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
- Tiling

Recall from Week 1

- Asymptotic Analysis: O, Ω, Θ
- 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

```
for (i=1; i<=n*n; i++)     Executed n*n times
  for (j=0; j<i; j++)       Executed <= n*n times
    sum++;                   O(1)
```

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

$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

$$= \frac{n^4+n^2}{2}$$

$$\in \Theta(n^4)$$

The Divide and Conquer Approach

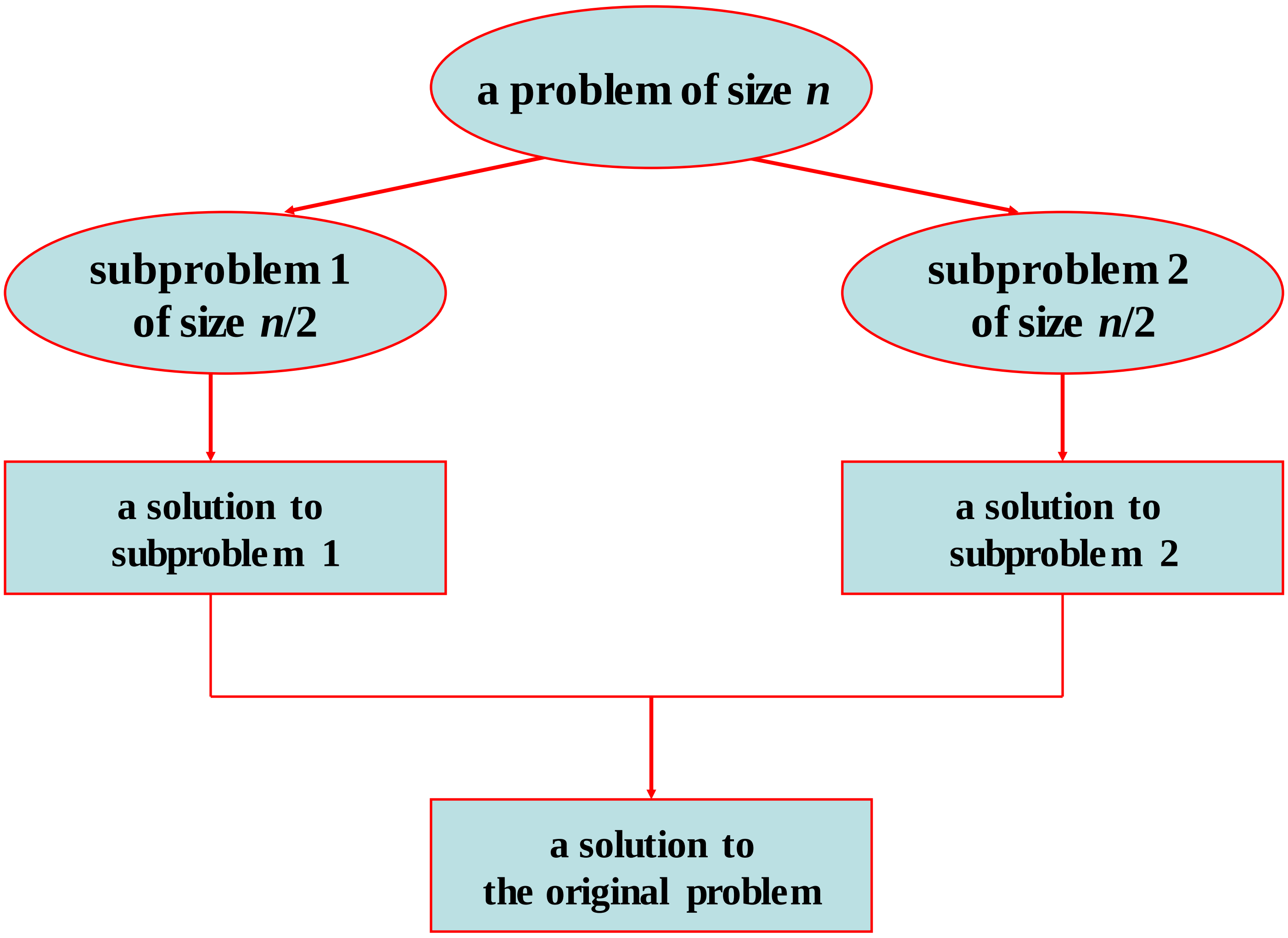
The most well known algorithm design strategy:

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.

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

- 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 $\boldsymbol{S}$ with $\boldsymbol{n}$ elements consists of three steps:
  - Divide: partition $\boldsymbol{S}$ into two sequences $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of about $\boldsymbol{n}/2$ elements each
  - conquer: recursively sort $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - combine: merge $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a unique sorted sequence

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

    Input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements, comparator $\boldsymbol{c}$

    Output sequence $\boldsymbol{S}$ sorted according to $\boldsymbol{c}$

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

        $(\boldsymbol{S}_1, \boldsymbol{S}_2) \leftarrow \boldsymbol{partition}(\boldsymbol{S}, \boldsymbol{n}/2)$

        $\boldsymbol{mergeSort}(\boldsymbol{S}_1, \boldsymbol{c})$

        $\boldsymbol{mergeSort}(\boldsymbol{S}_2, \boldsymbol{c})$

        $\boldsymbol{S} \leftarrow \boldsymbol{merge}(\boldsymbol{S}_1, \boldsymbol{S}_2)$

Recurrence Equation

- The conquer 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.

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

 # subproblems

subproblem size

work dividing and combining

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	$\boxed{9}$	12	15

Binary Search

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

 # subproblems

subproblem size

work dividing and combining

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

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: $a^n = a*a*\ldots*a$

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 $a^1 = a$

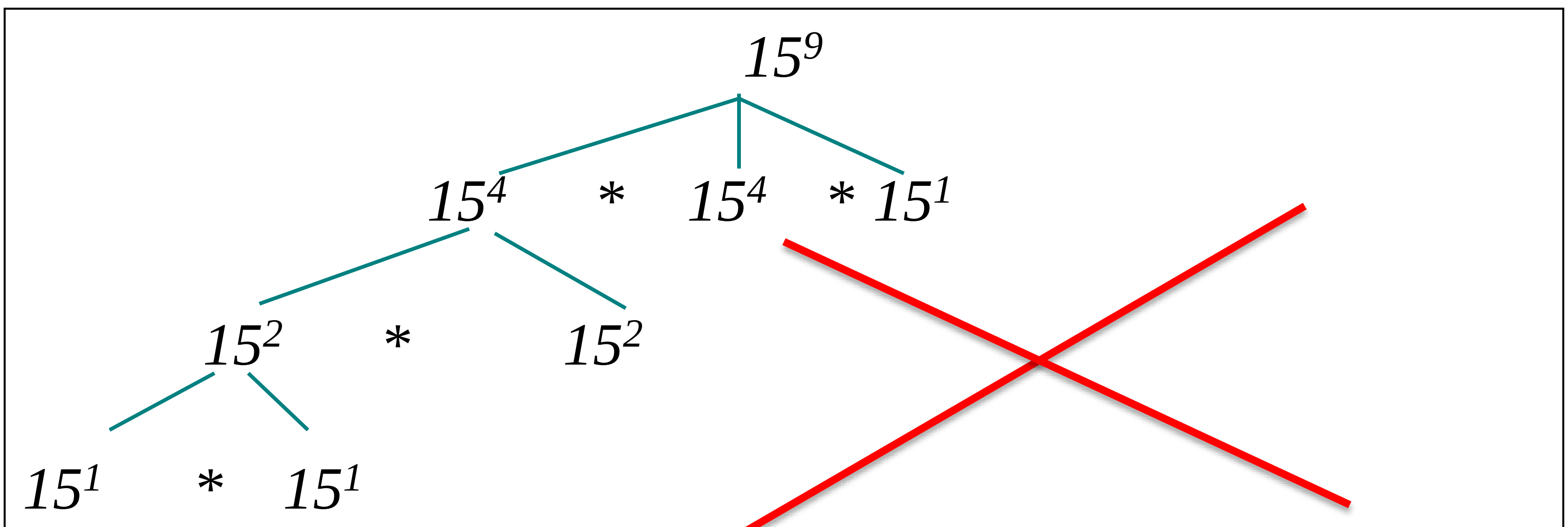
$$T(n) = T(n/2) + \Theta(1)$$

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:

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

$$T(n) = T(n/2) + c \qquad \text{if } n > 2$$

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:

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

$$T(n) = 2T(n/2) + c \qquad \text{if } n > 2$$

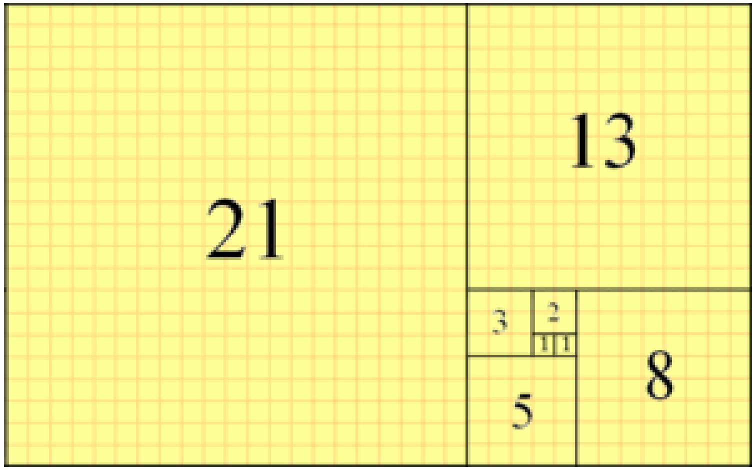
Running time $\Theta(n)$

Fibonacci Numbers

Recursive definition:

