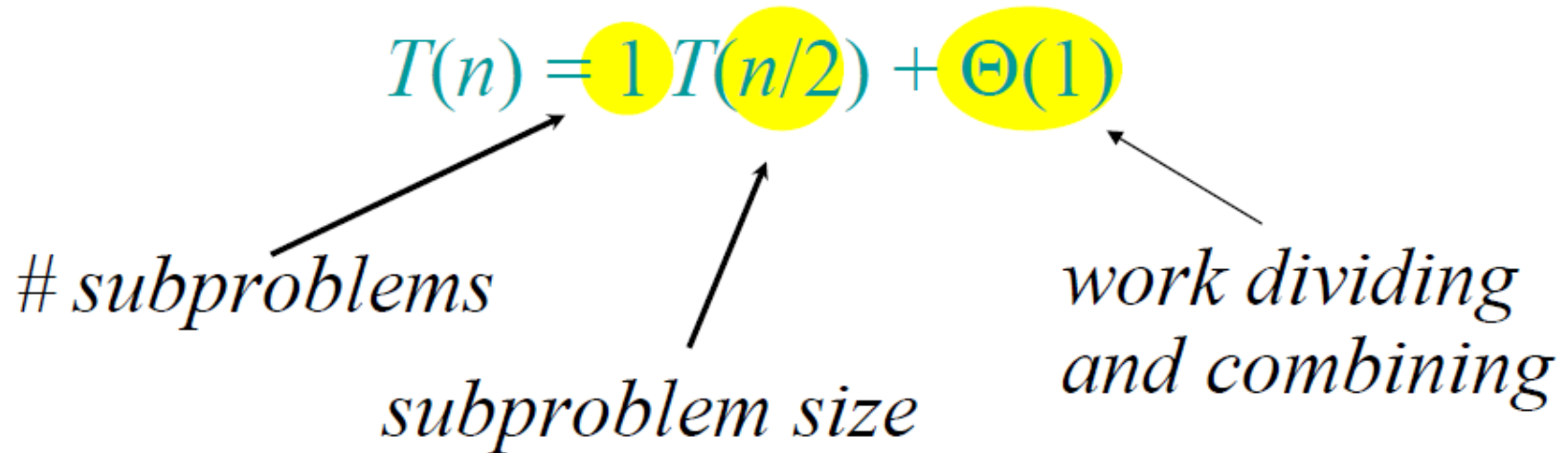


# Binary Search

---

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

*# subproblems*      *subproblem size*      *work dividing and combining*



Closed form:  $T(n) = \Theta(\lg n)$

# Power of a Number

---

**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

**Naive algorithm:**  $\Theta(n)$ .

Counting the number of operations which are multiplications

**Example:**  $a^n = a * a * \dots * a$

**Example:**  $15^9 = 15 * 15 * 15 * 15 * 15 * 15 * 15 * 15 * 15$

# Power of a Number

---

**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

**Divide-and-conquer algorithm:**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

Base cases  $a^0 = 1$  and  $a^1 = a$

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

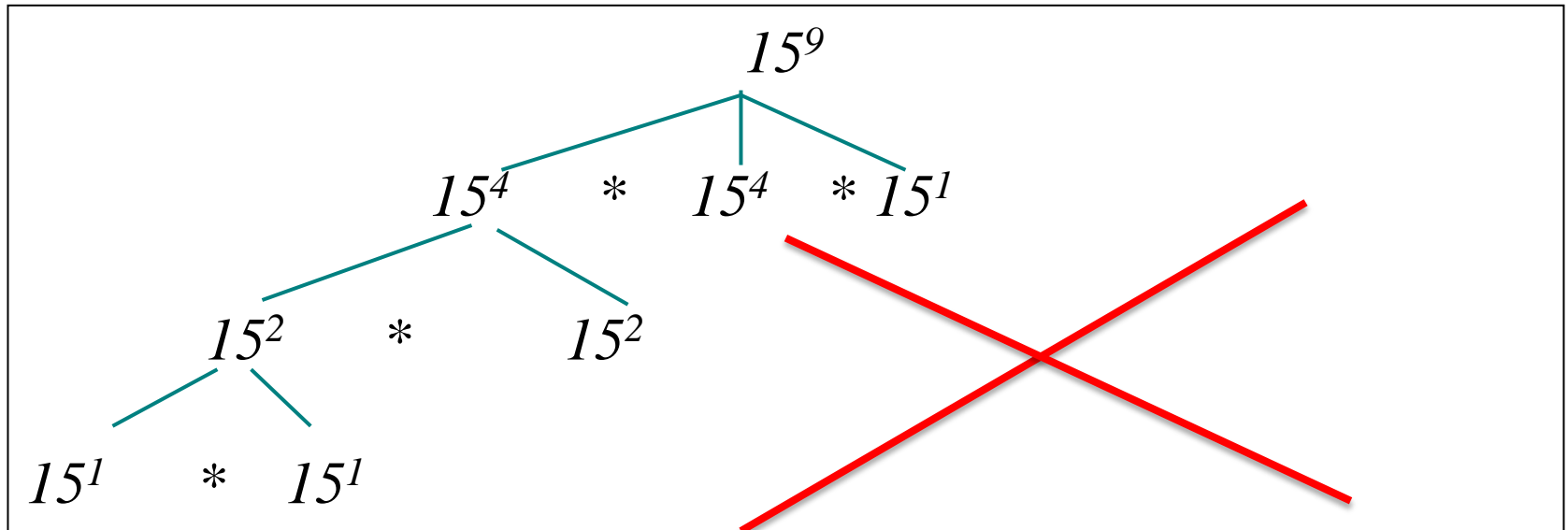
**Problem:** Compute  $a^n$ , where  $n \in \mathbb{N}$ .

---

**Divide-and-conquer algorithm:**

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

Base cases  $a^0 = 1$  and  $a^1 = a$



# Recurrence Relations from Code

---

```
long power (long x, long n) {  
    if(n == 0)  
        return 1;  
    else if(n == 1)  
        return x;  
    else if ((n % 2) == 0){  
        temp = power(x, n/2);  
        return temp*temp;  
    }  
    else {  
        temp = power(x, (n-1)/2)  
        return x * temp* temp;  
    }  
}
```

The recurrence relation is:

$$T(n) = 1$$

*if  $n = 0$  or  $n = 1$*

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

*if  $n > 2$*

Running time  $\Theta(\lg n)$

# Extra Recursion

---

```
long power (long x, long n) {  
    if(n == 0)  
        return 1;  
    else if(n == 1)  
        return x;  
    else if ((n % 2) == 0)  
        return power (x, n/2) * power (x, n/2);  
    else  
        return x * power (x, (n-1)/2) * power (x, (n-1)/2);  
}
```

The recurrence relation is:

$$T(n) = 1$$

$$\text{if } n = 0 \text{ or } n = 1$$

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

$$\text{if } n > 2$$

Running time  $\Theta(n)$

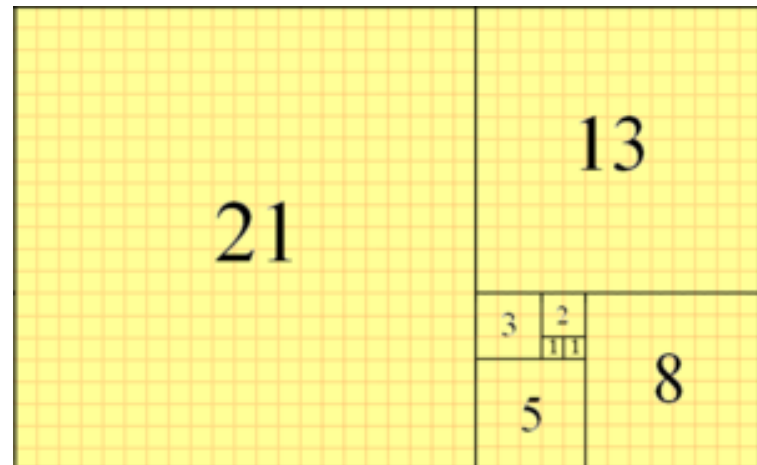
# Fibonacci Numbers

---

## Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0   1   1   2   3   5   8   13   21   34   ...



