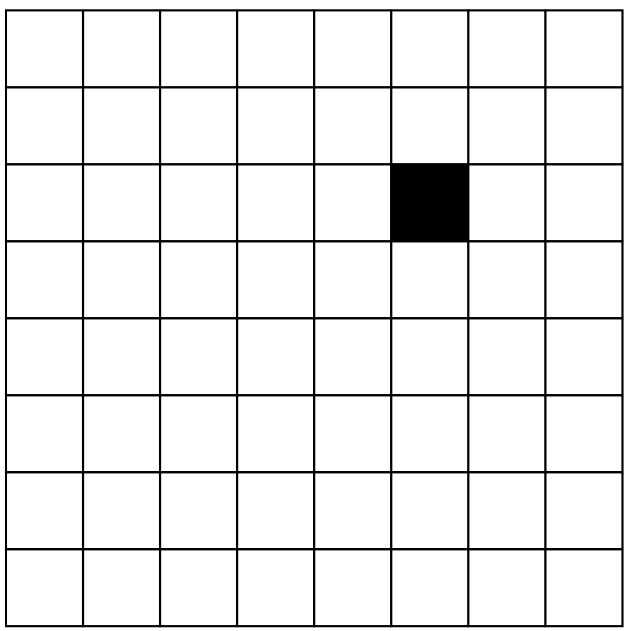
CS 4102: Algorithms
Spring 2020

Lecture 2: Recurrences

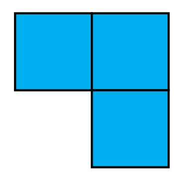
Co-instructors: Robbie Hott and Tom Horton

(These are slides for Horton’s section)

Warm Up



Can you cover an 8×8 grid with 1 square missing using “trominoes?”



Tromino

https://nstarr.people.amherst.edu/trom/puzzle-8by8/

Office Hours

TA Offices: TBD! (They're hired, not on-boarded yet.)

Prof. Horton:

- Mon, and Weds., 1:30-2:30pm
- Tue. and Thu., 10:30-11:30
  - But this week and next, Thu. 10-10:50 due to faculty candidate talks
- Also Thu., 1-2pm

Prof. Hott:

- Mondays and Wednesdays, 11am-12pm
- Tuesdays 3-4pm
- This Week Only: Friday 1-3pm

Today’s Keywords

Recursion

Recurrences

Asymptotic notation and proof techniques

Divide and conquer

Trominoes

CLRS Readings: Chapter 3
- Order classes; math review in 3.2

CLRS Readings: Chapter 4
- 4.1 and 4.2 for today’s lecture; the rest in next lecture

Homework

HW0 due 11pm Tuesday, Jan. 21

- Submit 2 attachments (**zip** and **pdf**)

HW1 released next week

- Written (use LaTeX!)
- Asymptotic notation
- Recurrences
- Divide and conquer

Attendance

How many people are here today?