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

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$$



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);
}
```

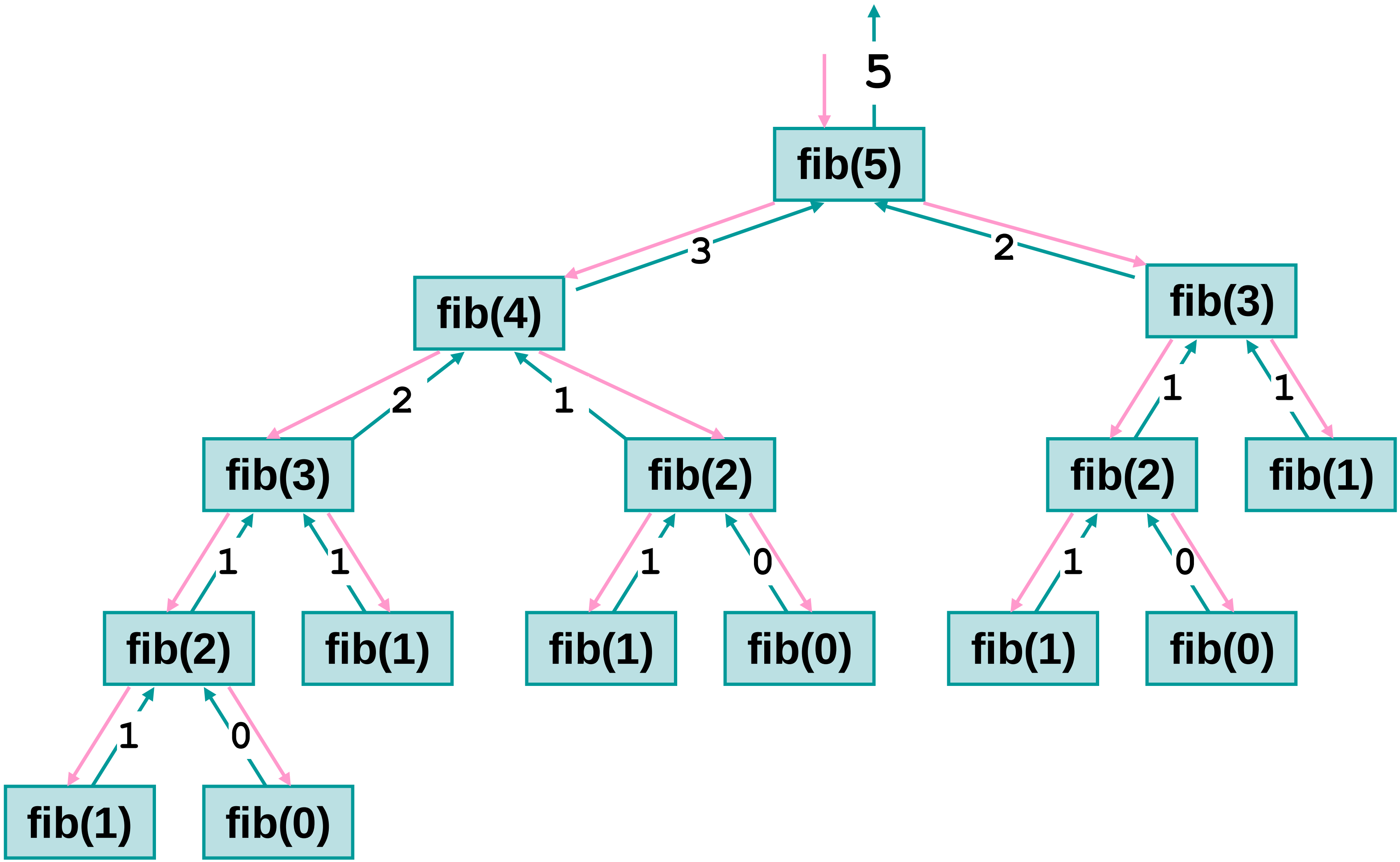
The recurrence relation is:

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

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

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

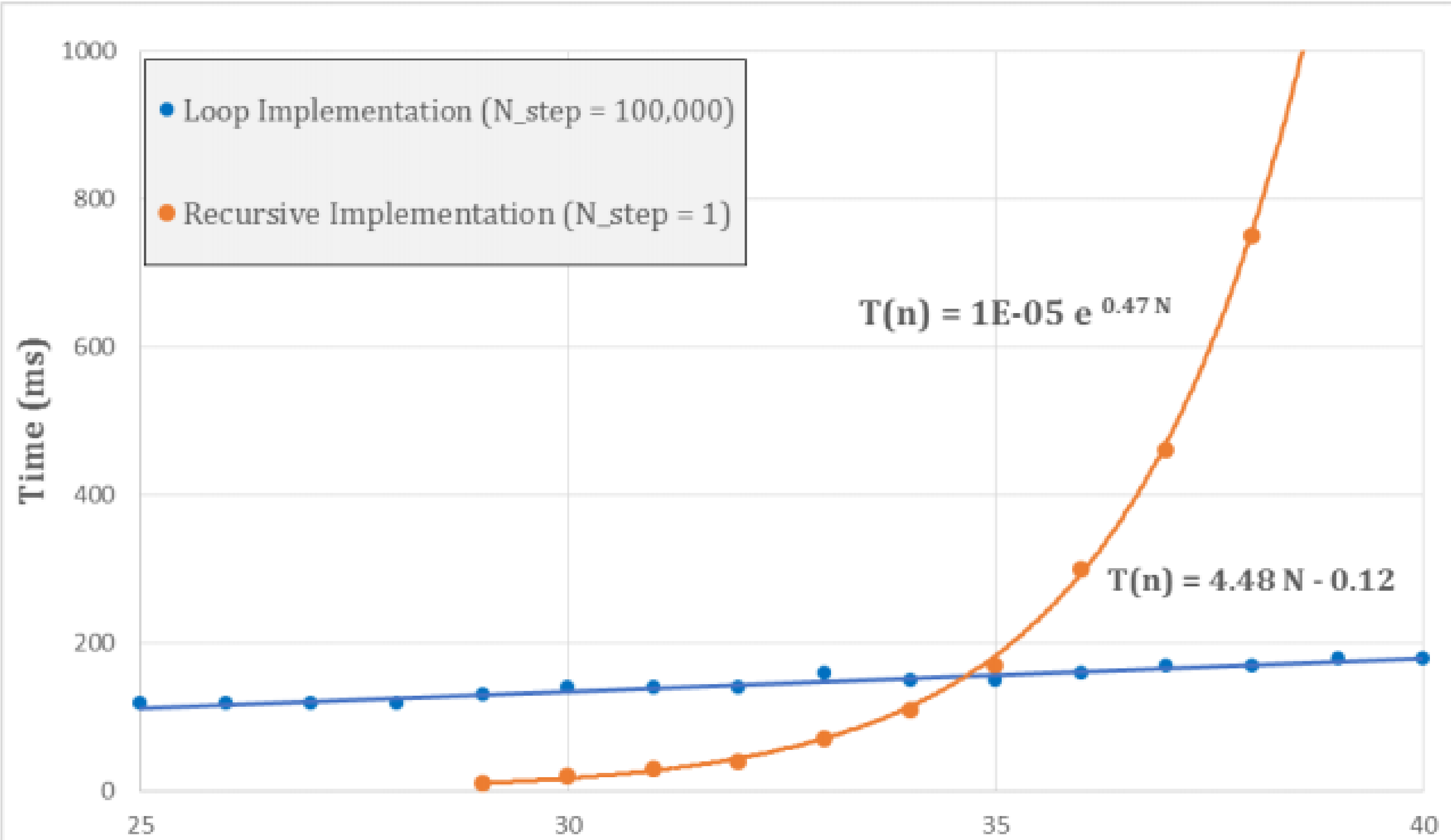
Function Analysis for call `fib(5)`



HW 1 Solution

Fibonnacci Performance Comparison:

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



Tower of Hanoi

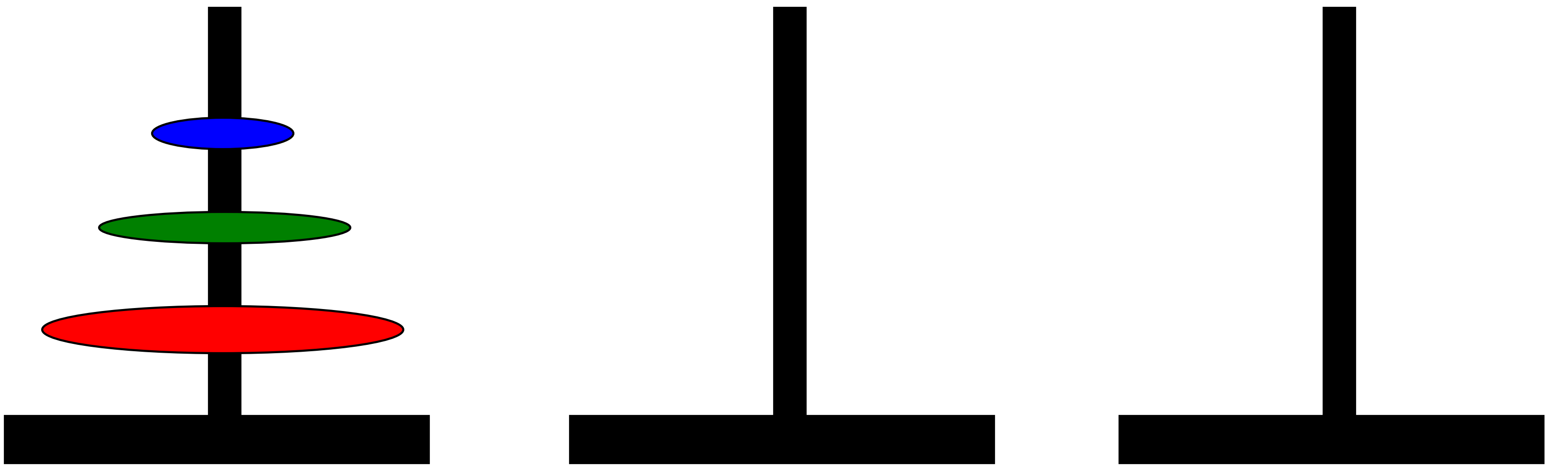
- There are three towers
- N gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

Tower of Hanoi

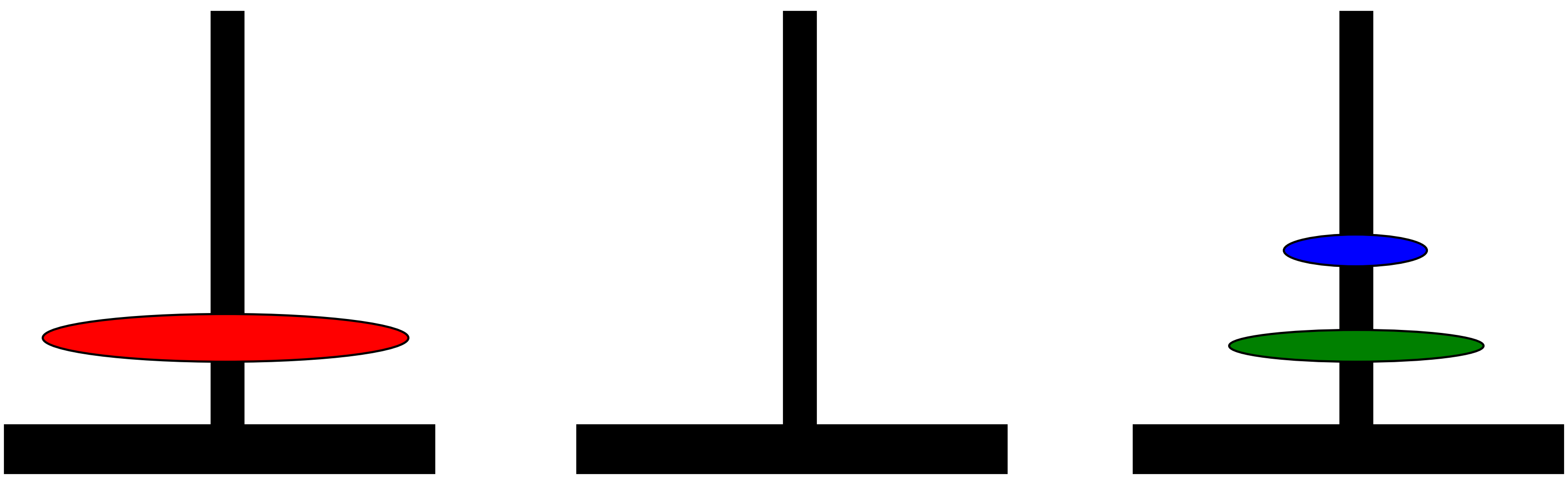
- The disks must be moved within one week. Assume one disk can be moved in 1 second. Is this possible?

- 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.

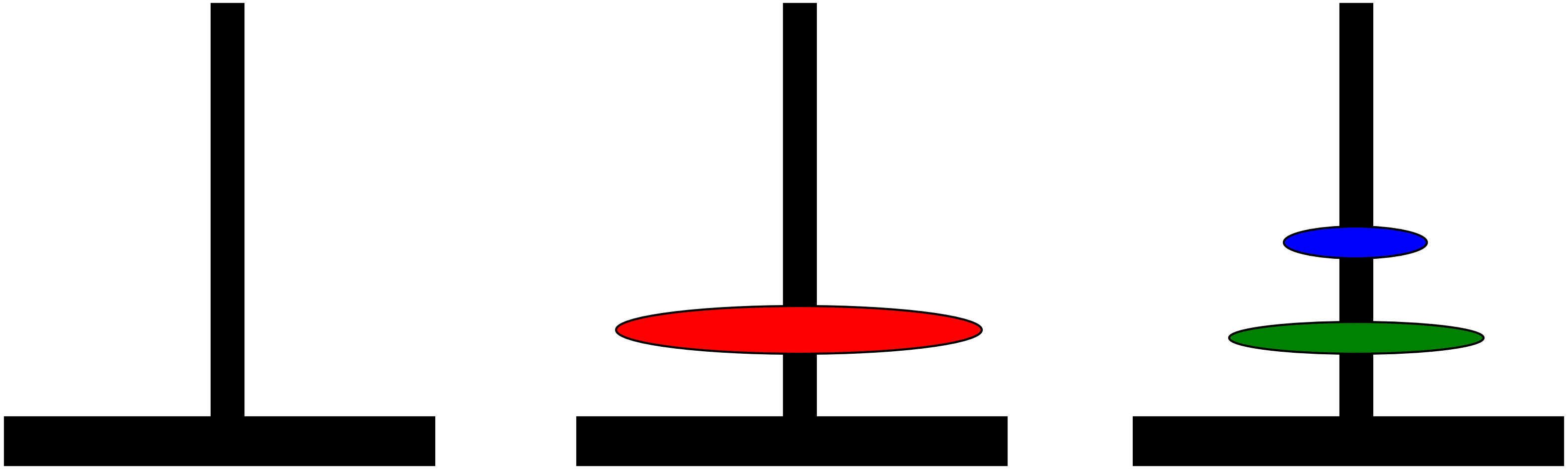
Recursive Solution



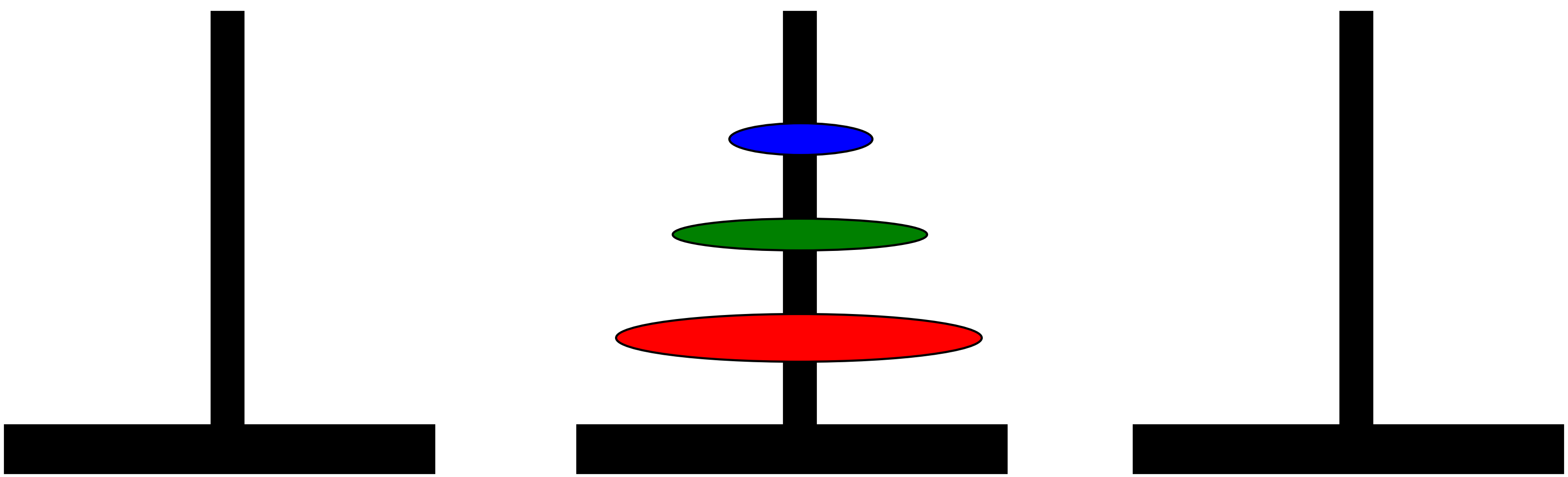
Recursive Solution



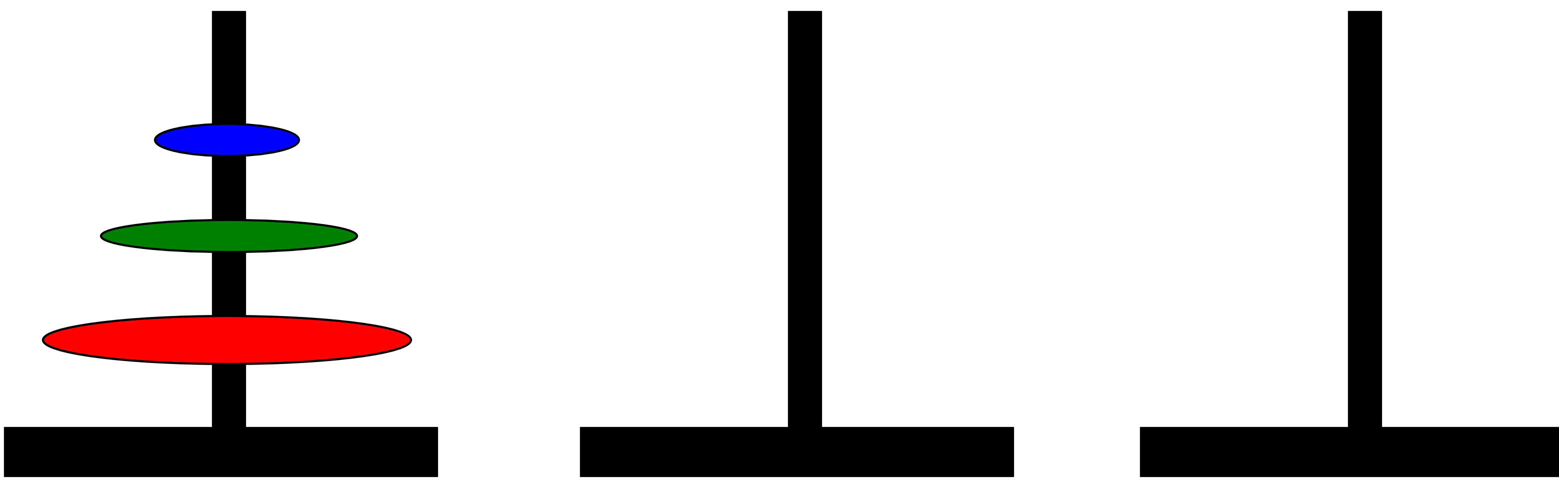
Recursive Solution



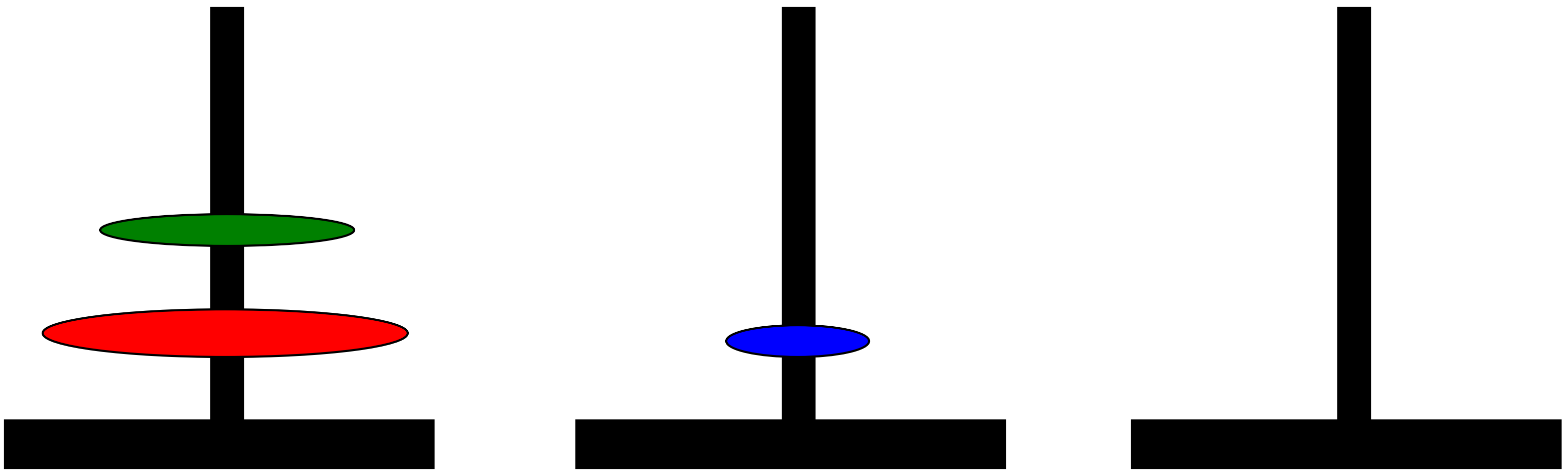
Recursive Solution



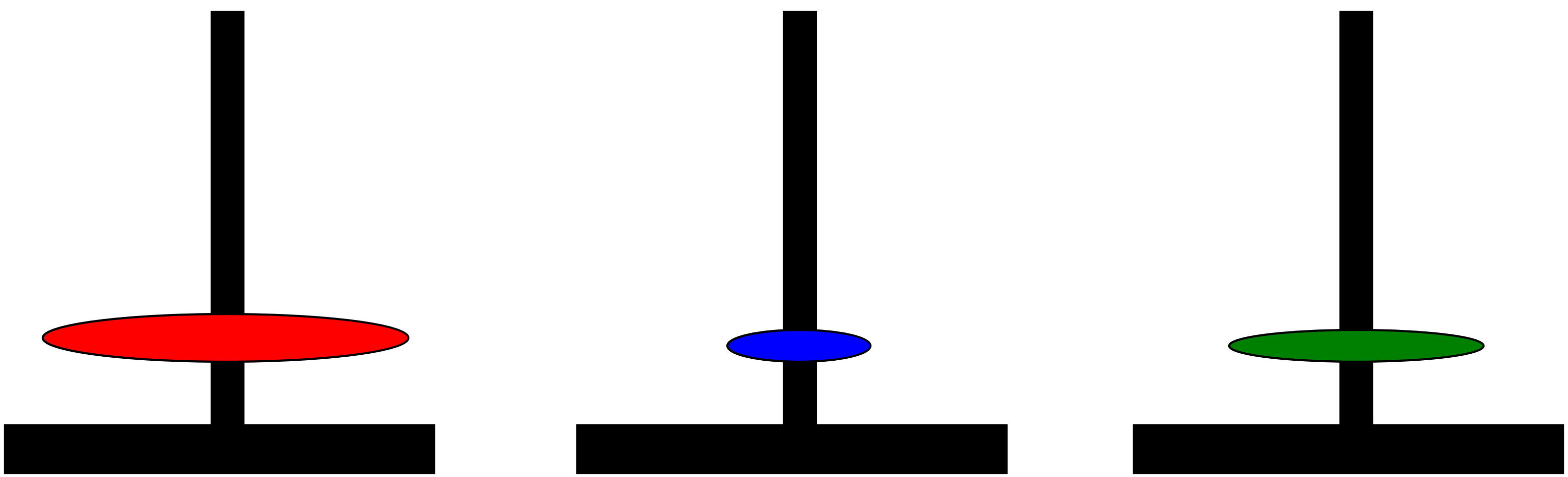
Tower of Hanoi



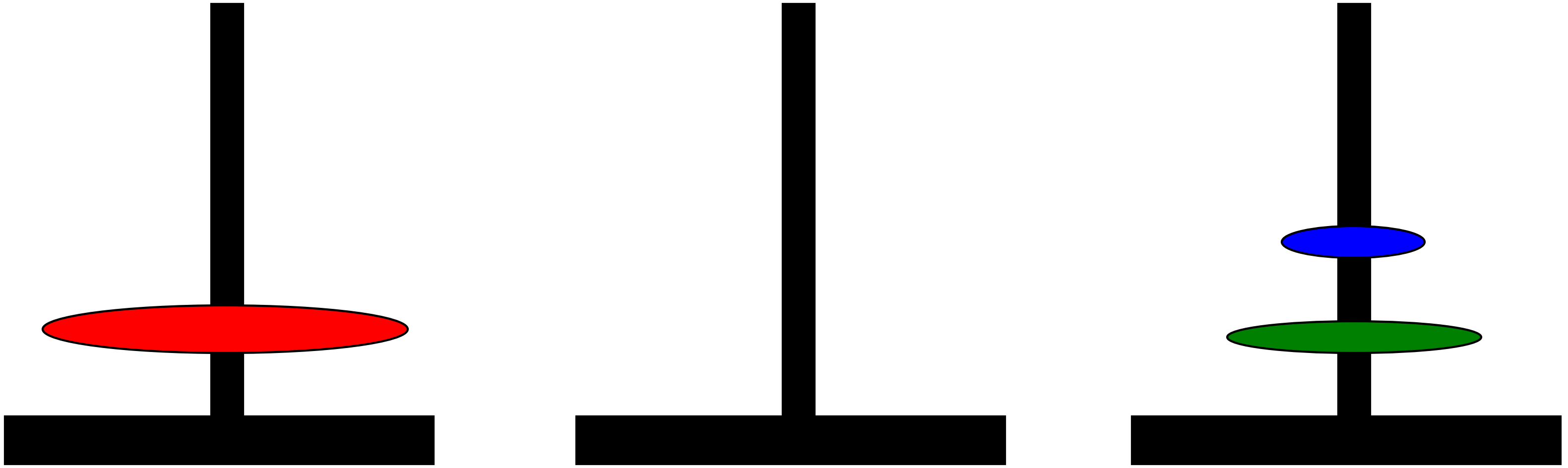
Tower of Hanoi



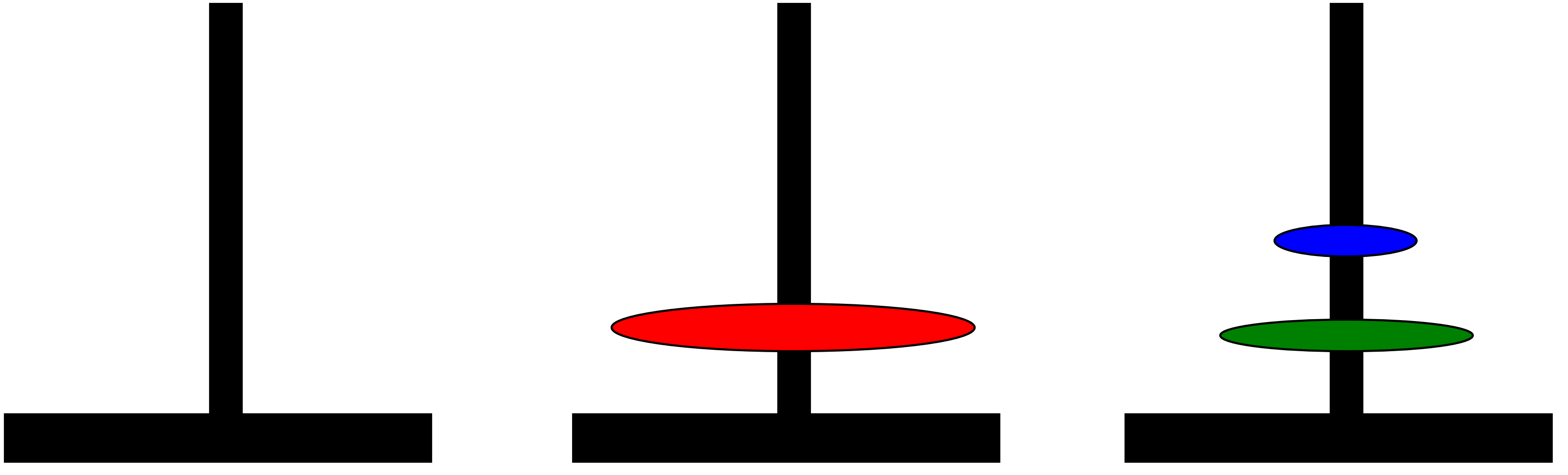
Tower of Hanoi



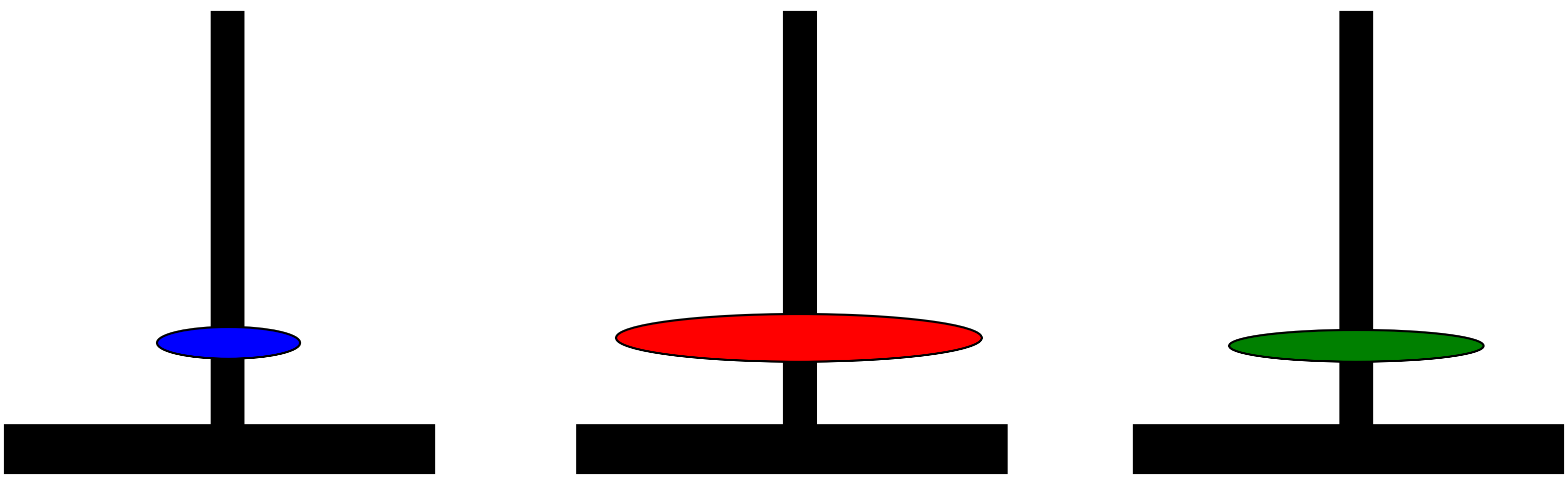
Tower of Hanoi



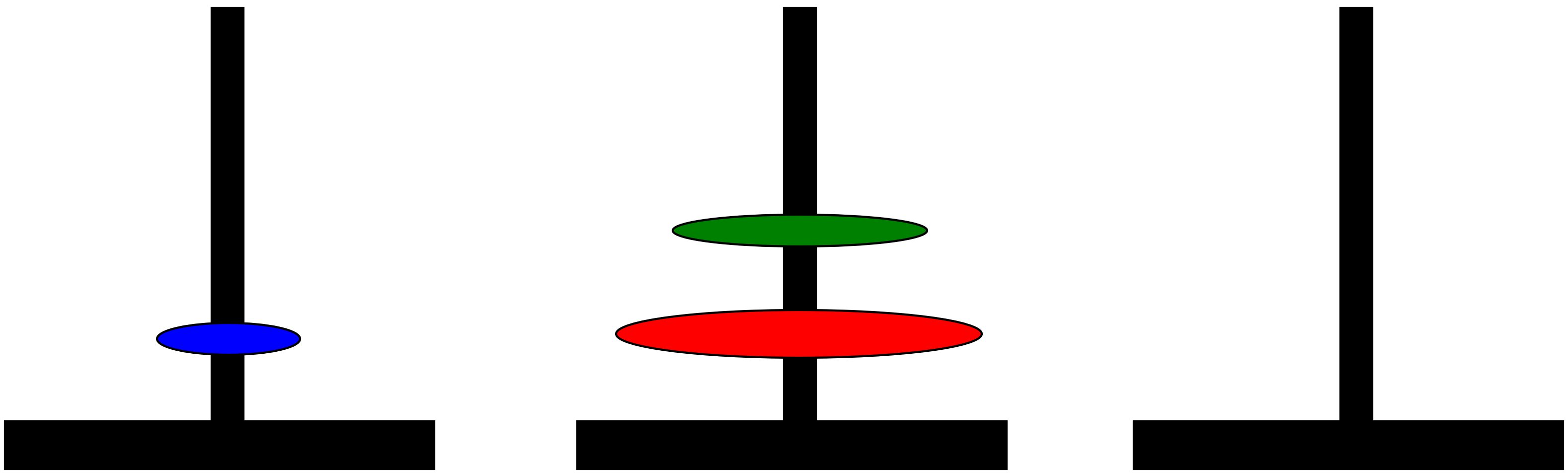
Tower of Hanoi



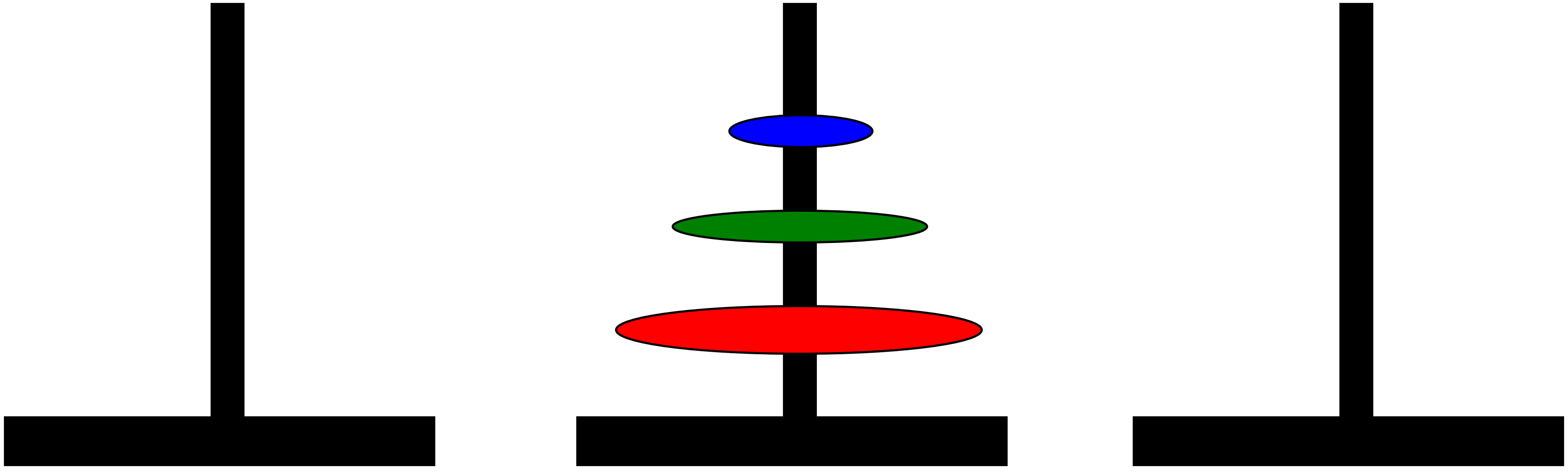
Tower of Hanoi



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:

$$T(1) = 1$$

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

$$\mathbf{T(n) = \Theta(2^n)}$$