# Fibonacci

```
long fibonacci (int n) {  
    // Recursively calculates Fibonacci number  
    if( n == 0)  
        return 0;  
    else if( n == 1)  
        return 1;  
    else  
        return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

The recurrence relation is:

$$T(n) = 1 \quad \text{if } n = 0 \text{ or } n = 1$$

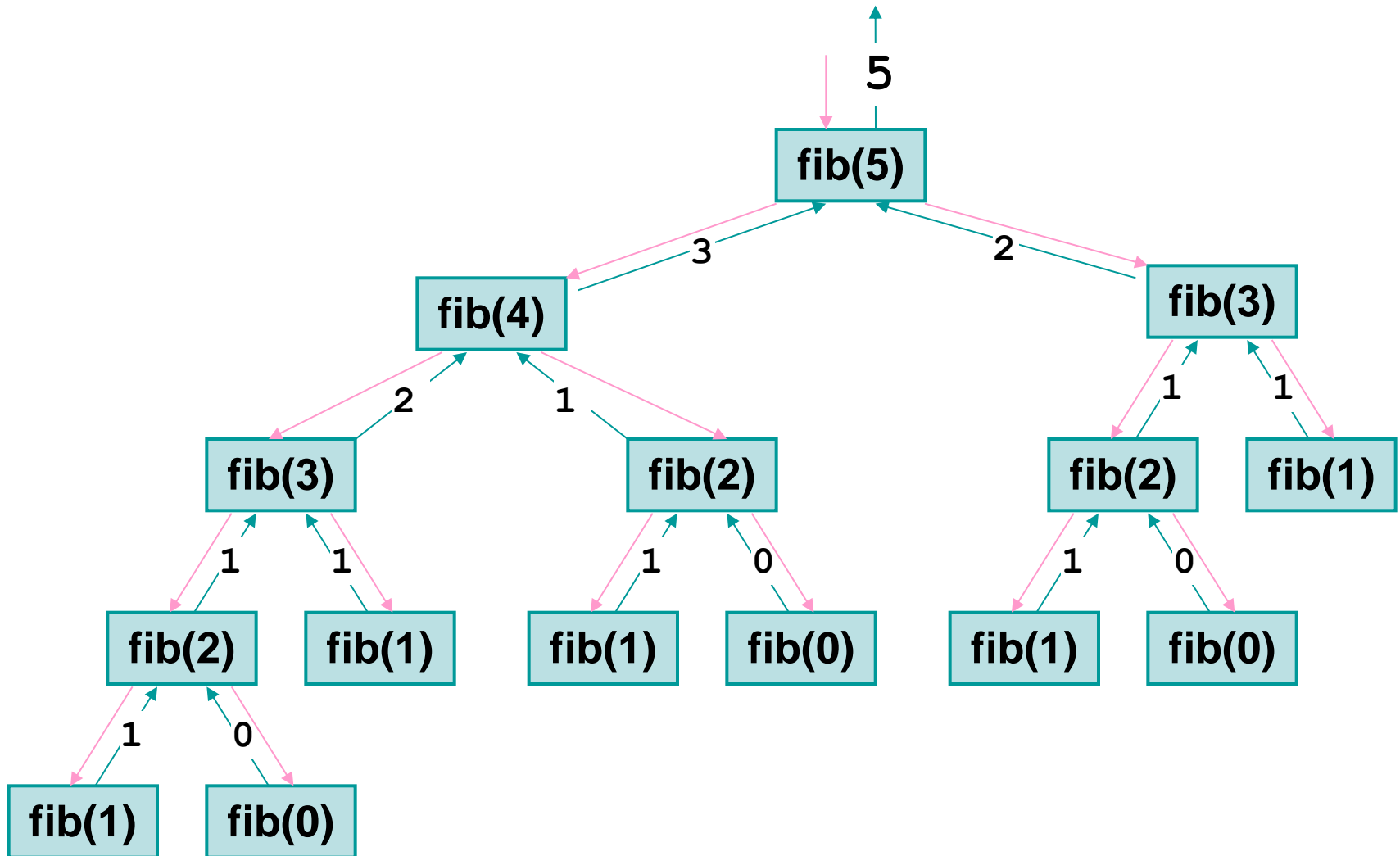
$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{if } n \geq 2$$

$$T(n) = \Theta(\phi^n) \text{ where } \Phi = \frac{1+\sqrt{5}}{2} \text{ golden ratio} = 1.618..$$



# Function Analysis for call `fib(5)`

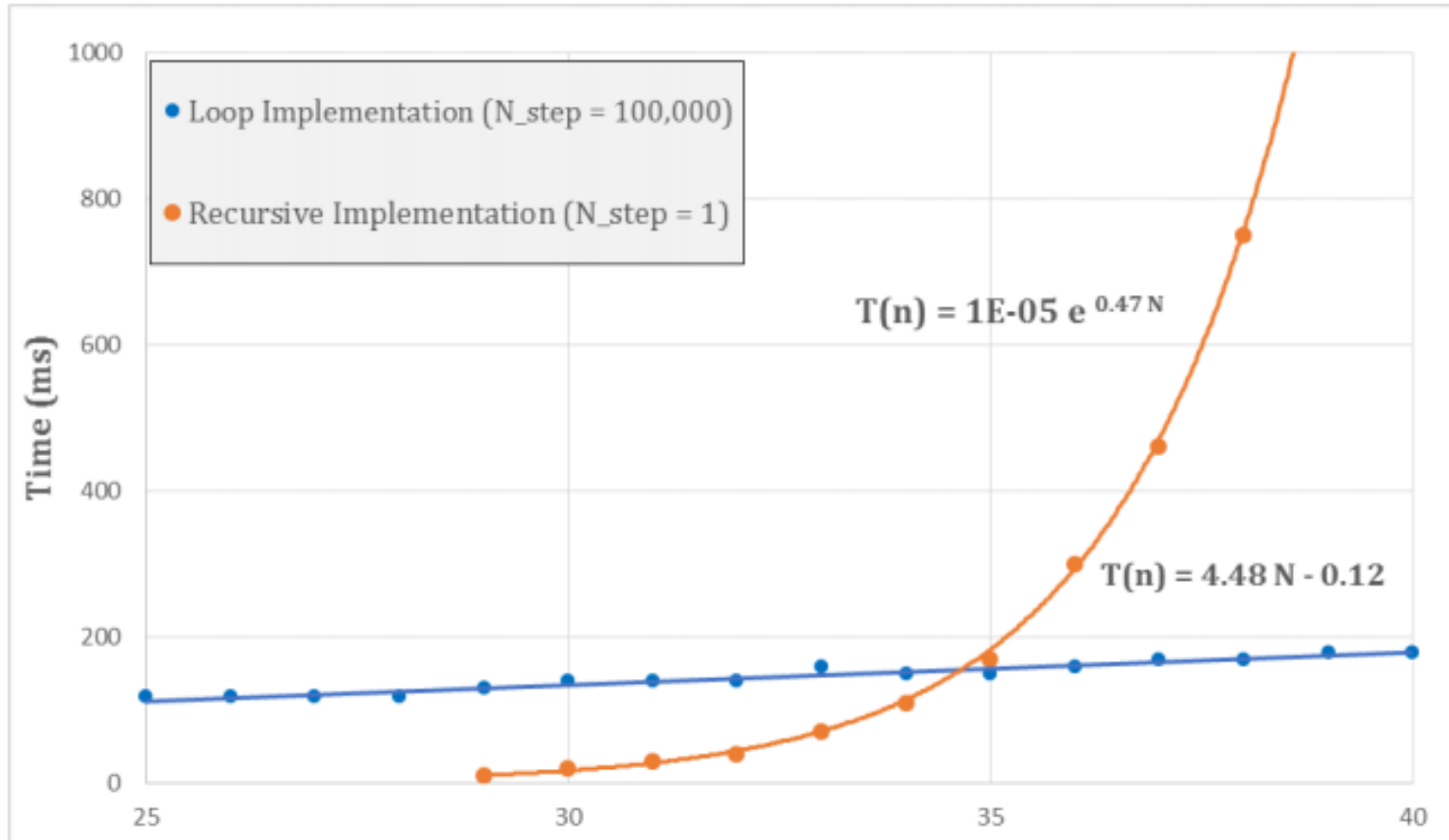
---



# HW 1 Solution

## Fibonacci Performance Comparison:

Note:  $e^{0.47n} \sim 1.60^n$



# Tower of Hanoi

---

- There are three towers
- N gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

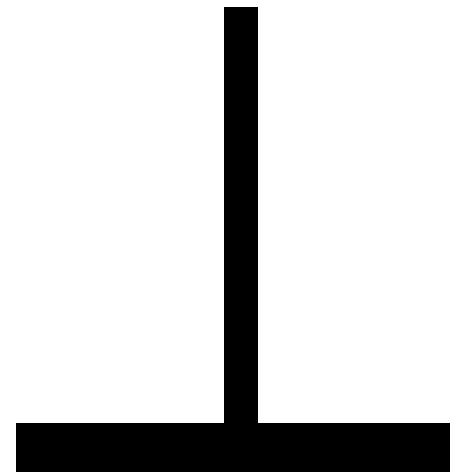
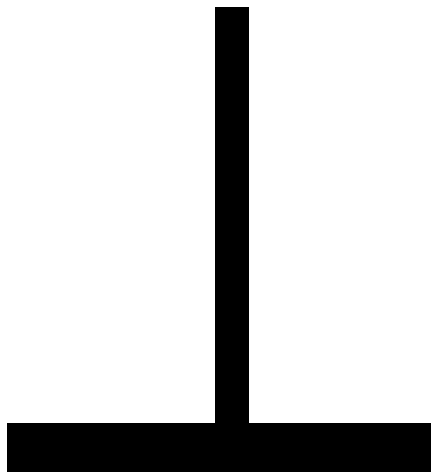
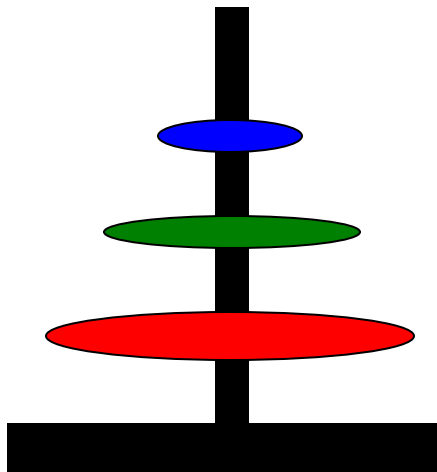
# Tower of Hanoi

---

- The disks must be moved within one week. Assume one disk can be moved in 1 second. Is this possible?
- To create an algorithm to solve this problem, it is convenient to generalize the problem to the “N-disk” problem, where in our case  $N = 64$ .

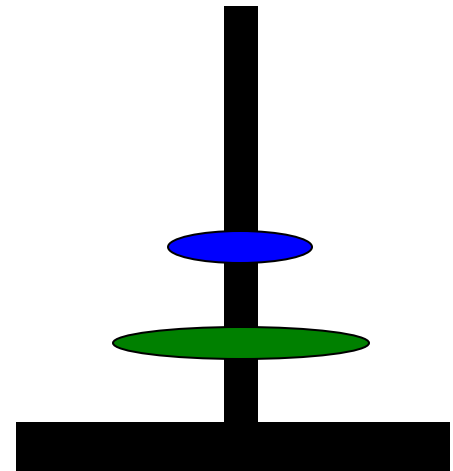
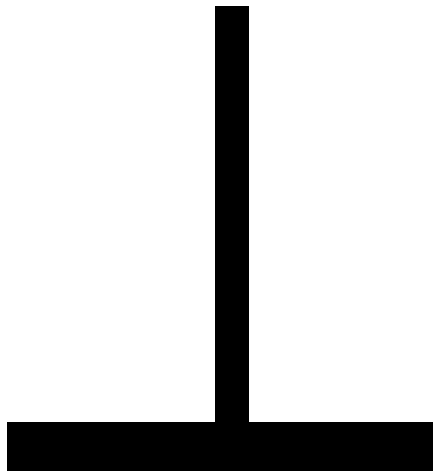
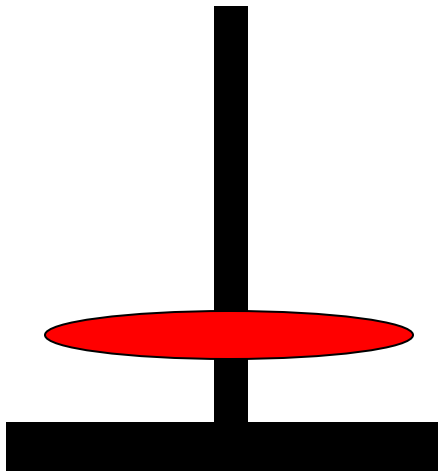
# Recursive Solution

---