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

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

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

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

$\Theta(2^{n})$

If n=64 the 2^{64} seconds about 1.84×10^{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:

$$T(0) = 1$$

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

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

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

$$\mathbf{T(n) = c + T(n/2)}$$

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

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

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

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

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

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

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

Iteration - Mergesort

$$\mathbf{T(n) = n + 2T(n/2)}$$

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

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

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

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

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

… $= in + 2^iT(n/2^i)$ 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

  $T(n) \leq c\, g(n)$, for some $c > 0$ 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:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq c\phi^{n-1} + c\phi^{n-2} \\ &\leq c\phi^{n-2}(\phi + 1) \\ &\leq c\phi^{n-2}(\phi^2) \end{aligned}$$

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

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

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

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

$$\boldsymbol{\Phi + 1 = \Phi^2}$$

The Recursion-Tree method

Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

Used to “guess” a solution for the recurrence

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 is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- 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 $T(n) = T(n/4) + T(n/2) + n^2$:

n^2

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

Example of recursion tree

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

n^2

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

$T(n/16)$ $T(n/8)$ $T(n/8)$ $T(n/4)$

Example of recursion tree

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

n^2

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

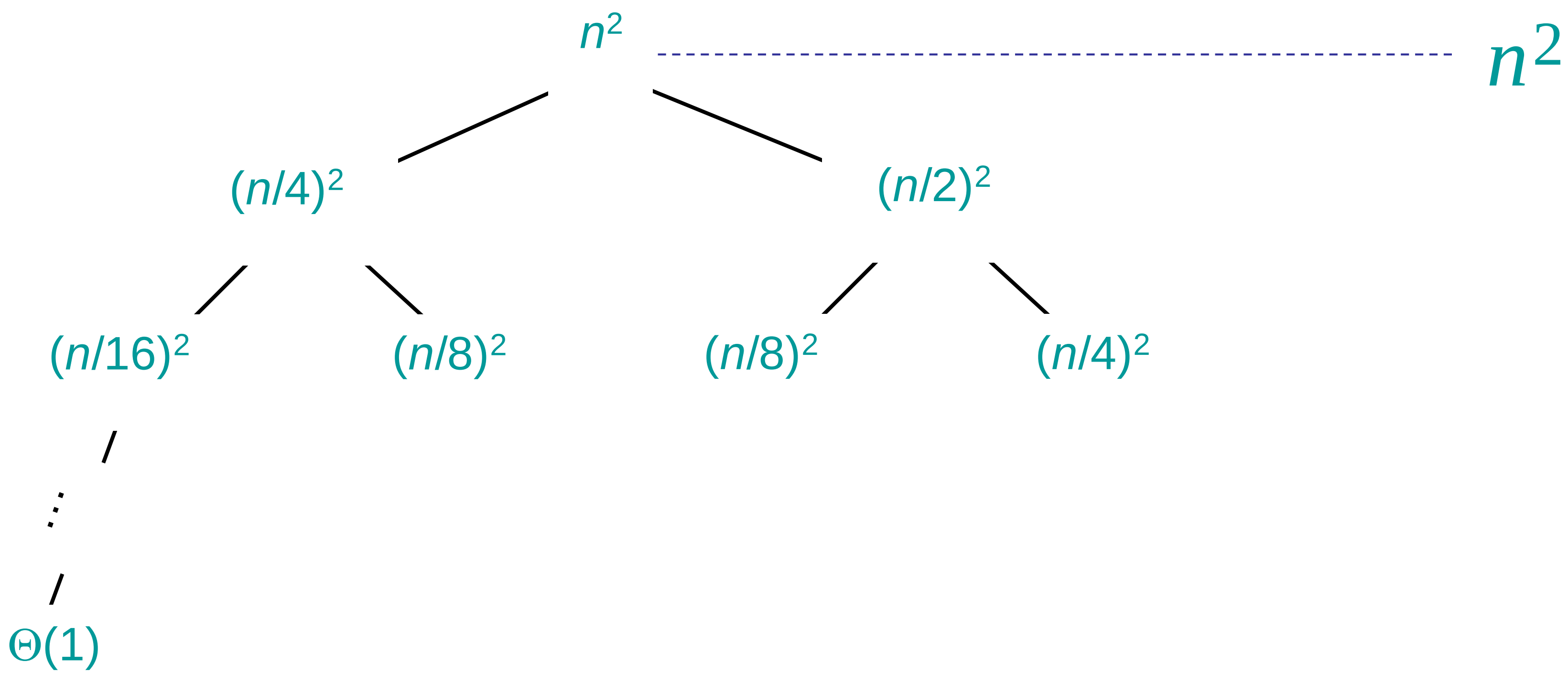
$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$

$\vdots$

$\Theta(1)$

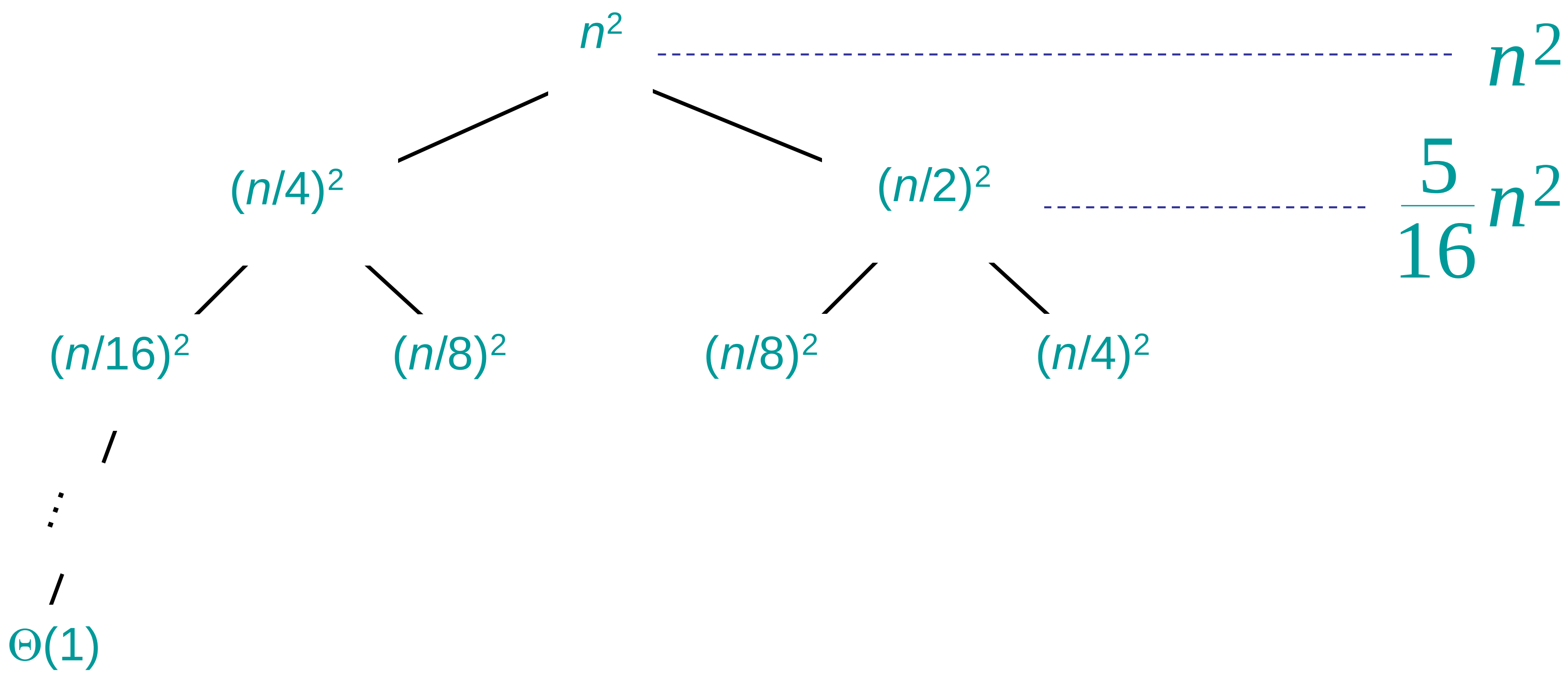
Example of recursion tree

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



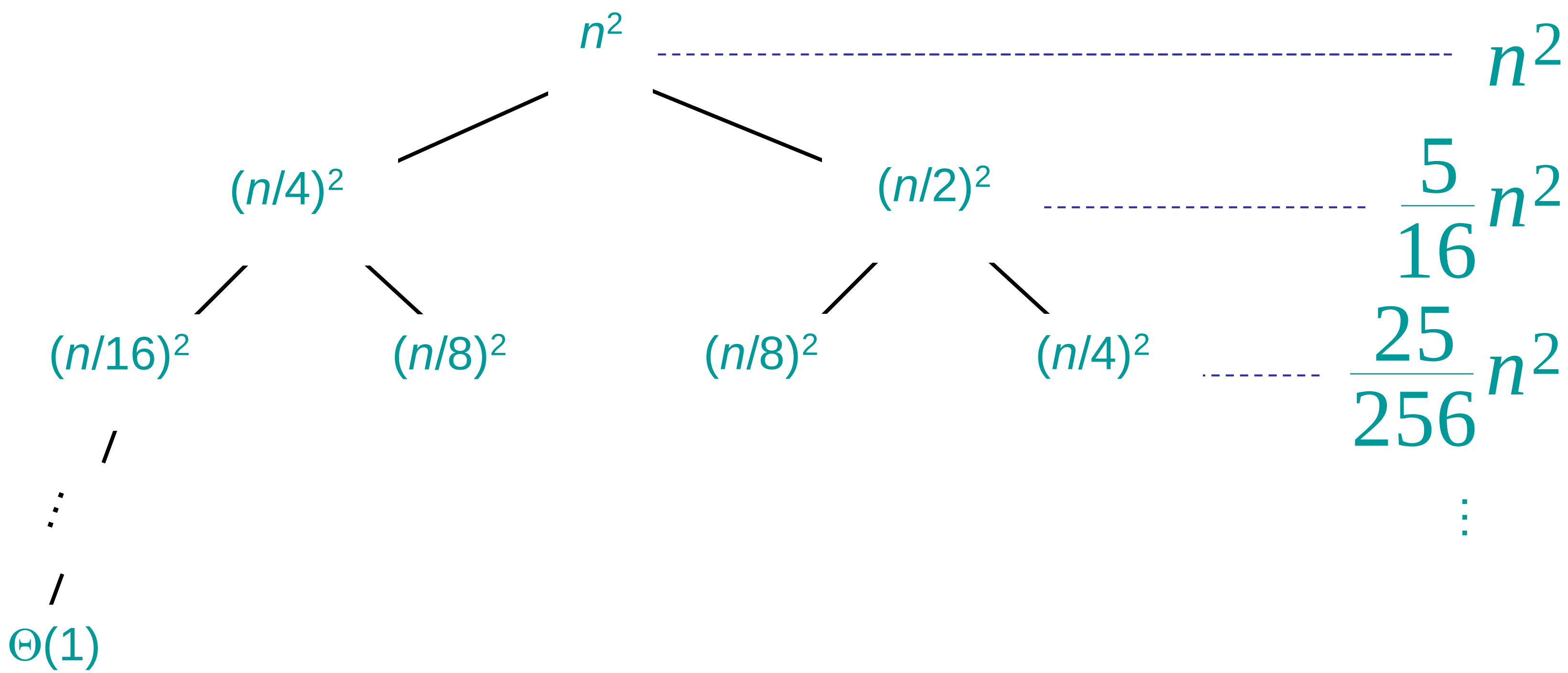
Example of recursion tree

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



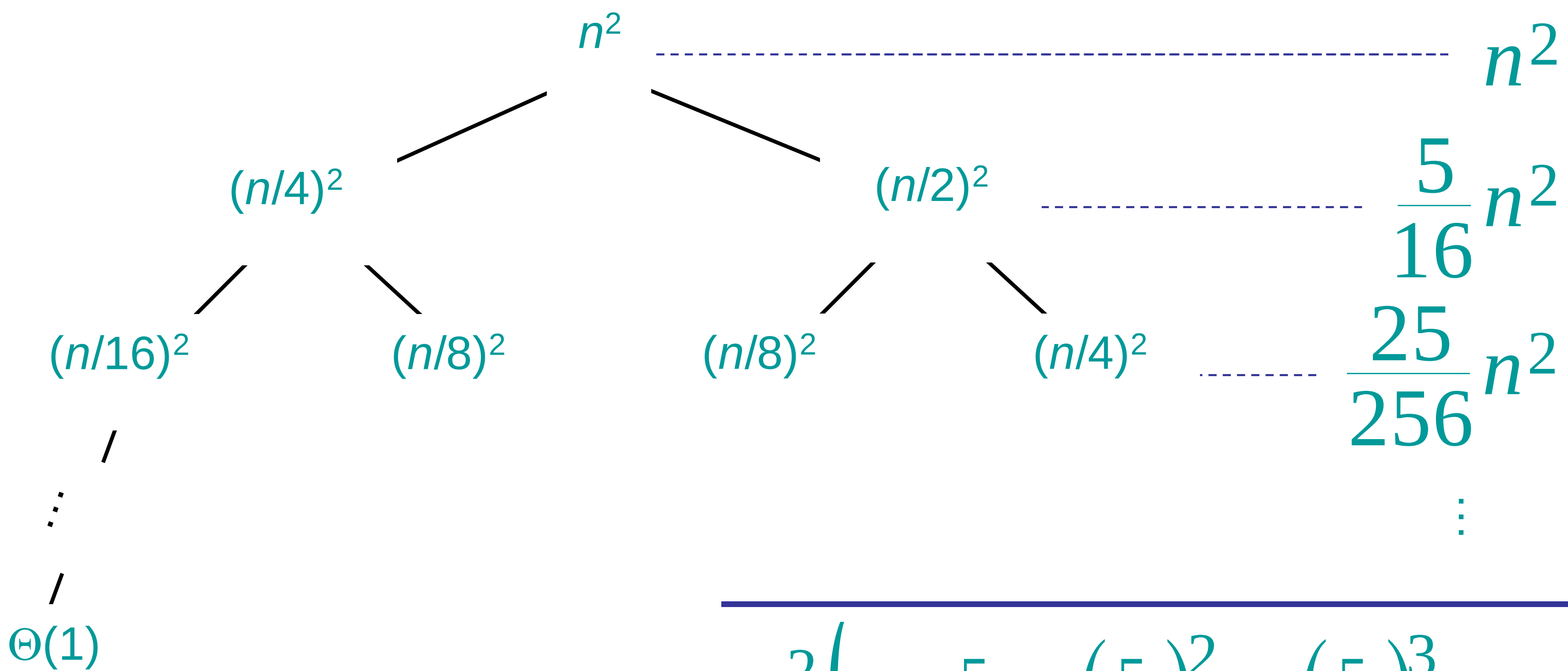
Example of recursion tree

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



Example of recursion tree

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



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

geometric series

$$= \mathrm{O}(n^2)$$

Geometric series

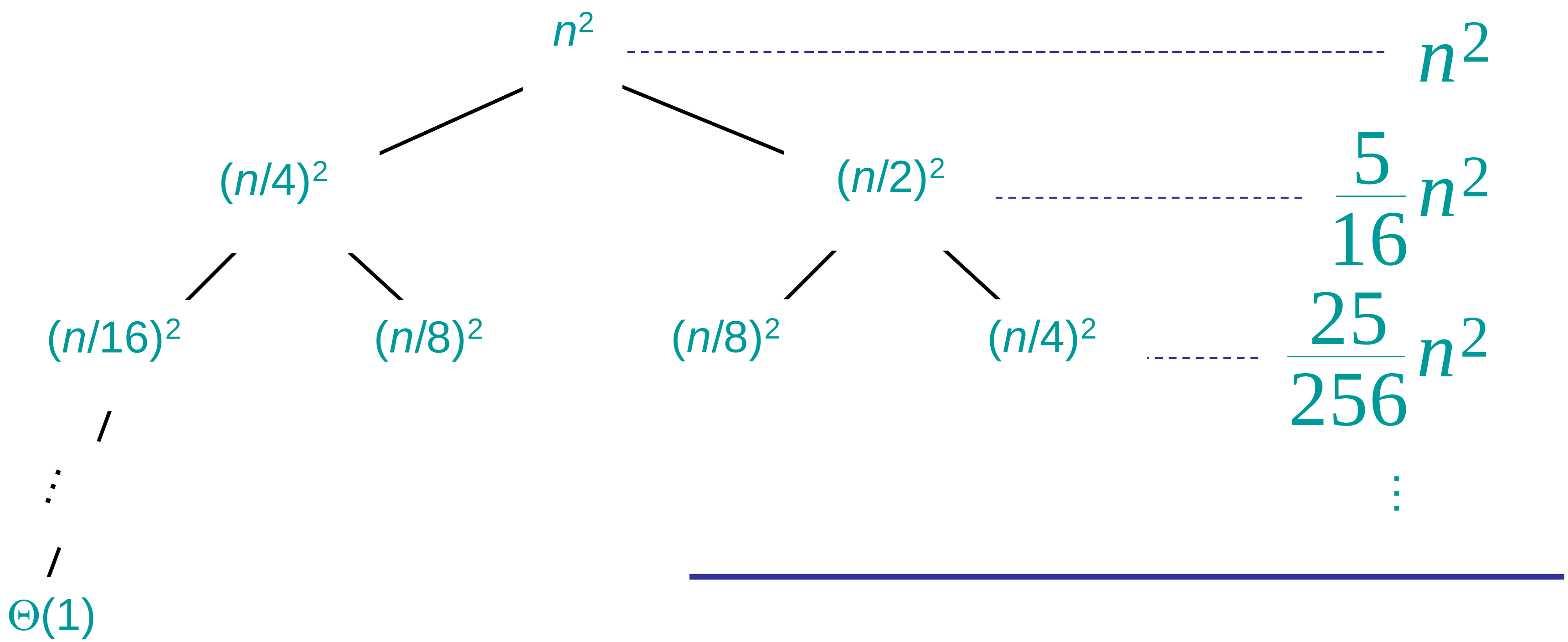
$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x}$$ for $x \neq 1$

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

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

Example of recursion tree

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

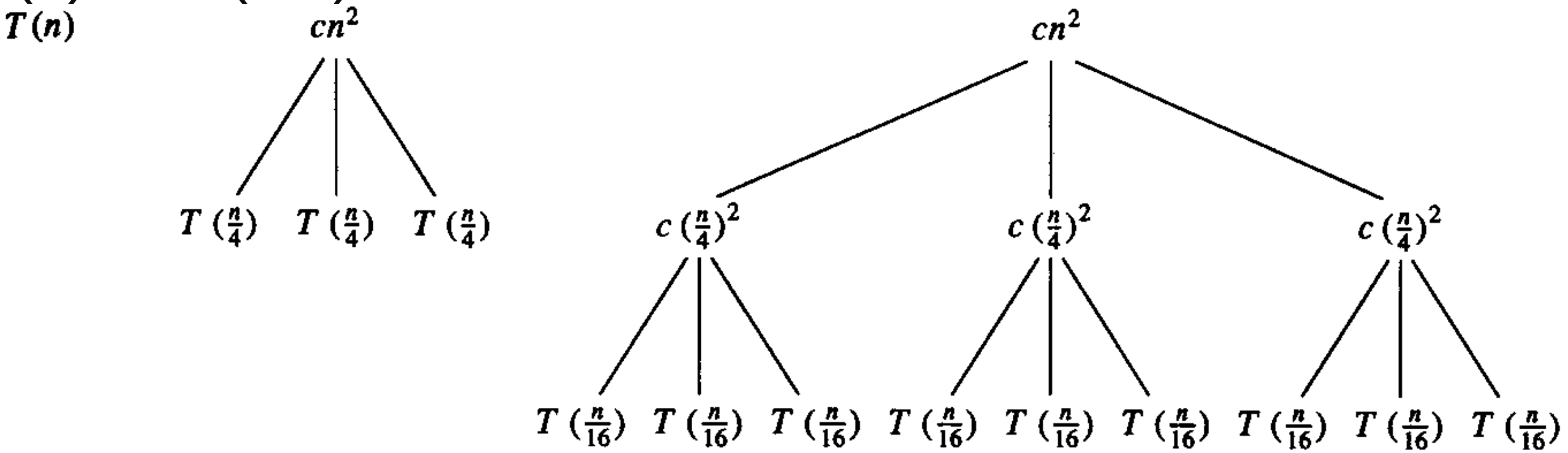


Total $\geq n^2 \quad = \Omega(n^2)$

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

Recursion tree Example 2

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



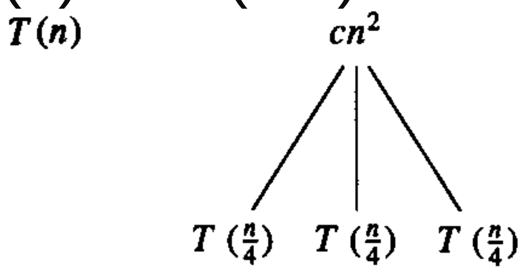
- Subproblem size at level i is: $n/4^i$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level i = $c(n/4^i)^2$
- Number of nodes at level i = $3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes
- Total cost:

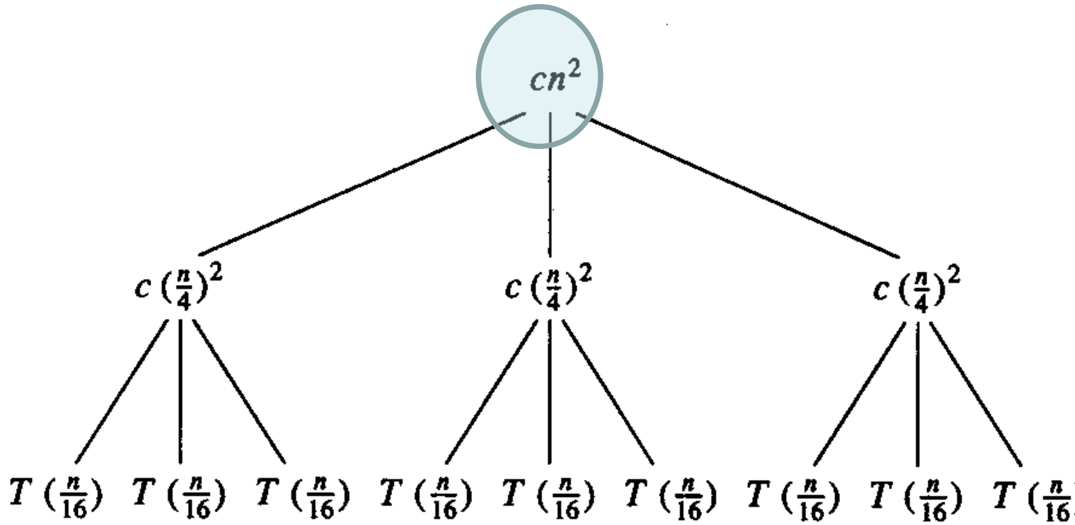
$$T(n) = \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \le \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1-\frac{3}{16}}cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

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

Recursion tree Example 2

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





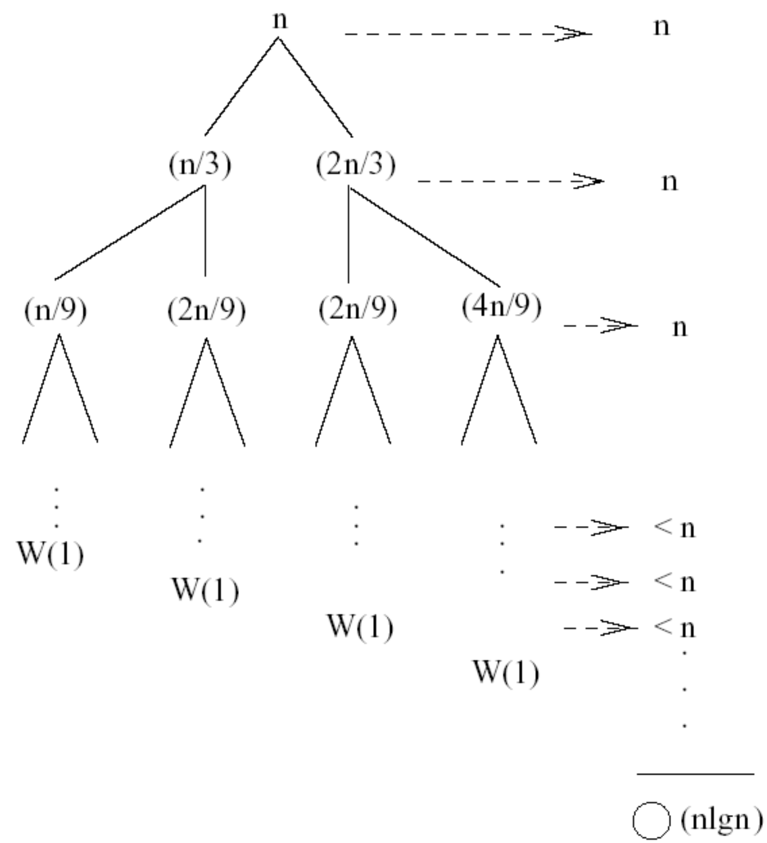
$T(n) = O(n^2)$

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

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

Recursion Tree – Example 3

$T(n) = T(n/3) + T(2n/3) + n$



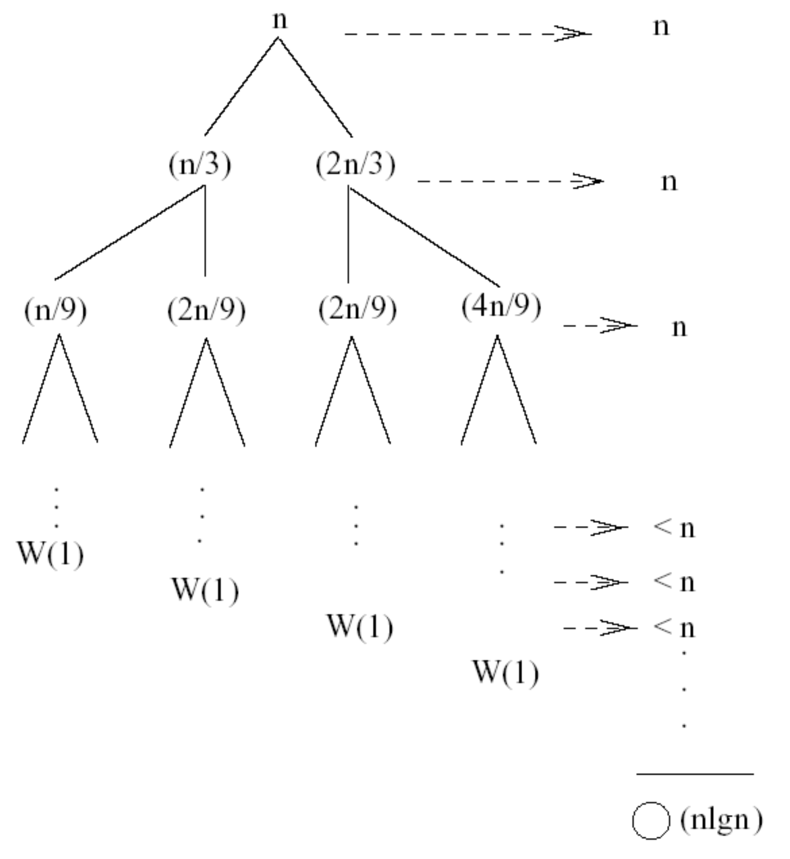
- The longest path from the root to a leaf is: $n \to (2/3)n \to (2/3)^2\, n \to \ldots \to 1$
- Subproblem size hits 1 when $1 = (2/3)^i n \Leftrightarrow i=\log_{3/2}n$
- cost of the problem at level i = n
- Total cost:

$$T(n) < n + n + \ldots = n(\log_{3/2} n) = n\frac{\lg n}{\lg \frac{3}{2}} = O(n \lg n)$$

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

Recursion Tree – Example 3

$T(n) = T(n/3) + T(2n/3) + n$



$T(n) = \Omega(n)$