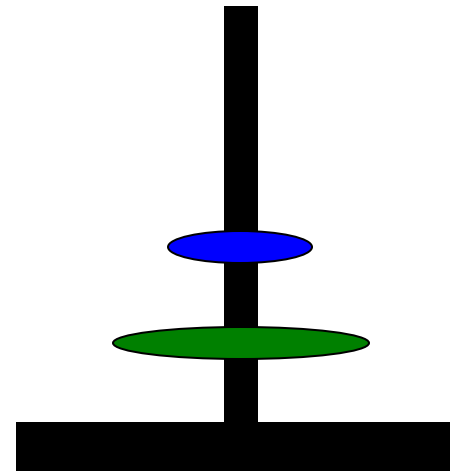
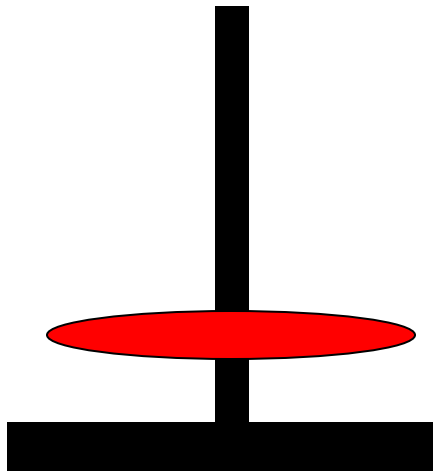
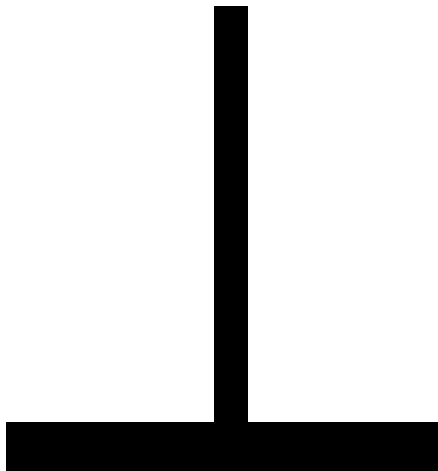
# Recursive Solution

---



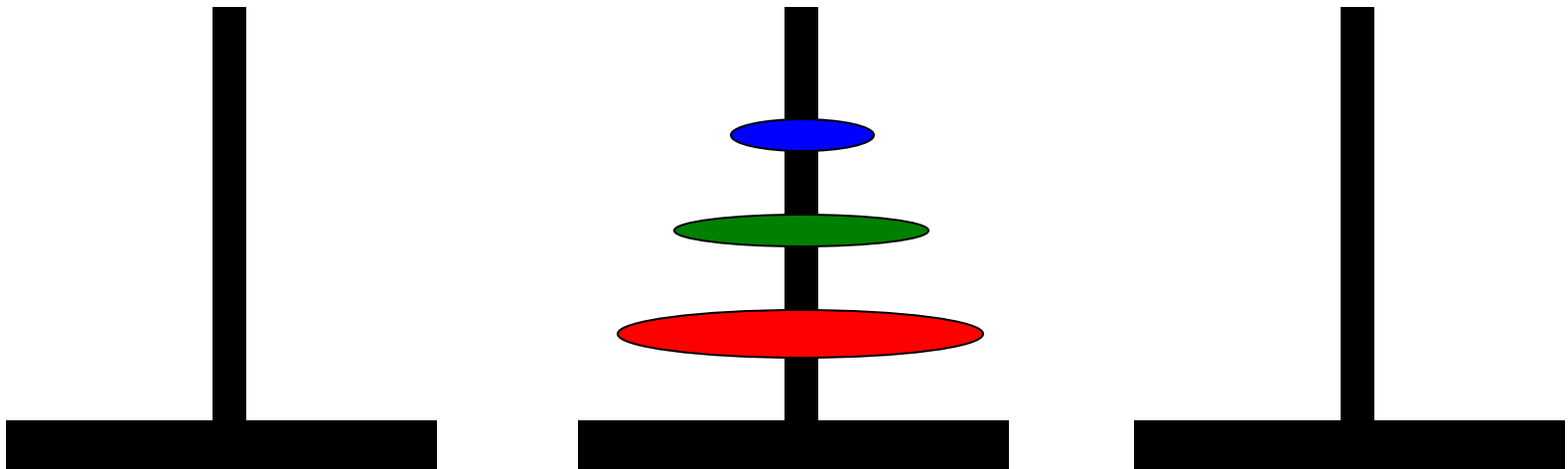
# Recursive Solution

---



# Recursive Solution

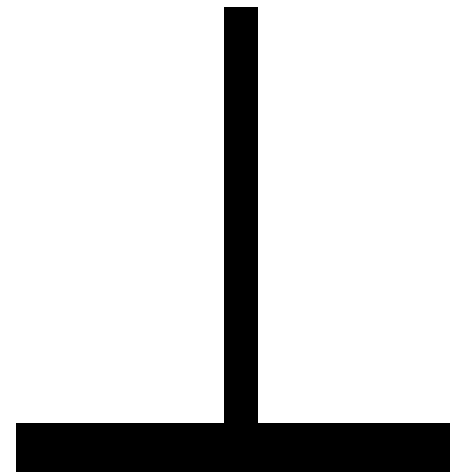
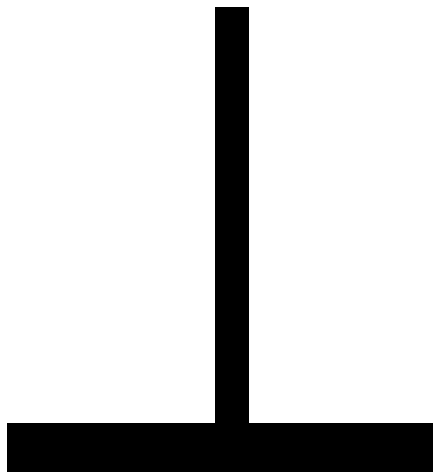
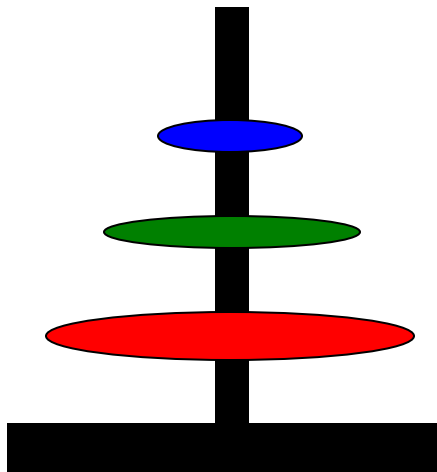
---





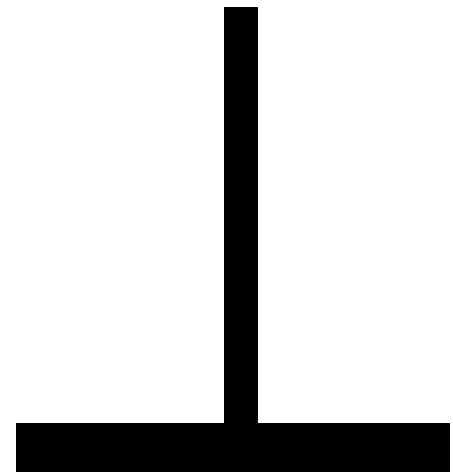
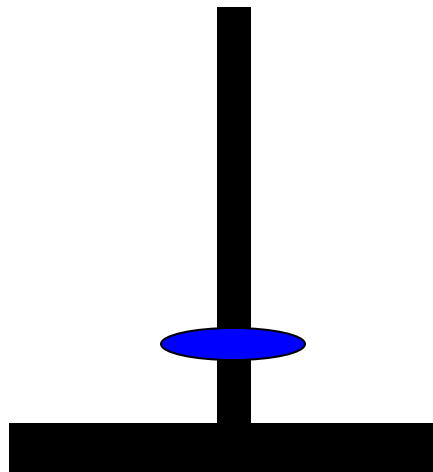
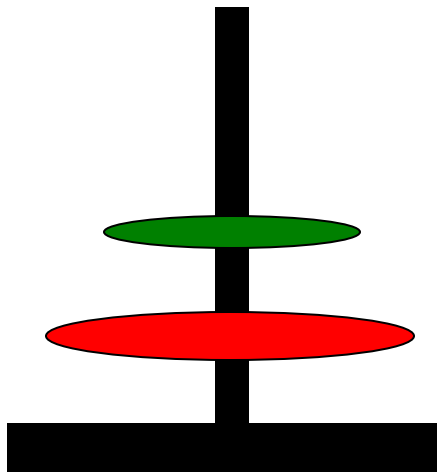
# Tower of Hanoi

---



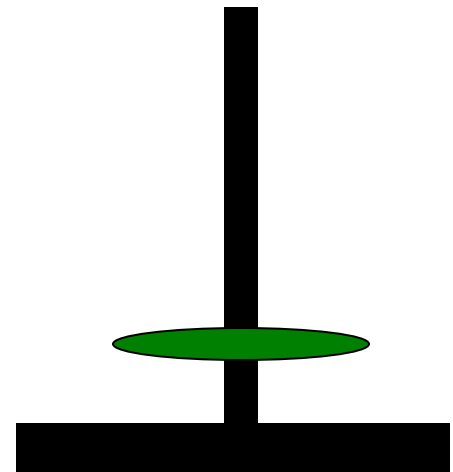
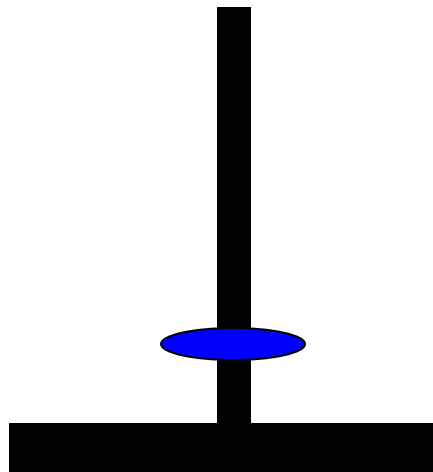
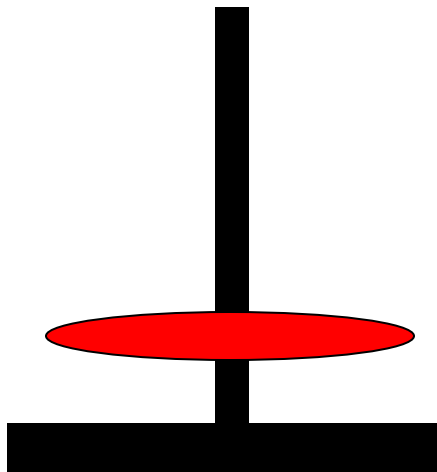
# Tower of Hanoi

---



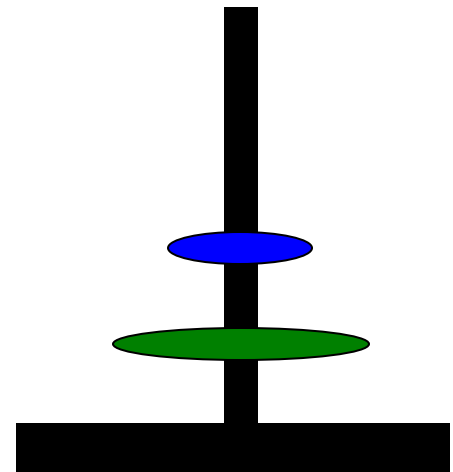
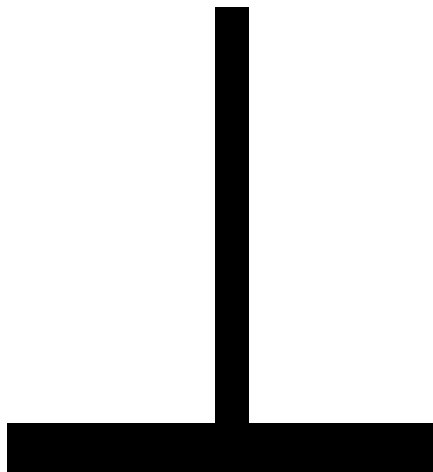
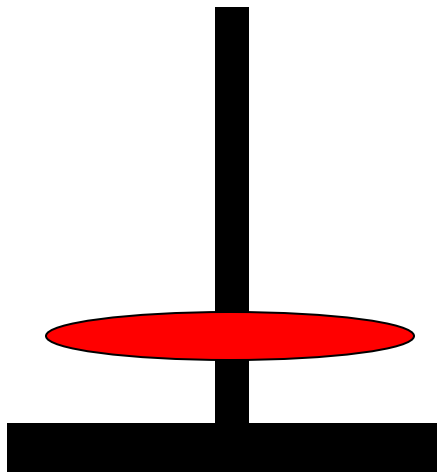
# Tower of Hanoi

---



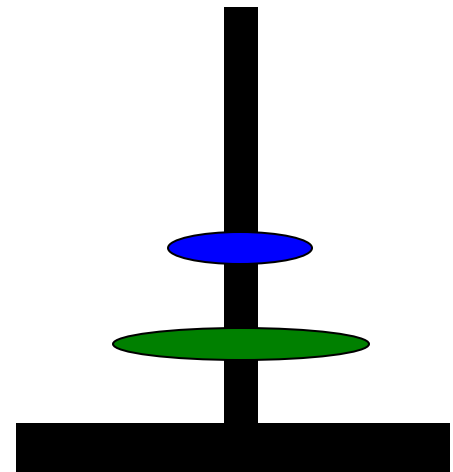
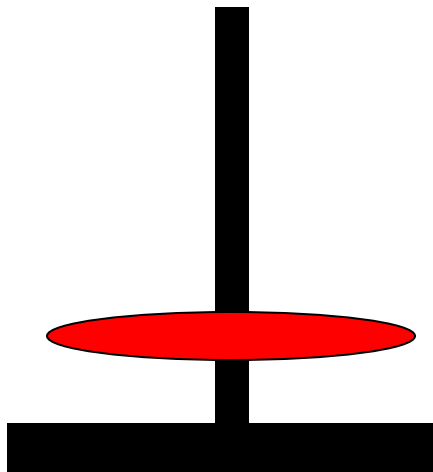
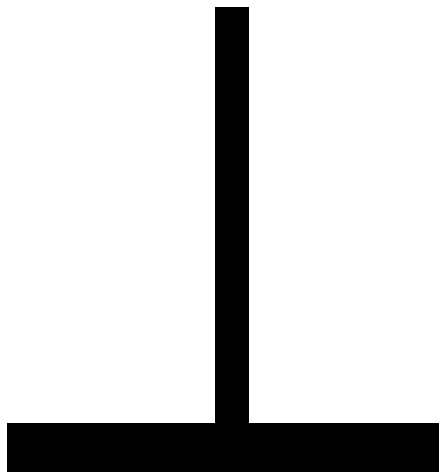
# Tower of Hanoi

---

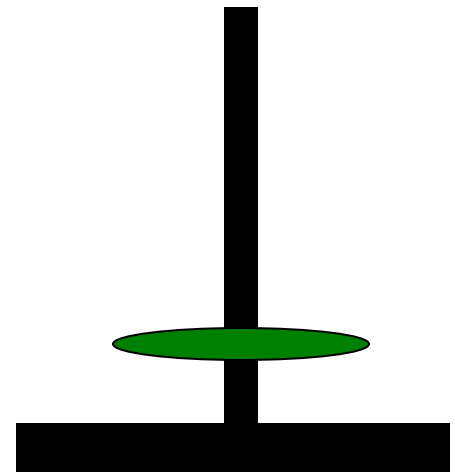
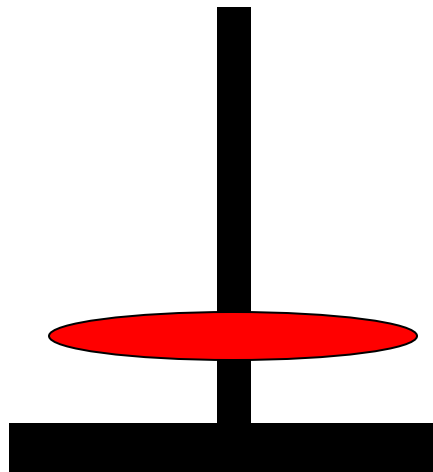
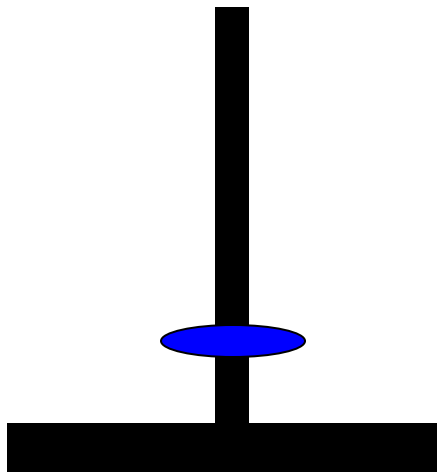


# Tower of Hanoi

---

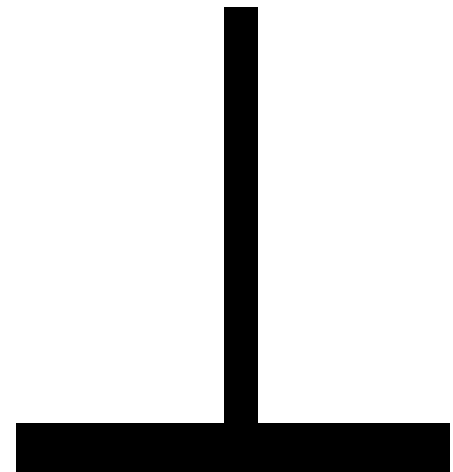
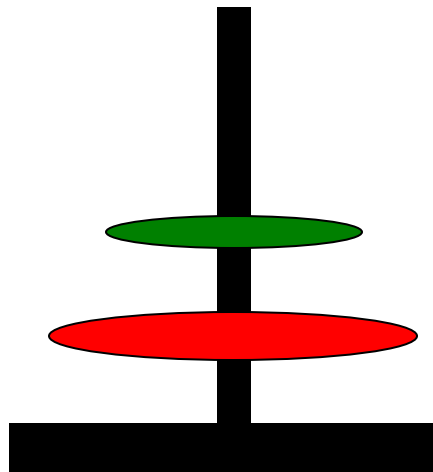
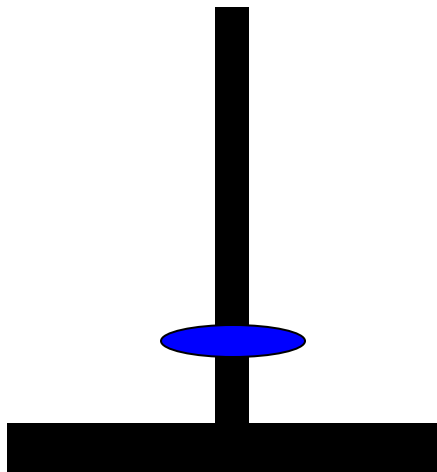


# Tower of Hanoi



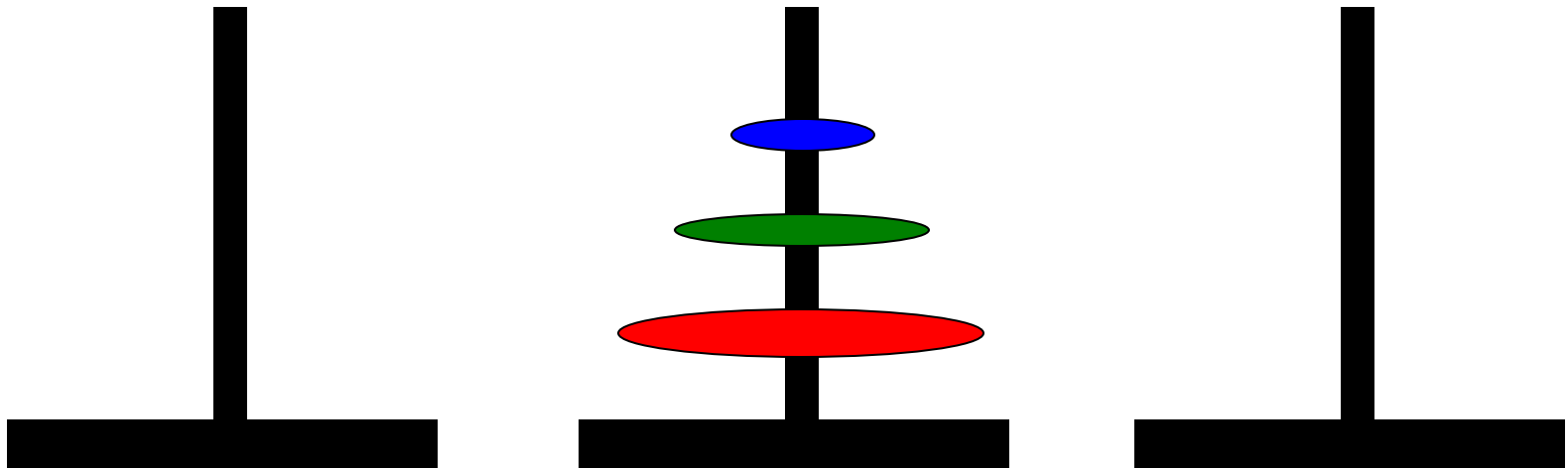
# Tower of Hanoi

---



# Tower of Hanoi

---





# Towers of Hanoi

---