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

The Master Method

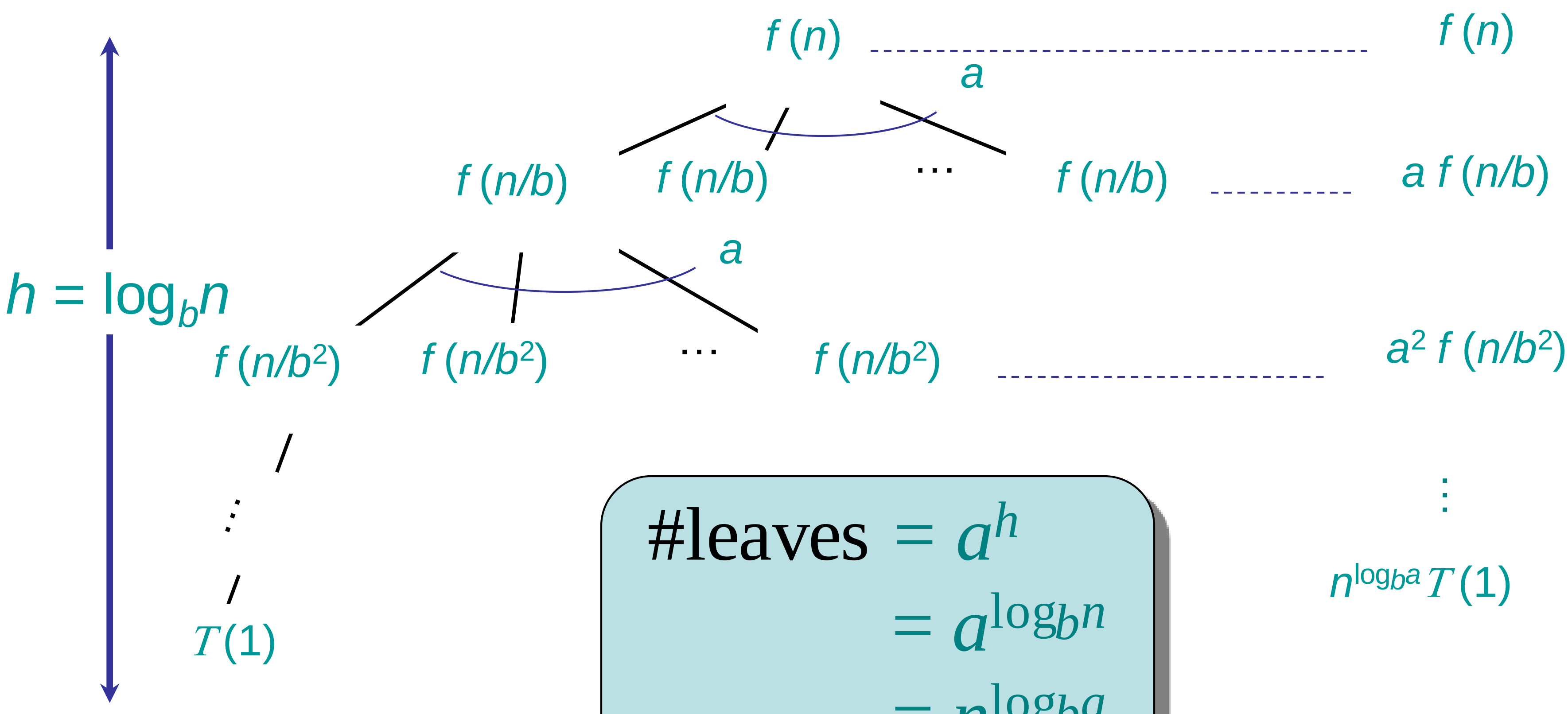
The master method applies to recurrences of the form

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

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

Idea of Master Method

Recursion tree:

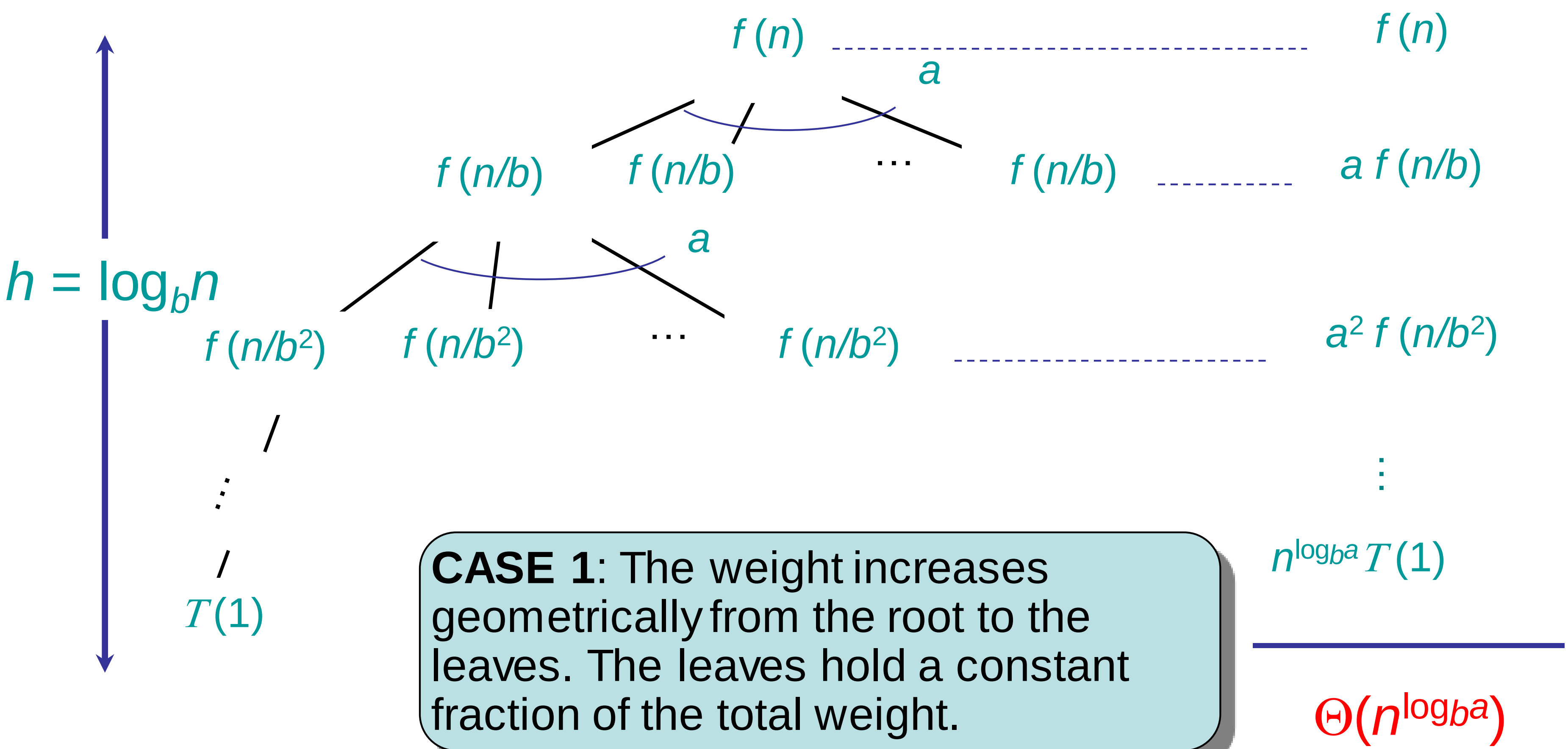


$$\begin{aligned}\text{\#leaves} &= a^h \\ &= a^{\log_b n} \\ &= n^{\log_b a}\end{aligned}$$

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.

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

Case 1

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

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

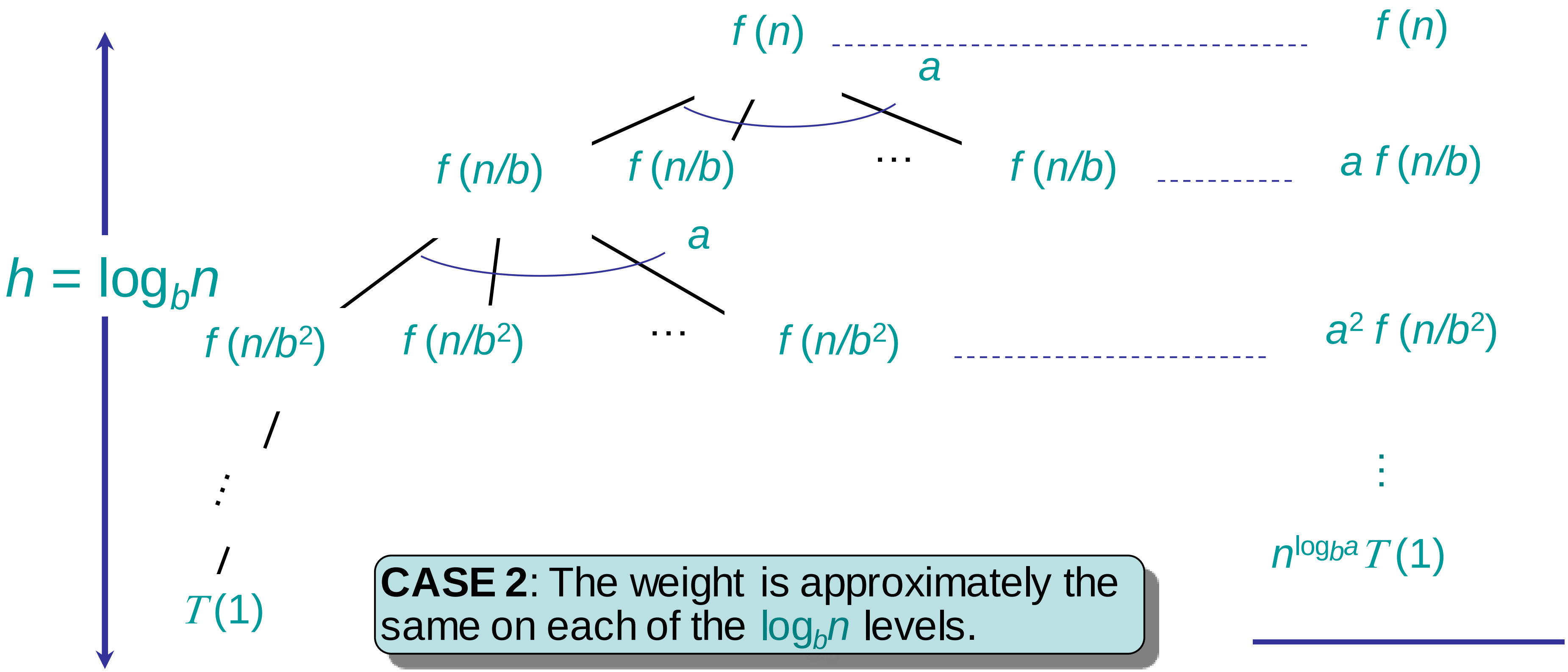
CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

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

Idea of Master Method

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

Recursion tree:



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

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

Case 2

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

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

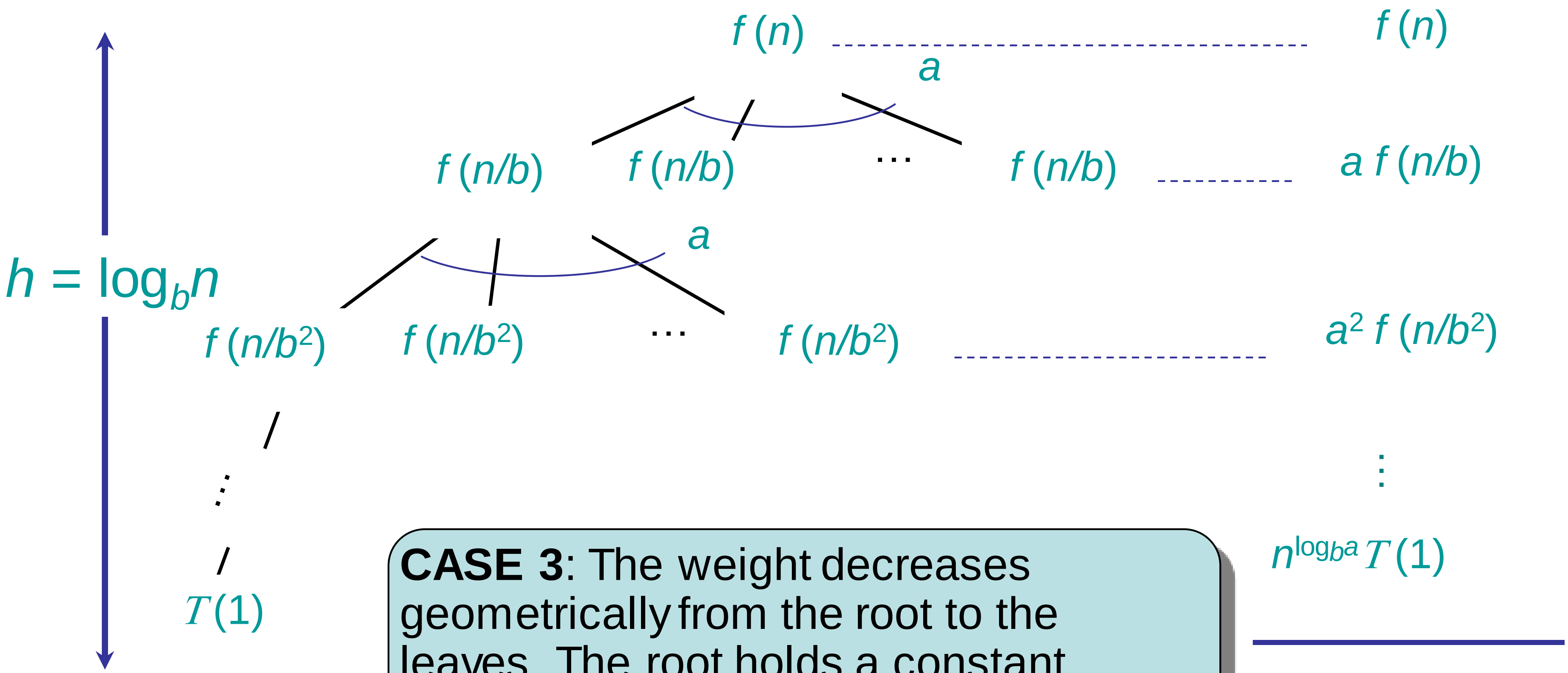
CASE 2: $f(n) = \Theta(n^2)$

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

Idea of master theorem

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

Recursion tree:



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***

$4(cn/2)^3 \leq cn^3$ (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. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

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:

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

regularity

Master Method – Binary Search

$T(n) = T(n/2) + c$ $\quad$ $a = 1,\ b = 2,\ \log_2 1 = 0$

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

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

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

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

Master Method – Example 1

$T(n) = 2T(n/2) + n^2$ $\qquad$ $a = 2$, $b = 2$, $\log_2 2 = 1$

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.

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

$\Leftrightarrow 2\, n^2/4 \leq c\, n^2 \Rightarrow c = ½$ 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 $f(n) = n^{1/2}$

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

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ case 3

check regularity condition:

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

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

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 $f(n) = nn$

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

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 Theorem for *"decrease and conquer"* recurrences of the form

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

for some integer constants a, b > 0, d ≥ 0 .

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

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

Decrease and Conquer: Towers

$$T(n) = 2\,T(n\text{-}1) + 1$$

$$T(n) = a\,T(n\text{-}b) + f(n)$$

a = 2, b= 1, f(n) = 1 so d = 0.

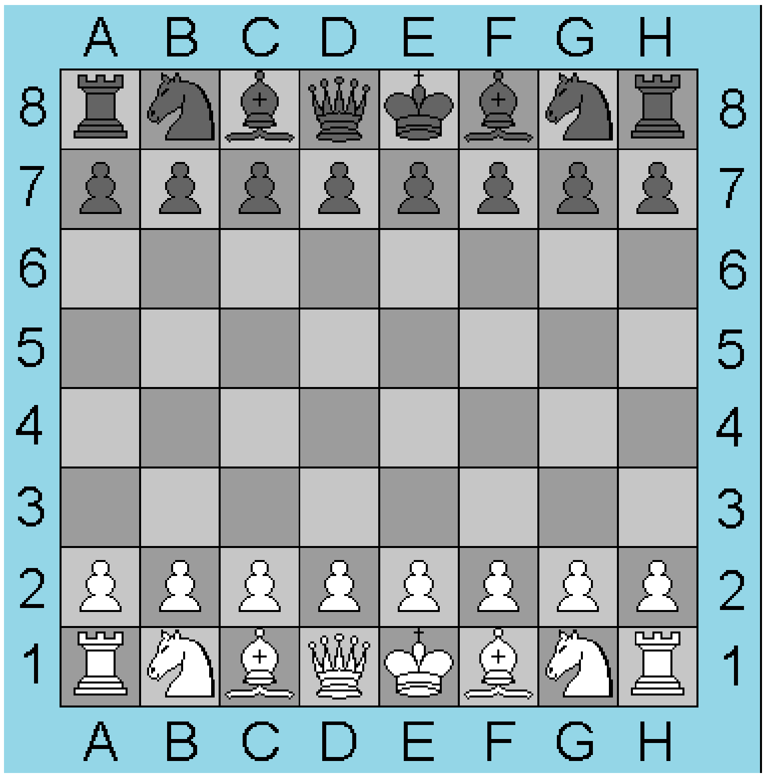
$$T(n) = \begin{cases} O(n^d), & if\ a < 1, \\ O(n^{d+1}), & if\ a = 1 \\ O(n^d a^{n/b}), & if\ a > 1. \end{cases}$$

T(n) is $O(2^n)$ even better *f(n)* is $\Theta(n^d)$ so we could conclude that T(n) is $\Theta(2^n)$.