```
Hanoi(n, from, to, temp) {  
    if (n == 1)  
        Move(from, to);  
    else{  
        Hanoi(n - 1, from, temp, to);  
        Move(from, to);  
        Hanoi(n - 1, temp, to, from);  
    }  
}
```

The recurrence relation for the running time of the method **Hanoi** is:

$$T(1) = 1$$

$$T(n) = 2T(n - 1) + 1 \quad \text{if } n > 1$$

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

# Guess and Prove

---

- Calculate  $T(n)$  for small  $n$  and look for a pattern.
- Guess the result and prove your guess correct using induction.

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

n	T(n)
1	1
2	3
3	7
4	15
5	31

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

# Iteration Method

---

Unwind recurrence, by repeatedly replacing  $T(n)$  by the r.h.s. of the recurrence until the base case is encountered.

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

$$= 2*[2*T(n-2)+1] + 1 = 2^2 * T(n-2) + 1+2$$

$$= 2^2 * [2*T(n-3)+1] + 1 + 2$$

$$= 2^3 * T(n-3) + 1+2 + 2^2$$

# Geometric Series

---

After k steps

$$T(n) = 2^k * T(n-k) + 1+2 + 2^2 + \dots + 2^{k-1}$$

$$T(n) = 2^{n-1} * T(1) + 1+2 + 2^2 + \dots + 2^{n-2}$$

$$\begin{aligned} &= 1 + 2 + \dots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i \\ \Theta(2^n) \end{aligned}$$

If  $n=64$  the  $2^{64}$  seconds about  $1.84 \times 10^{19}$  seconds or 584+ billion years

# Forming Recurrence Relations

---