More Applications

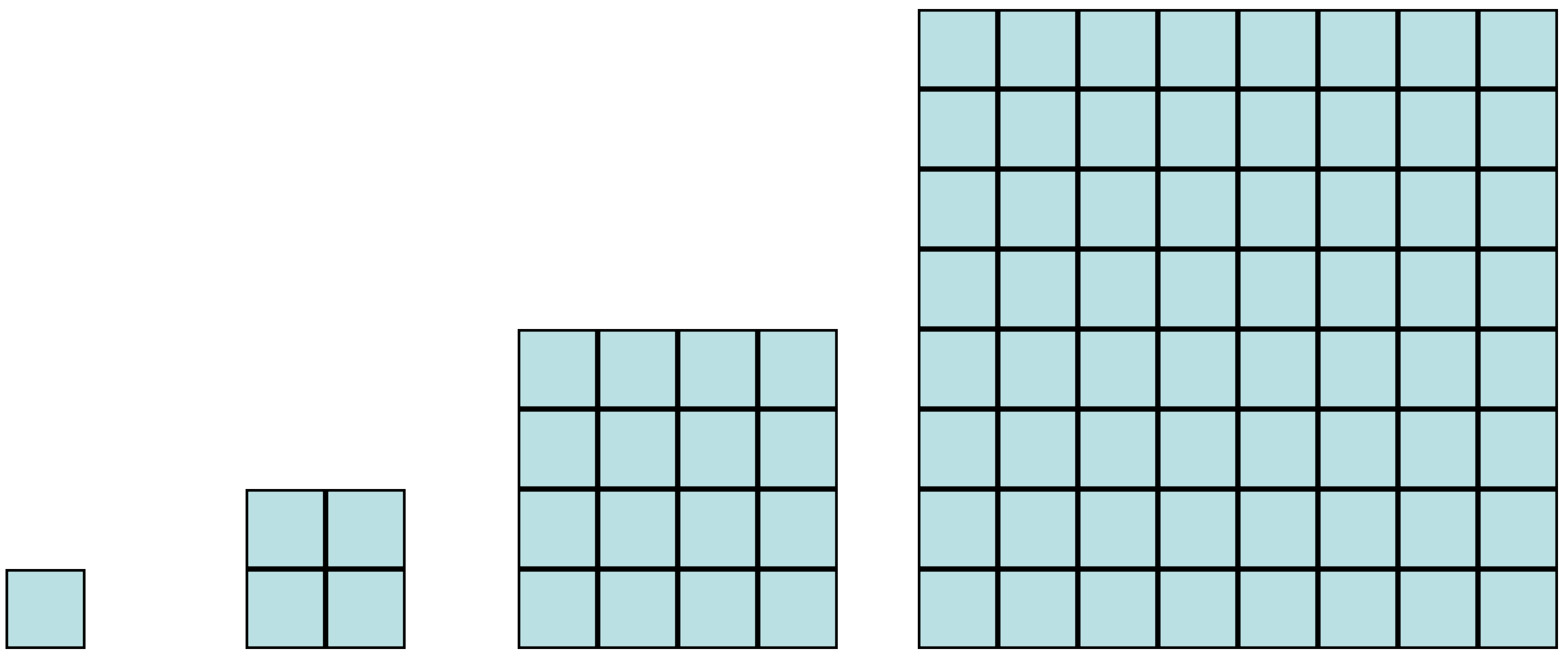
- Tiling
- Skylines

Tiling A Defective chessboard



Our Definition Of A chessboard

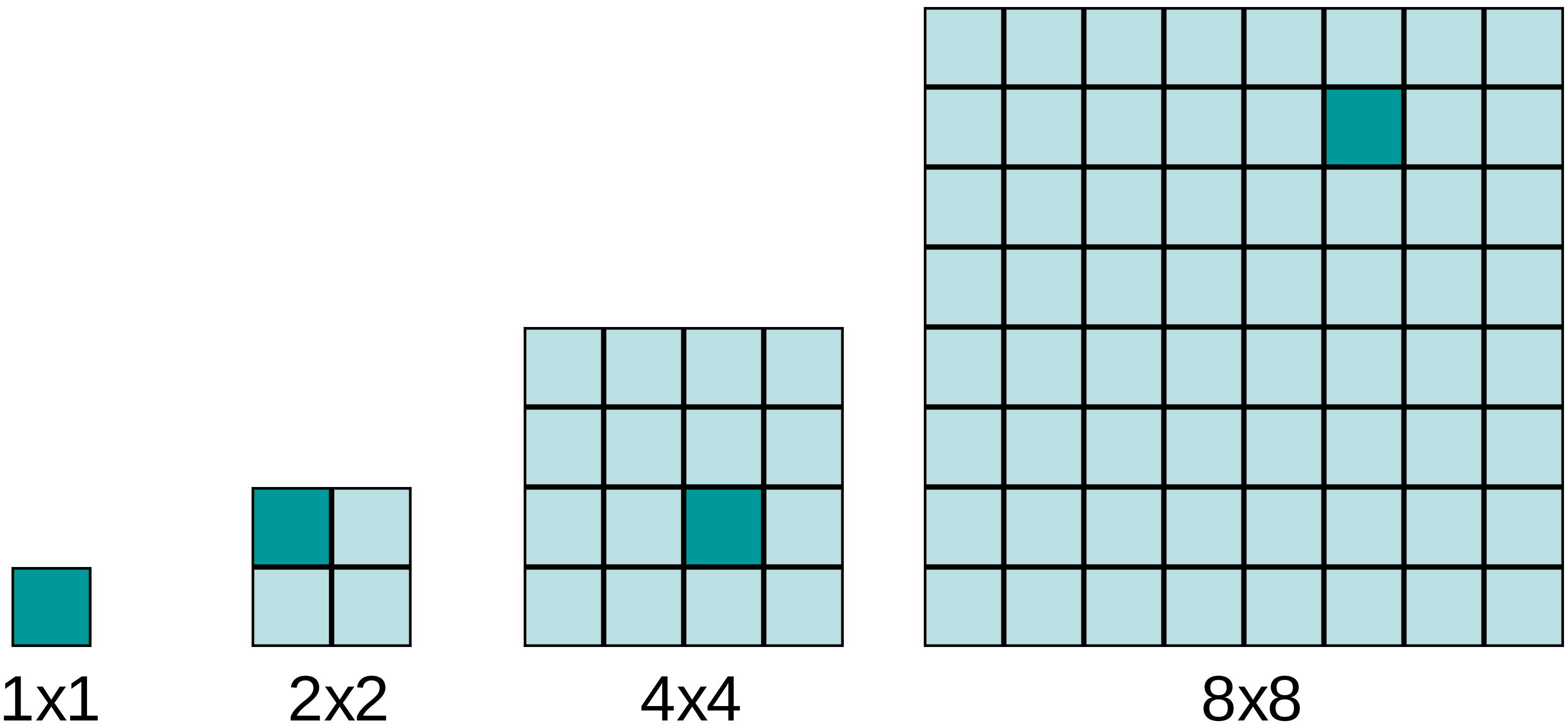
A chessboard is an n x n grid, where n is a power of 2.



1x1 2x2 4x4 8x8

A Defective chessboard

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



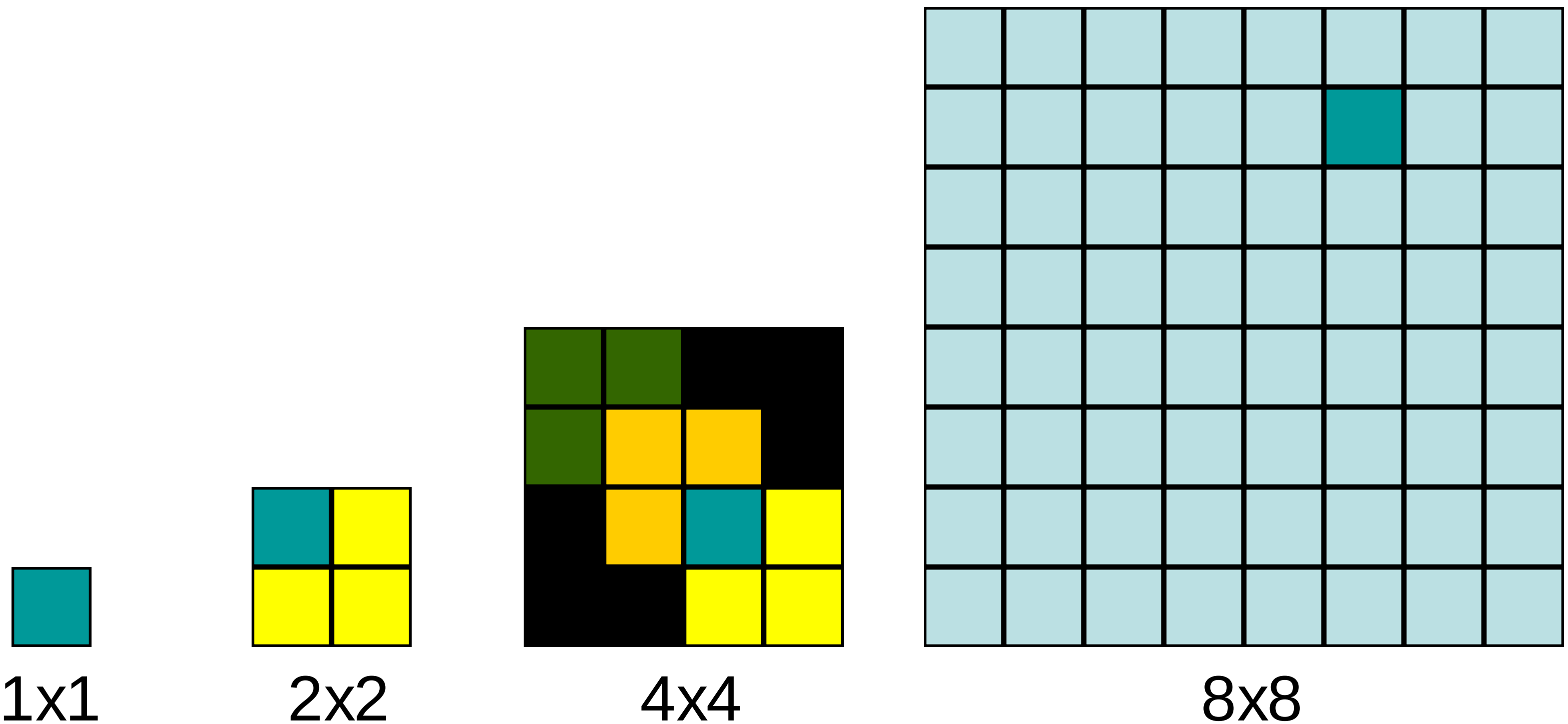
A Triomino

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

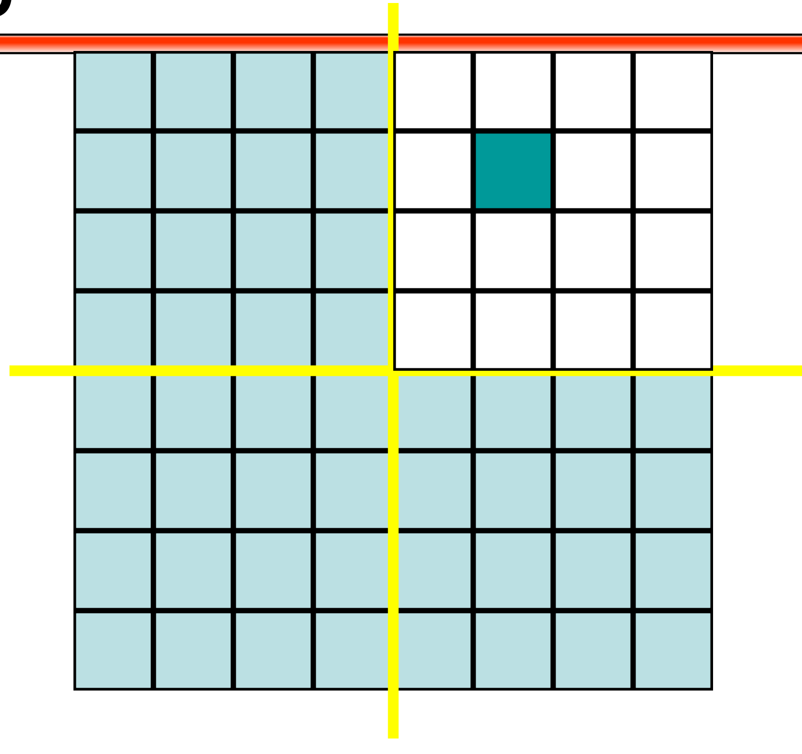
A triomino has four orientations.

Tiling A Defective chessboard

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



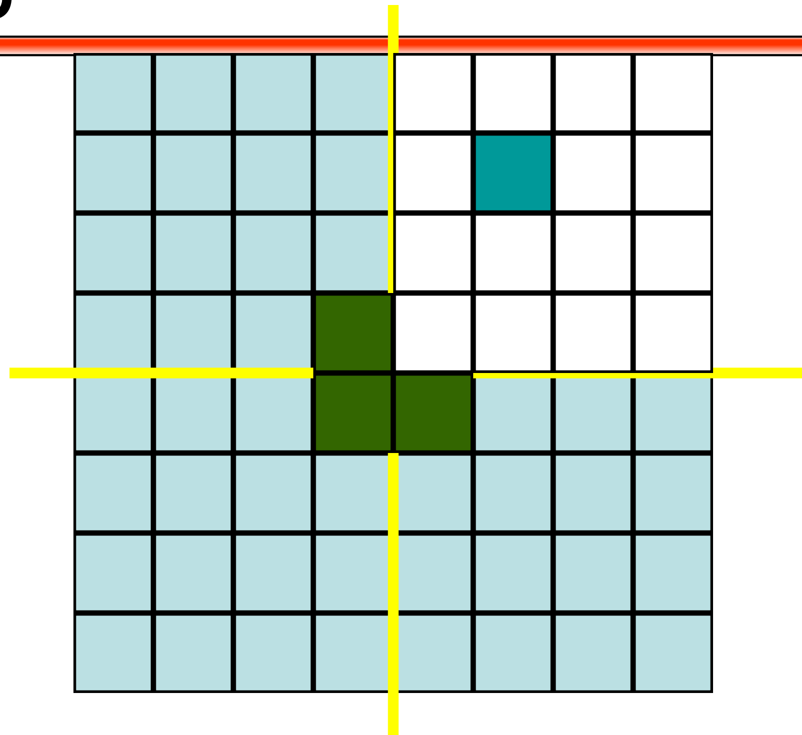
Tiling A Defective chessboard



Divide into four smaller chessboards. 4 x 4

One of these is a defective 4 x 4 chessboard.

Tiling A Defective chessboard



Mane 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^nx2^n board), *L* – location of the 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
   Place one tromino at the center to cut out 3 additional
    holes (orientation based on where existing hole, L, is)
   Let L1, L2, L3, L4 denote the positions of the 4 holes
   Tile(n-1, L1)
   Tile(n-1, L2)
   Tile(n-1, L3)
   Tile(n-1, L4)
```

Recurrence

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

$T(1) = 1,$

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

Substitution Method

$$
\begin{aligned}
T(n) &= 4T(n-1) + c \\
&= 4[4T(n-2) + c] + c \\
&= 4^2\ T(n-2) + 4c + c \\
&= 4^2[4T(n-3) + c] + 4c + c \\
&= 4^3\ T(n-3) + 4^2c + 4c + c \\
&= \ldots \\
&= 4^{n-1}\ T(1) + 4^{n-2}c + \ldots + 4^2c + 4c + c \\
&= 4^n * 1 + 4^{n-1}c + 4^{n-2}c + \ldots + 4^2c + 4c + c \\
&= \Theta(4^n)
\end{aligned}
$$