```
public void f (int n) {  
    if (n > 0) {  
        System.out.println(n);  
        f(n-1);  
    }else  
        return;  
}
```

The recurrence relation is:

$$T(0) = 1$$

$$T(n) = T(n-1) + b \quad \text{if } n > 0$$

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

# Recurrences Solutions

---

- $T(n) = T(n-1) + n$   $\Theta(n^2)$ 
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$   $\Theta(\lg n)$ 
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$   $\Theta(n)$ 
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$   $\Theta(n)$ 
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

# Methods for Solving Recurrences

---

- Iteration method
- Substitution method
- Recursion tree method
- Master method
- Muster method

# The Iteration Method

---

- Convert the recurrence into a summation and try to bound it using a known series
  - Iterate the recurrence until the initial condition is reached.
  - Use back-substitution to express the recurrence in terms of  $n$  and the initial (boundary) condition.



# Iteration Method – Binary Search

---

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

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

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

$$= c + c + c + T(n/8)$$

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

$$T(n/4) = c + T(n/8)$$

Stop when  $n/2^i = 1 \Rightarrow i = \lg n$

$$T(n) = \underbrace{c + c + \dots + c}_{n \text{ times}} + T(1)$$

n times

$$= c \lg n + T(1)$$

$$= \Theta(\lg n)$$

# Iteration - Mergesort

---

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

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

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

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

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

$$= n + n + 4(n/4 + 2T(n/8))$$

$$= n + n + n + 8T(n/8)$$

$$\dots = in + 2^iT(n/2^i) \quad \text{stop at } i = \lg n$$

$$= n\lg n + 2^{\lg n}T(1)$$

$$= n\lg n + nT(1)$$

$$= \Theta(n\lg n)$$

# Substitution Method

---

- Guess a solution

$$T(n) = O(g(n))$$

Induction goal: apply the definition of the asymptotic notation

$$T(n) \leq c g(n), \text{ for some } c > 0 \text{ and } n \geq n_0$$

- Induction hypothesis:  $T(k) \leq c g(k)$  for all  $k < n$
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants  $d$  and  $n_0$  for which the **induction goal** holds

# Substitution: $T(n) = T(n-1) + T(n-2)$

---

Guess:  $T(n) = O(\phi^n)$

Induction goal:  $T(n) \leq c\phi^n$ , for some  $c$  and  $n \geq n_0$

- Induction hypothesis:  $T(k) \leq c\phi^k$  for  $k < n$
- Proof of induction goal:

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

$$\leq c\phi^{n-1} + c\phi^{n-2}$$

$$\leq c\phi^{n-2} (\phi + 1)$$

$$\leq c\phi^{n-2} (\phi^2)$$

$$T(n) \leq c\phi^n$$

$$T(n) = O(\phi^n)$$

Properties

$$\Phi = \frac{1 + \sqrt{5}}{2}$$

$$\Phi^2 = \frac{3 + \sqrt{5}}{2}$$

$$\Phi + 1 = \Phi^2$$

# Recursion-tree method

---

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- Convert the recurrence into a tree:
  - Each node represents the cost incurred at various levels of recursion
  - Sum up the costs of all levels
- The recursion-tree method can be unreliable, just like any method that uses ellipses (...).
- Usually involves geometric series

# Example of recursion tree

---

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

# Example of recursion tree

---

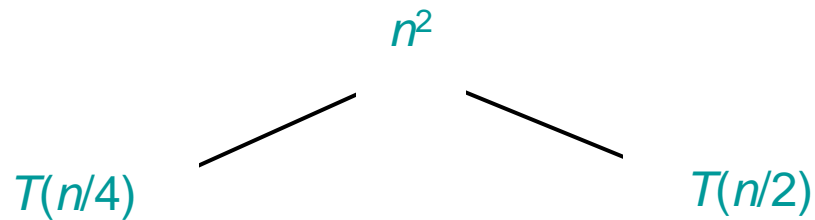
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :

$T(n)$

# Example of recursion tree

---

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

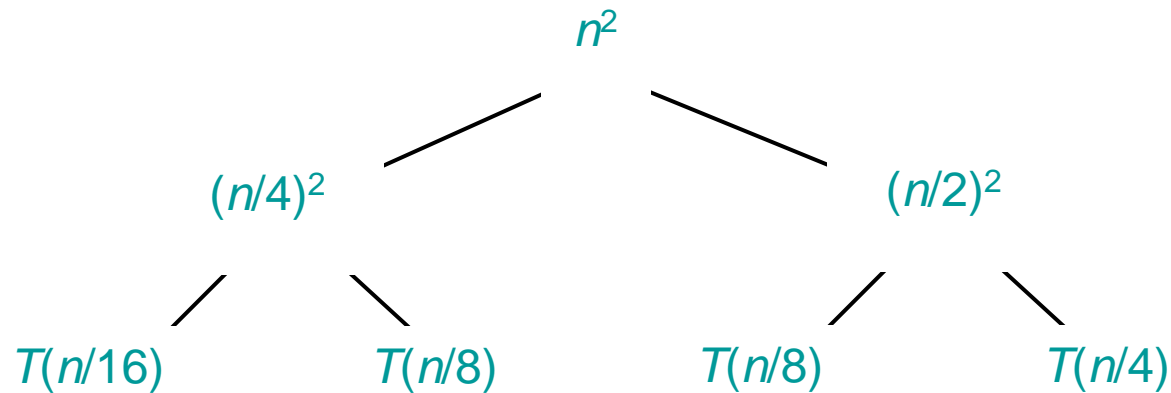




# Example of recursion tree

---

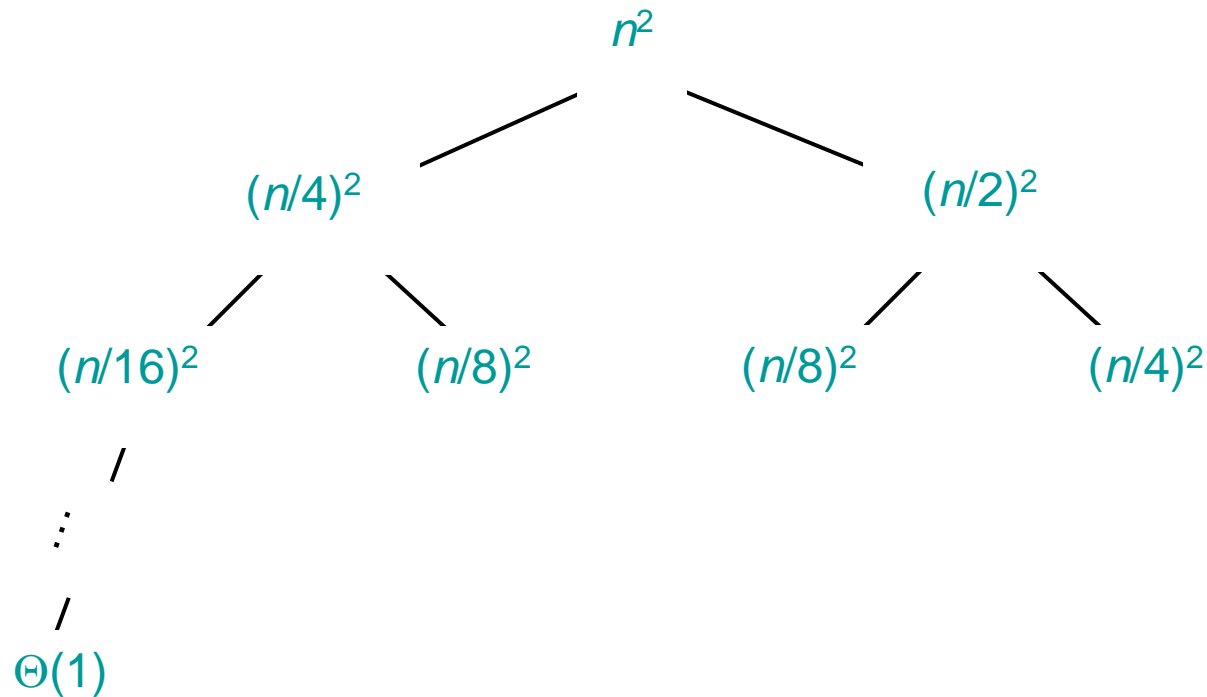
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



# Example of recursion tree

---

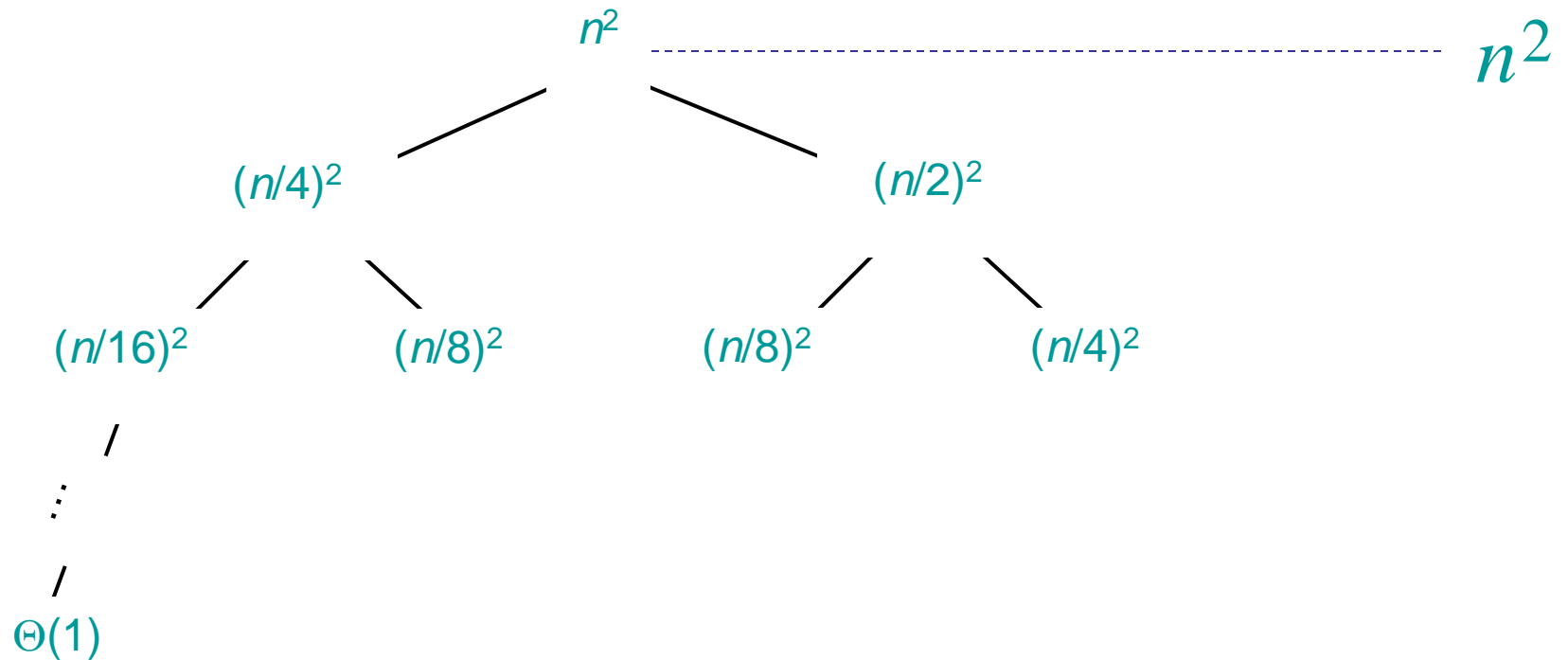
Solve  $T(n) = T(n/4) + T(n/2) + n^2$ :



# Example of recursion tree

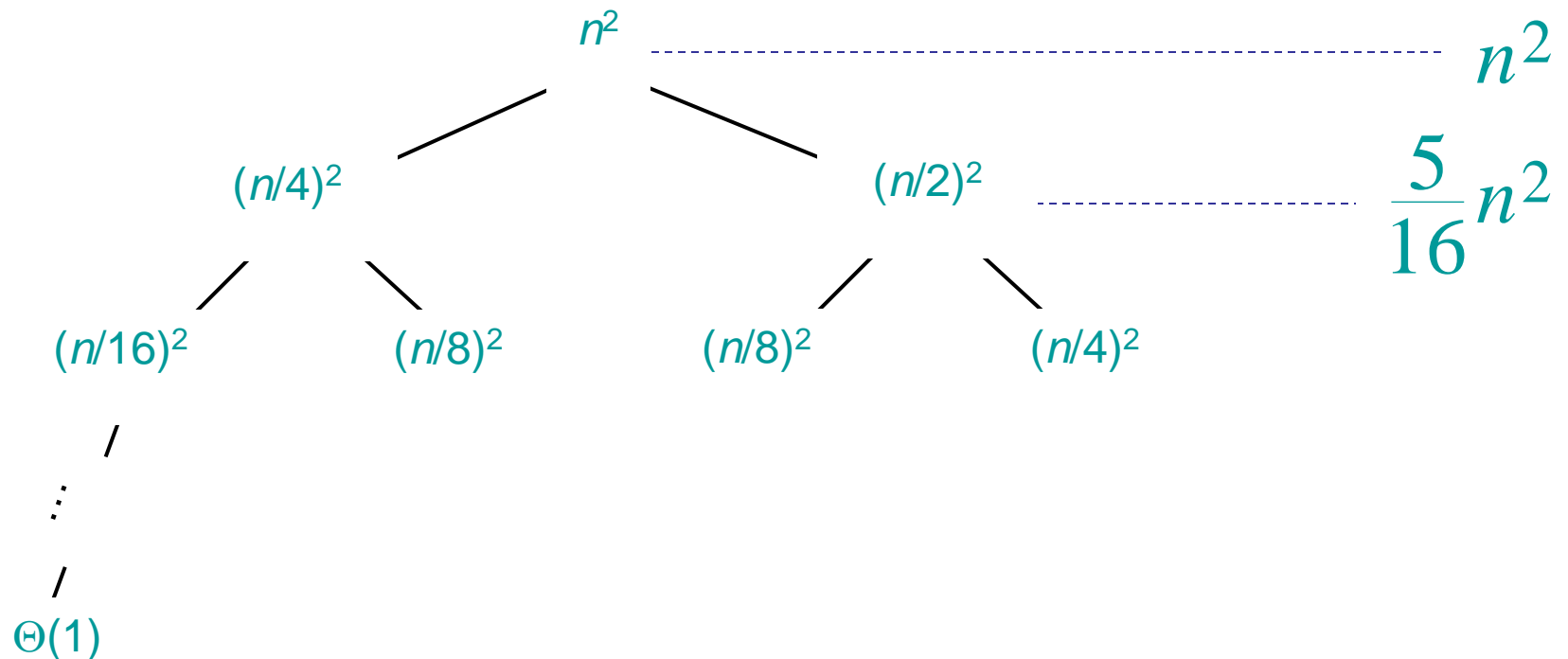
---

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



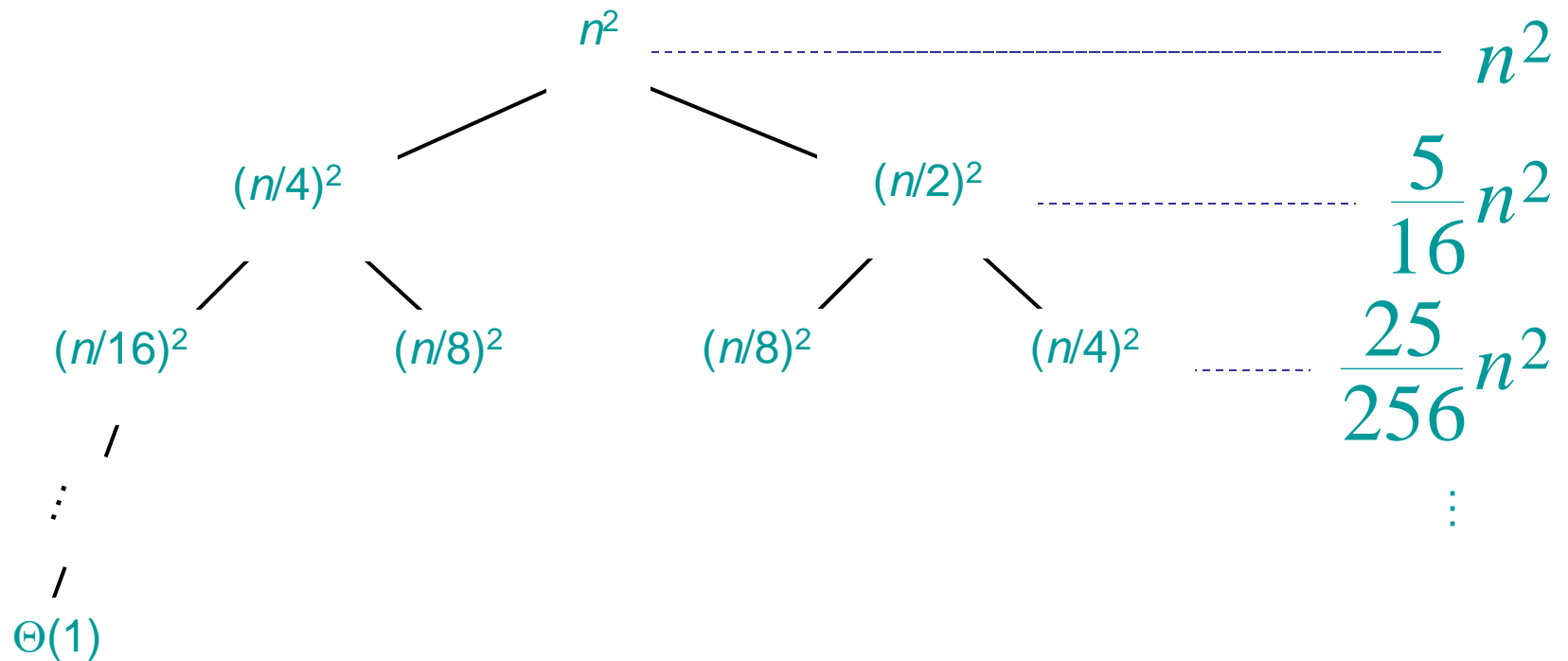
# Example of recursion tree

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



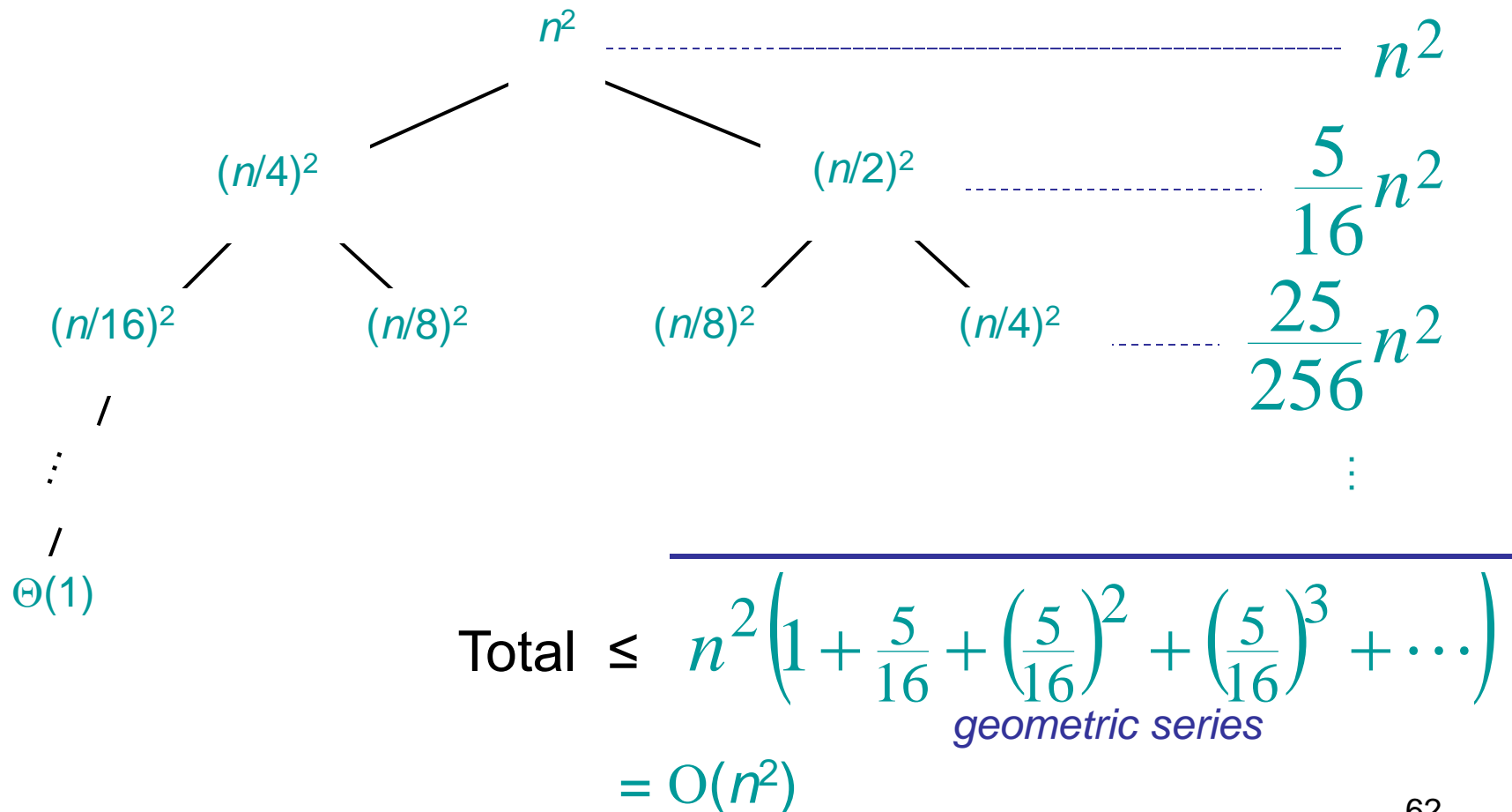
# Example of recursion tree

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



# Example of recursion tree

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



# Geometric series

---

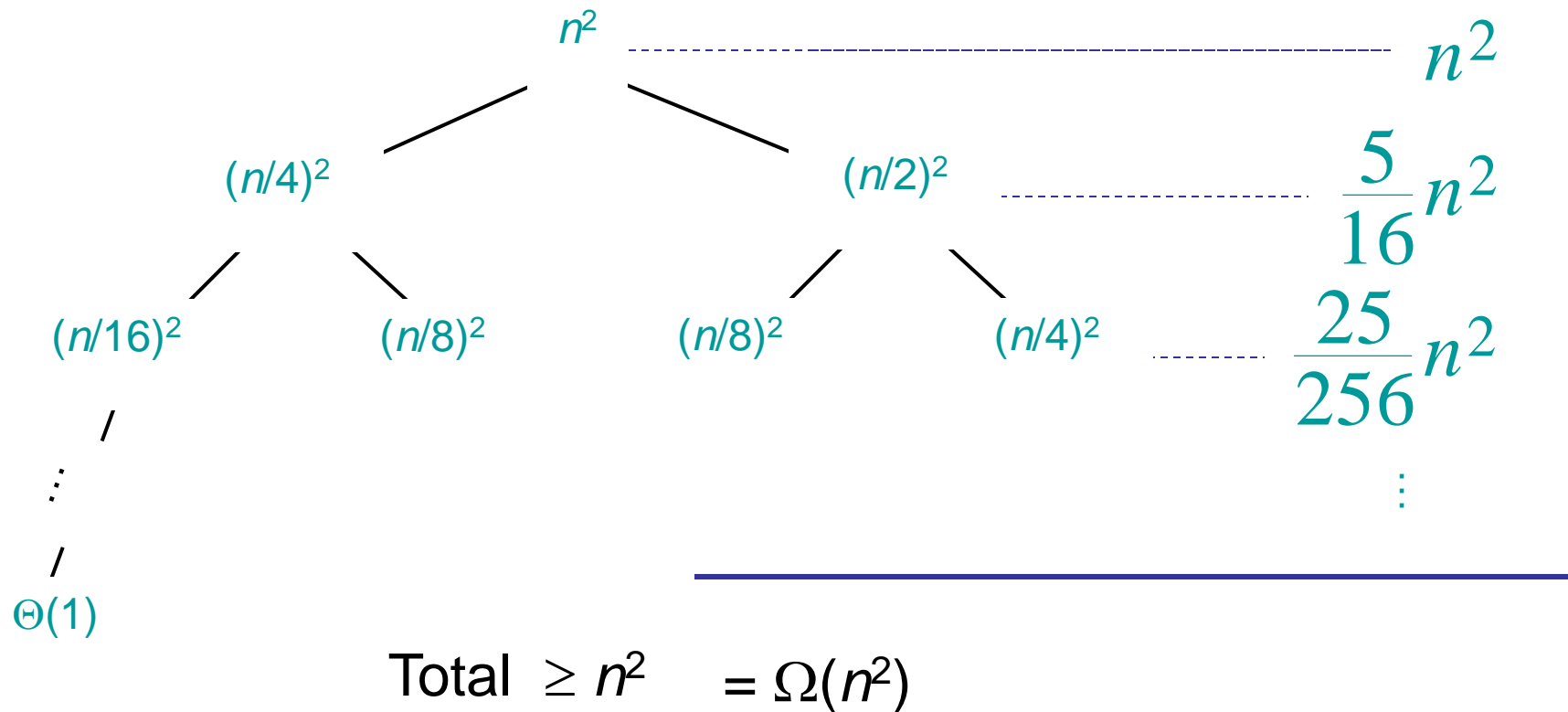
$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

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

$$n^2 \left( 1 + \frac{5}{16} + \left( \frac{5}{16} \right)^2 + \left( \frac{5}{16} \right)^3 + \cdots \right) = n^2 \left( \frac{1}{1 - \frac{5}{16}} \right) = \frac{16}{11} n^2$$

# Example of recursion tree

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



Therefore  $T(n) = \Theta(n^2)$



# The Master Method

---

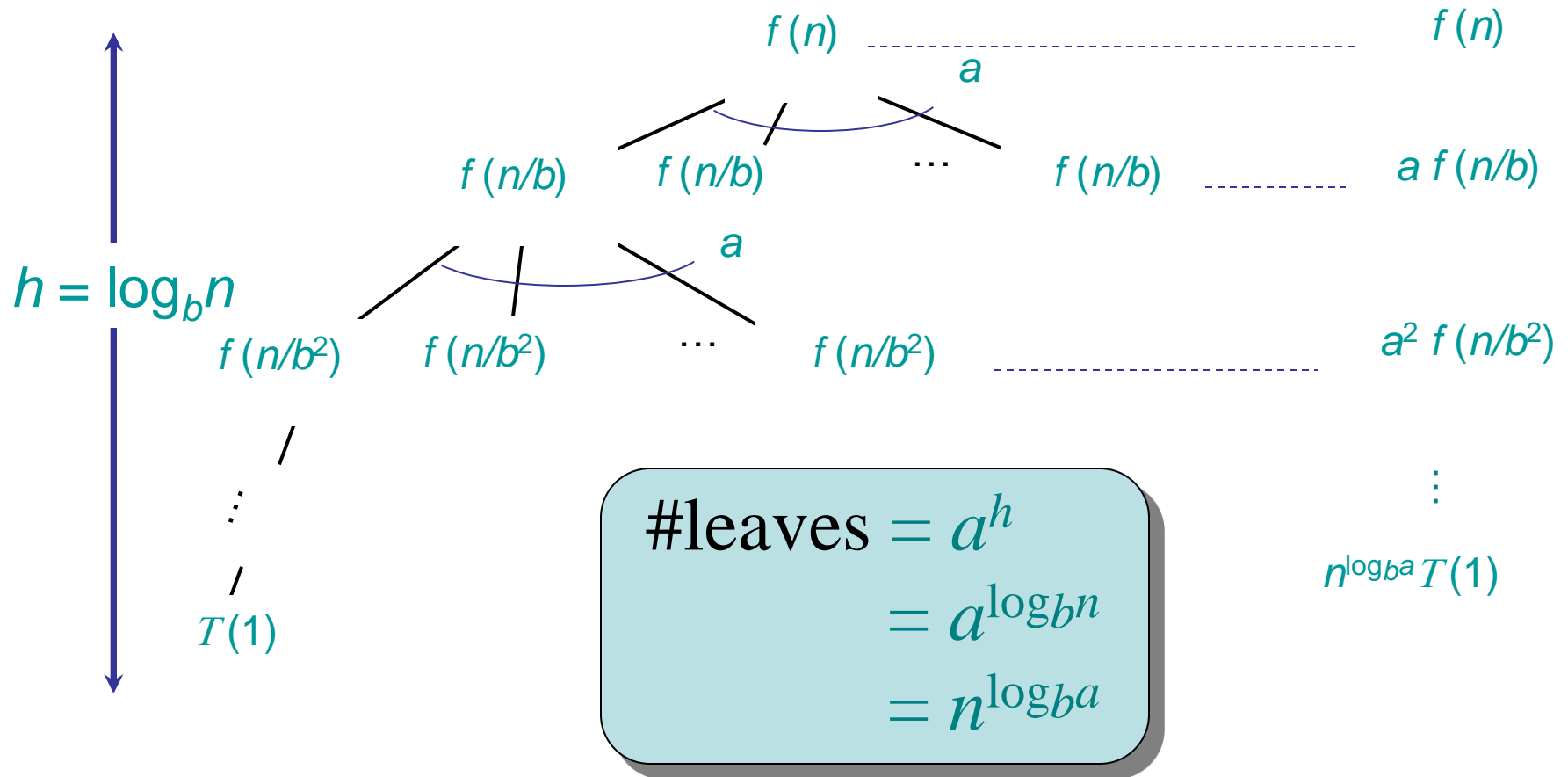
The master method applies to recurrences of the form

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

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

# Idea of Master Method

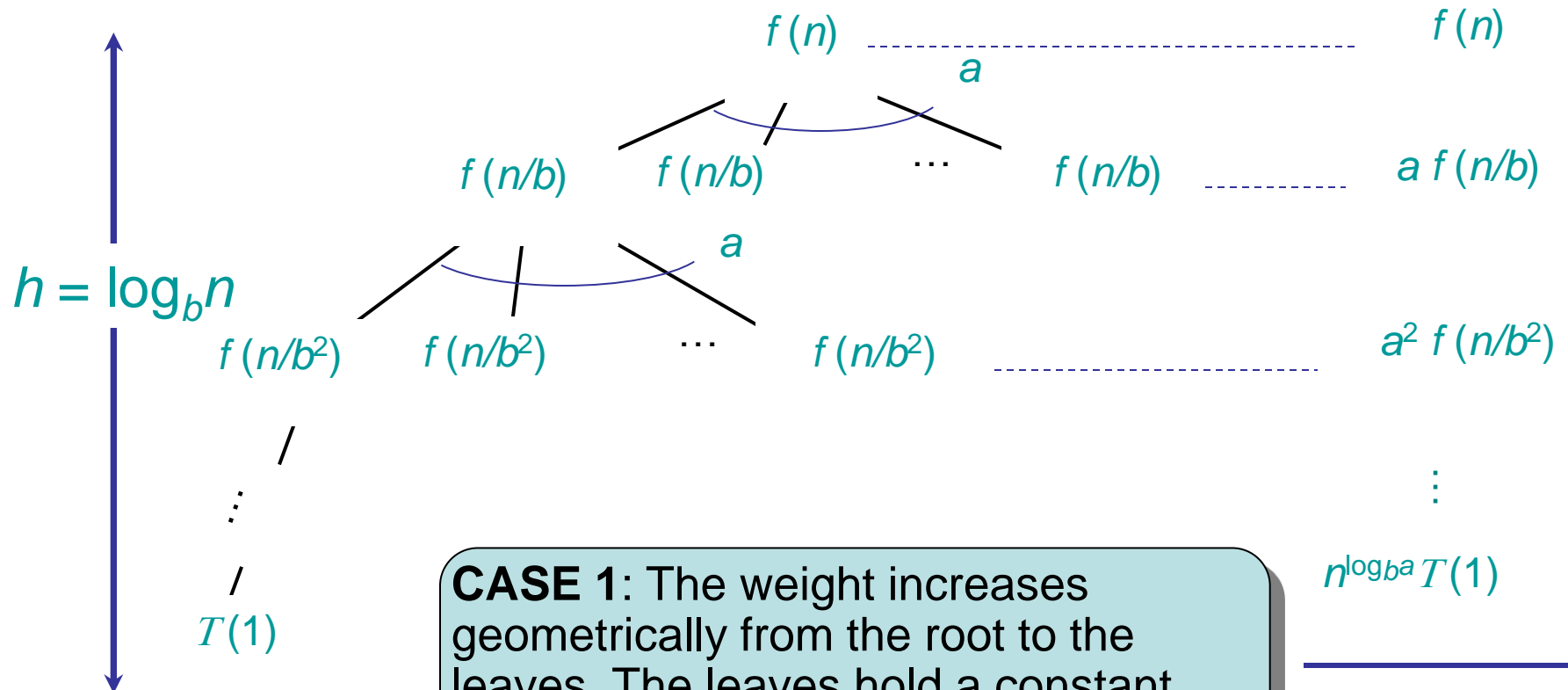
*Recursion tree:*



# Idea of Master Method

$$f(n) = O(n^{\log_b a - \varepsilon})$$

*Recursion tree:*



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

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

# 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  $n^\varepsilon$  factor).

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

# Case 1

---

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

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

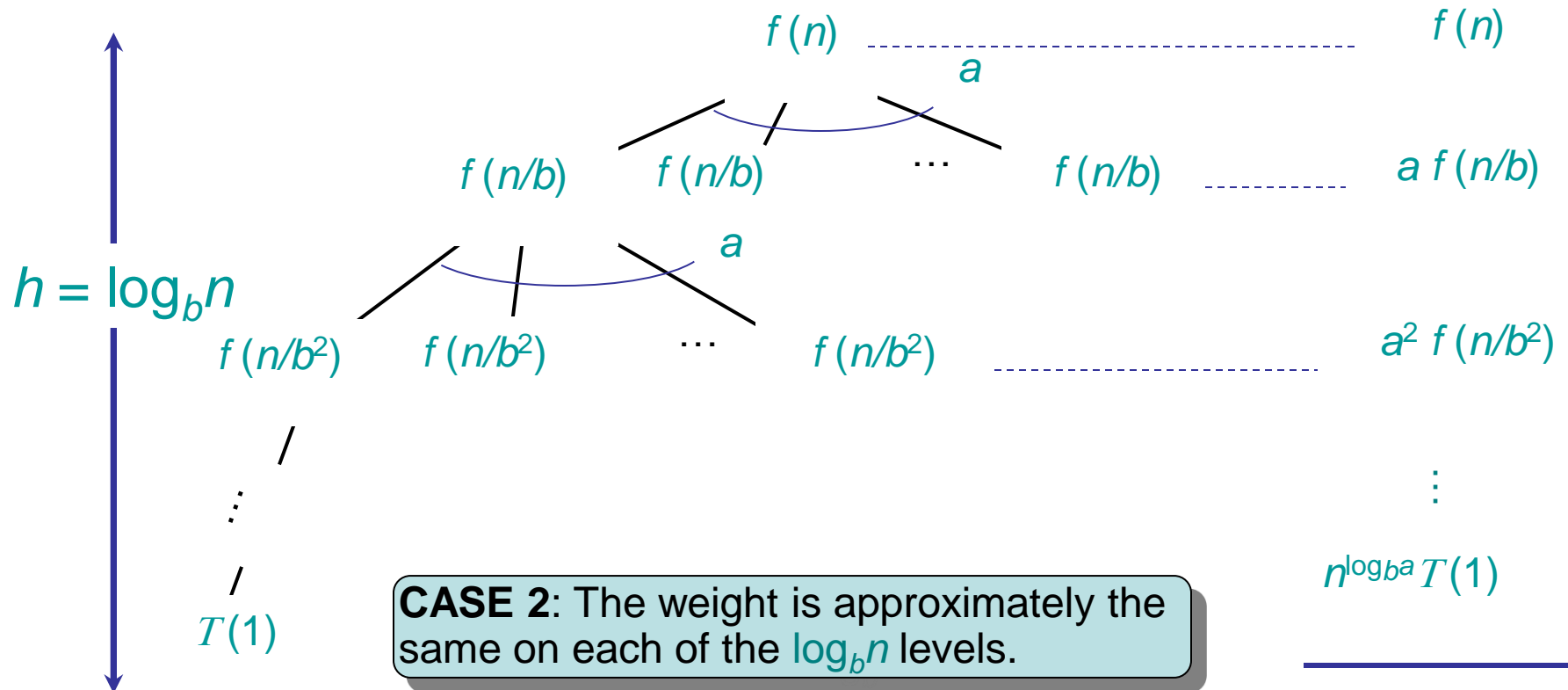
**CASE 1:**  $f(n) = O(n^{2-\varepsilon})$  for  $\varepsilon = 1$ .

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

# Idea of Master Method

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

Recursion tree:



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

# Case 2

---

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

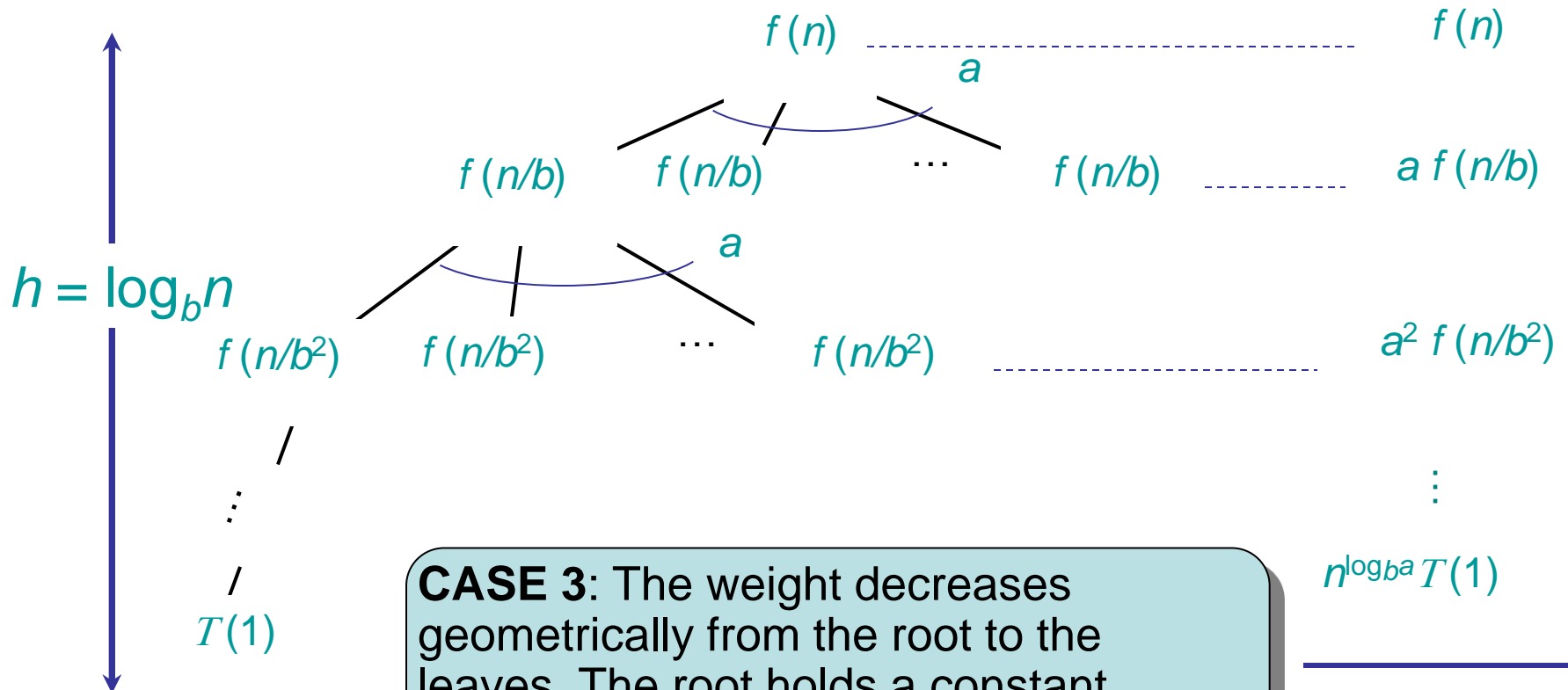
$$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$$

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

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

# Idea of master theorem

Recursion tree:



$$f(n) = \Omega(n^{\log_b a + \varepsilon})$$

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

$$\Theta(f(n))$$



# Case 3

---

**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*

$$4(cn/2)^3 \leq cn^3 \text{ (reg. cond.) for } c = 1/2.$$

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

# No Cases

---

*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.

# Master Method

---

“Formula” for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$


where,  $a \geq 1$ ,  $b > 1$ , and  $f(n) > 0$

**case 1:** if  $f(n) = O(n^{\log_b a - \varepsilon})$  for some  $\varepsilon > 0$ , then:  $T(n) = \Theta(n^{\log_b a})$

**case 2:** if  $f(n) = \Theta(n^{\log_b a})$ , then:  $T(n) = \Theta(n^{\log_b a} \lg n)$

**case 3:** if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$ , and if

$af(n/b) \leq cf(n)$  for some  $c < 1$  and all sufficiently large  $n$ , then:

  
regularity

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

# Master Method – Binary Search

---

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

$$a = 1, b = 2, \log_2 1 = 0$$

compare  $n^{\log_2 1} = n^0 = 1$  with  $f(n) = c$

**Case 2:** if  $f(n) = \Theta(n^{\log_b a})$ , then:  $T(n) = \Theta(n^{\log_b a} \lg n)$

$$f(n) = \Theta(1) \Rightarrow \text{case 2}$$

$$\Rightarrow T(n) = \Theta(\lg n)$$

# Master Method – Example 1

---

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

$$a = 2, b = 2, \log_2 2 = 1$$

compare  $n$  with  $f(n) = n^2$

**case 3:** if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$  case 3  $\Rightarrow$  verify regularity cond.

$$a f(n/b) \leq c f(n)$$

$$\Leftrightarrow 2 n^2/4 \leq c n^2 \Rightarrow c = 1/2 \text{ is a solution } (c < 1)$$

$$\Rightarrow T(n) = \Theta(n^2)$$

# Master Method – Example 2

---

$$T(n) = 2T(n/2) + \sqrt{n} \quad a = 2, b = 2, \log_2 2 = 1$$

compare  $n$  with  $f(n) = n^{1/2}$

$$\Rightarrow f(n) = O(n^{1-\varepsilon}) \quad \text{case 1}$$

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

# Master Method - Example 3

---

$$T(n) = 3T(n/4) + n \lg n \quad a = 3, b = 4, \log_4 3 = 0.793$$

compare  $n^{0.793}$  with  $f(n) = n \lg n$

$$f(n) = \Omega(n^{\log_4 3 + \epsilon}) \quad \text{case 3}$$

check regularity condition:

$$3 * (n/4) \lg(n/4) \leq (3/4) n \lg n = c * f(n), \quad c = 3/4$$

$$\Rightarrow T(n) = \Theta(n \lg n)$$

# Master Method: Merge-Sort

---

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + kn$$

where,  $a = 2$ ,  $b = 2$ , and  $f(n) = n$

$$n^{\log_b a} = n^{\log_2 2} = n$$

**case 1:** if  $f(n) = O(n^{\log_b a - \varepsilon})$  for some  $\varepsilon > 0$ , then:  $T(n) = \Theta(n^{\log_b a})$

**case 2:** if  $f(n) = \Theta(n^{\log_b a})$ , then:  $T(n) = \Theta(n^{\log_b a} \lg n)$

**case 3:** if  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$ , and if

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



# Decrease and Conquer

---

**Muster Method** for “*decrease and conquer*”  
recurrences of the form

$$T(n) = a T(n-b) + f(n)$$

for some integer constants  $a, b > 0, d \geq 0$ .

If  $f(n)$  is  $O(n^d)$  then

$$T(n) = \begin{cases} O(n^d), & \text{if } a < 1, \\ O(n^{d+1}), & \text{if } a = 1 \\ O(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

# Decrease and Conquer

---

## Muster Method Version 2

$$T(n) = a T(n-b) + f(n)$$

for some integer constants  $a, b > 0, d \geq 0$ .

**If  $f(n)$  is  $\Theta(n^d)$  then**

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < 1, \\ \Theta(n^{d+1}), & \text{if } a = 1 \\ \Theta(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

# Decrease and Conquer: Example

---

**Tours of Hanoi recurrence:**  $T(n) = 2 T(n-1) + 1$

By the Muster Method:  $T(n) = a T(n-b) + f(n)$

$a = 2, b = 1, f(n) = 1$  so  $d = 0$ .  $f(n) = \Theta(n^0)$

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < 1, \\ \Theta(n^{d+1}), & \text{if } a = 1 \\ \Theta(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

Therefore,  $T(n)$  is  $\Theta(n^0 2^{n/1}) = \Theta(2^n)$ .

# Muster Method - Example

---

**Recurrence:**  $T(n) = 4T(n-1) + c$

By the Muster Method:  $T(n) = a T(n-b) + f(n)$

$$a = 4, b = 1, f(n) = c \text{ so } d = 0. \quad f(n) = \Theta(n^0)$$

$$T(n) = \begin{cases} \Theta(n^d), & \text{if } a < 1, \\ \Theta(n^{d+1}), & \text{if } a = 1 \\ \Theta(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

Therefore,  $T(n)$  is  $\Theta(n^0 4^{n/1}) = \Theta(4^n)$ .

# Substitution Method

---

$$T(n) = 4T(n-1) + c$$

$$= 4[4T(n-2) + c] + c$$

$$= 4^2 T(n-2) + 4c + c$$

$$= 4^2[4T(n-3) + c] + 4c + c$$

$$= 4^3 T(n-3) + 4^2c + 4c + c$$

$$= 4^k T(n-k) + \dots + 4^2c + 4c + c$$

Stop when  $k = n-1$  and  $T(1) = 1$

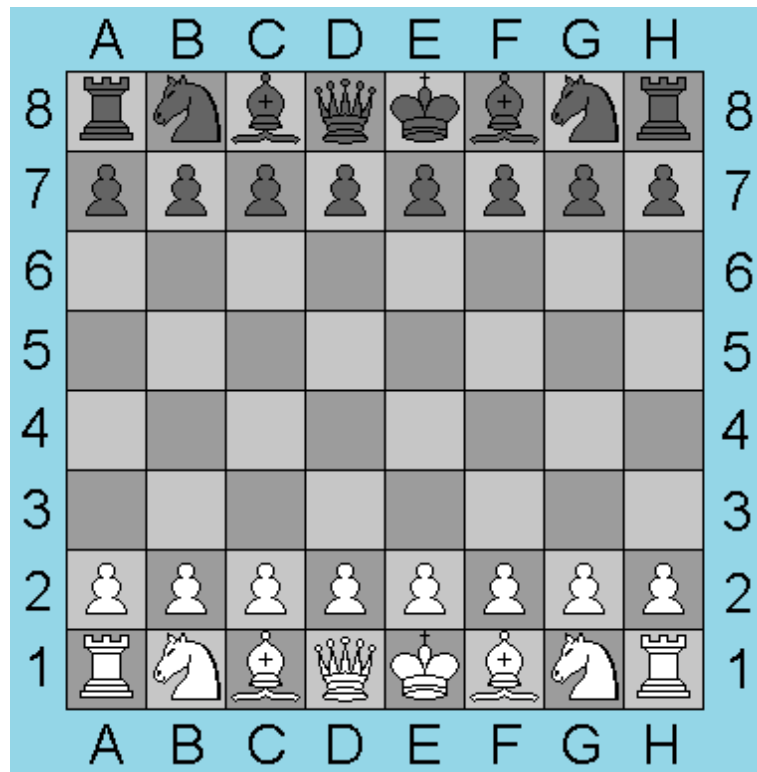
$$= 4^{n-1} T(1) + 4^{n-2}c + \dots + 4^2c + 4c + c$$

$$= 4^{n-1} + 4^{n-2}c + \dots + 4^2c + 4c + c$$

$$T(n) = \Theta(4^n)$$

# Tiling A Defective chessboard

---



**A real chessboard.**

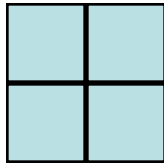
# Our Definition Of A chessboard

---

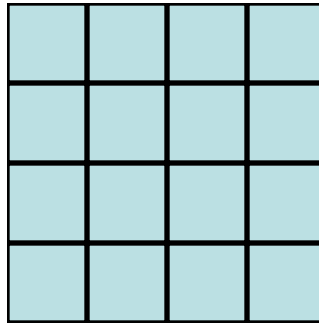
A chessboard is an  $n \times n$  grid, where  $n$  is a power of 2.



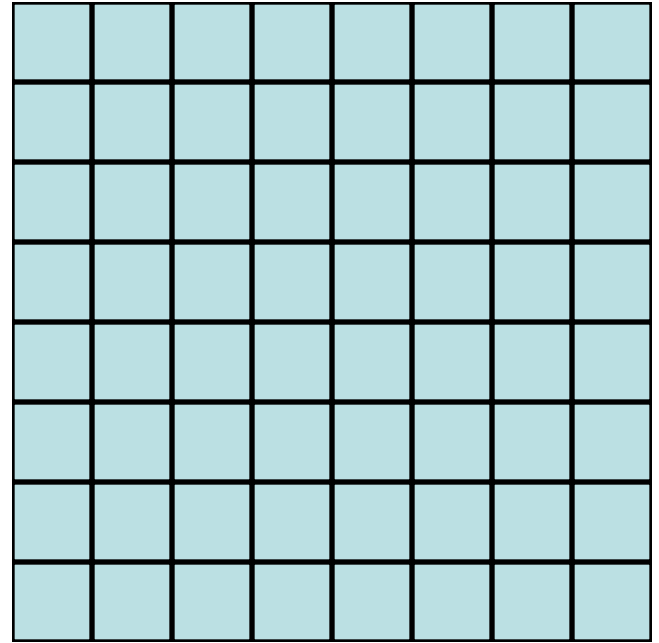
1x1



2x2



4x4



8x8



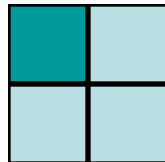
# A Defective chessboard



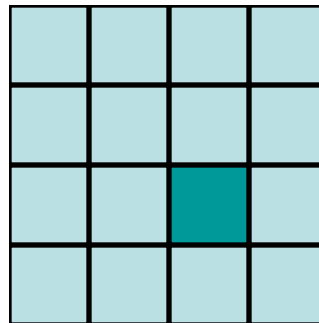
A defective chessboard is a chessboard that has one unavailable (defective) position.



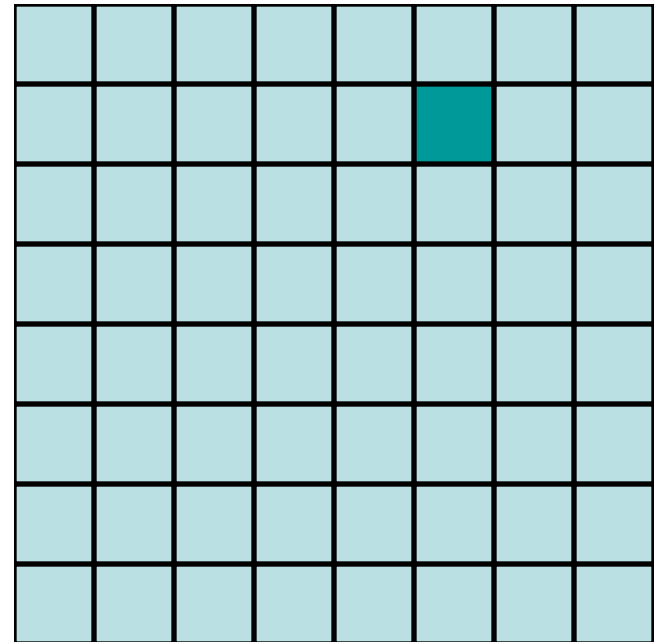
1x1



2x2



4x4



8x8

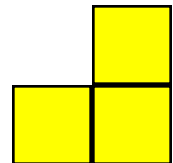
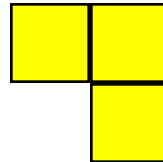
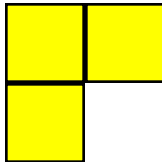
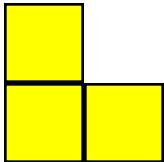


# A Triomino

---

A triomino is an L shaped object that can cover three squares of a chessboard.

A triomino has four orientations.



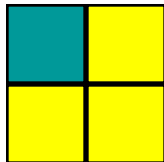
# Tiling A Defective chessboard

---

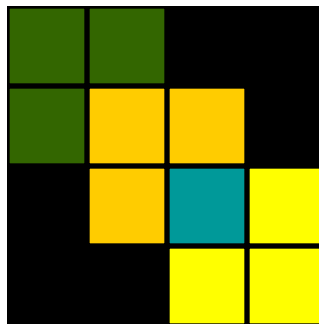
Place triominoes on an  $2^n \times 2^n$  defective chessboard so that all nondefective positions are covered.



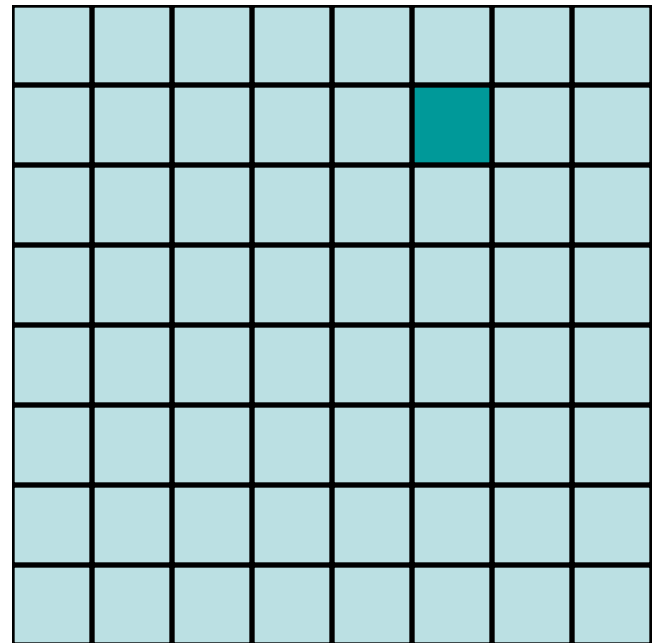
1x1



2x2

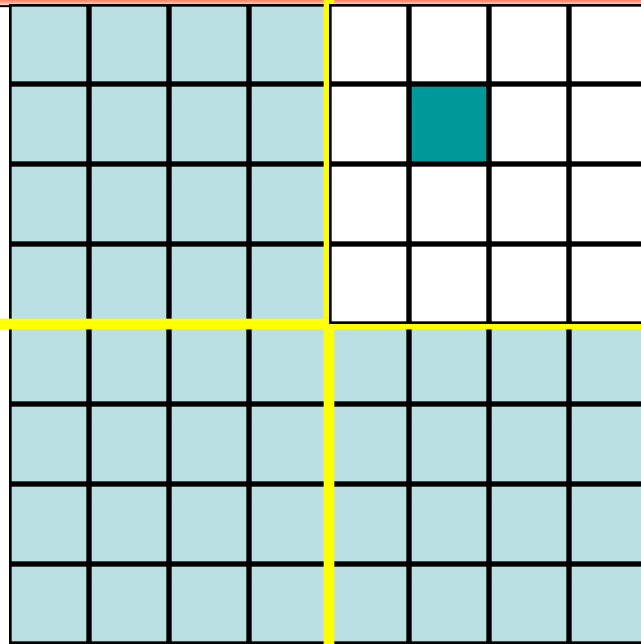


4x4



8x8

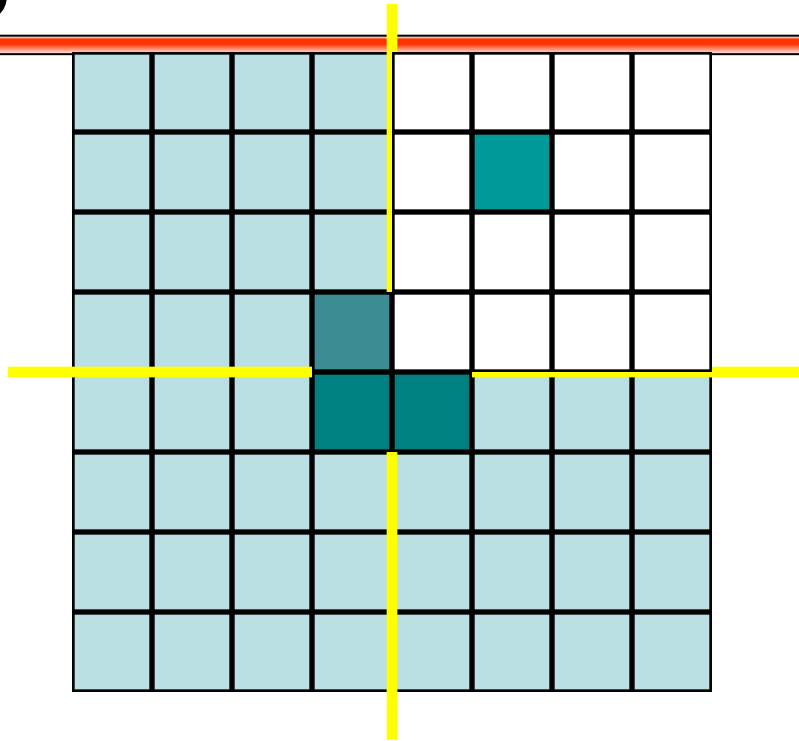
# Tiling A Defective chessboard



Divide the original chessboard into four smaller chessboards each 4 x 4

One of these is a defective 4 x 4 chessboard.

# Tiling A Defective chessboard



Make the other three 4 x 4 chessboards defective by placing a triomino at their common corner.

Recursively tile the four defective 4 x 4 chessboards.

# Tiling: Algorithm

**INPUT:**  $n$  – the board size ( $2^n \times 2^n$  board),  $L$  – location of the defective “hole”.

**OUTPUT:** tiling of the board

**Tile**( $n, L$ )

**if**  $n = 1$  **then**

*Trivial case*

    Tile with one tromino

**return**

Divide the board into four equal-sized boards  $2^{n-1} \times 2^{n-1}$

Place one tromino at the center to cut out 3 additional holes (orientation based on where existing hole,  $L$ , is)

Let  $L_1, L_2, L_3, L_4$  denote the positions of the 4 holes

**Tile**( $n-1, L_1$ )

**Tile**( $n-1, L_2$ )

**Tile**( $n-1, L_3$ )

**Tile**( $n-1, L_4$ )

# Recurrence

---

Let  $T(n)$  be the time taken to tile a  $2^n \times 2^n$  defective chessboard.

$$T(1) = 1,$$

$$T(n) = 4T(n-1) + c, \text{ when } n > 0.$$