Naïve algorithm

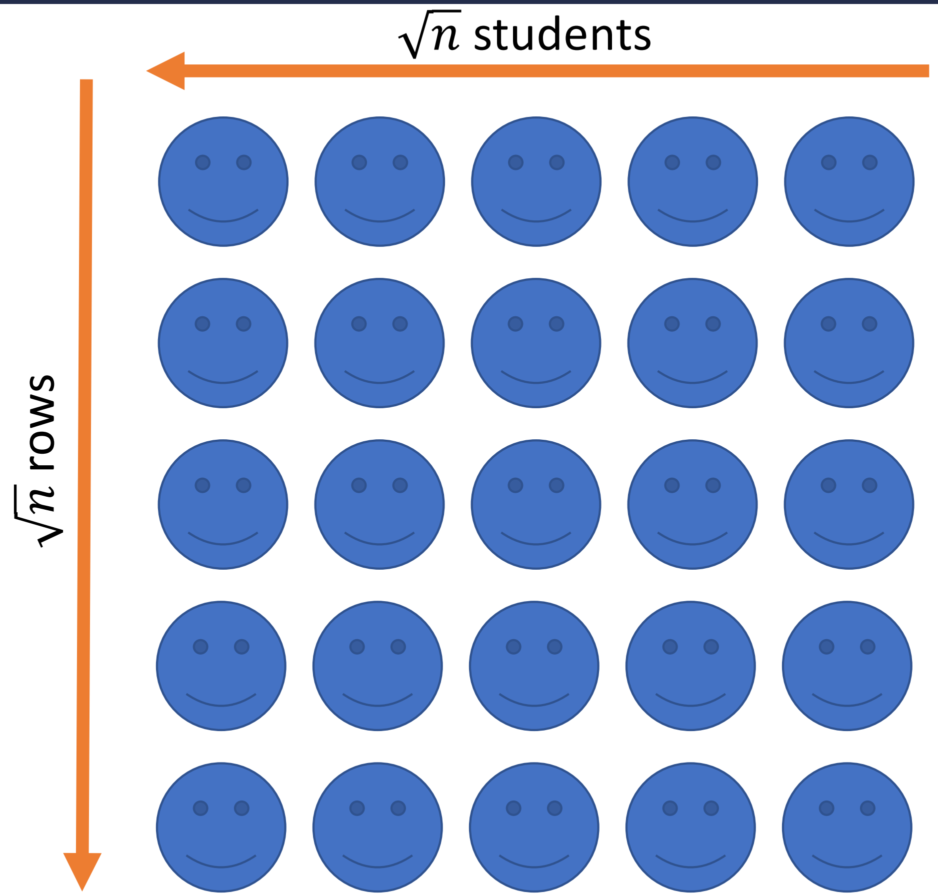
- Everyone stand
- Professor walks around counting people
- When counted, sit down

Complexity?

- Class of n students
- $O(n)$ "rounds"

Other suggestions?

Good Attendance



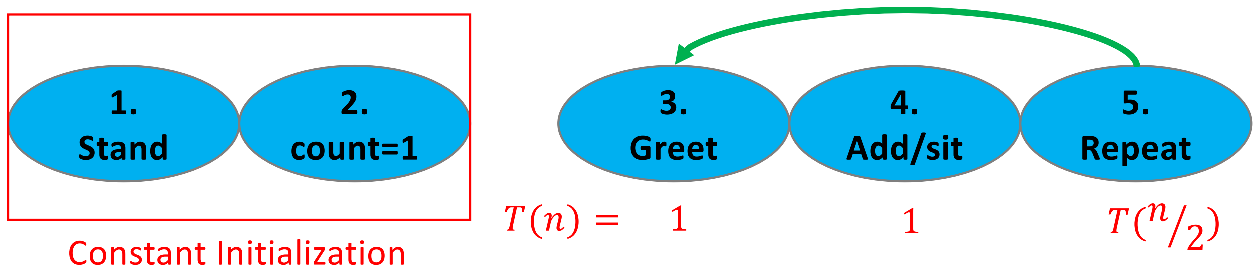
$$O(\sqrt{n})$$

Better Attendance

1. Everyone Stand
2. Initialize your “count” to 1
3. Greet a neighbor who is standing: share your name, full date of birth(pause if odd one out)
4. If you are older: give “count” to younger and sit.
   Else if you are younger: add your “count” with older’s
5. If you are standing and have a standing neighbor, go to Step 3

What was the run time of this algorithm?
What are we going to count?

Attendance Algorithm Analysis



Recurrence

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

How can we “solve” this?

$$T(1) = 3$$

Base case?

Do not need to be exact, asymptotic bound is fine.

Why?

Let's Solve the Recurrence!

Special case: $n = 2^k$

$T(1) = 3$

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

$2 + \cancel{T(n/4)}$

$2 + \cancel{T(n/8)}$

$\vdots$

3

k

$$T(n) = 3 + \sum_{i=1}^{\log_2 n} 2 = 2\log_2 n + 3$$

What if $n \neq 2^k$?

More people in the room ⇒ more time

- $\forall\, 0 < n < m, T(n) < T(m)$
- $T(n) \leq T(m) = T\left(2^{\lceil \log_2 n \rceil}\right) = 2\,\lceil \log_2 n \rceil + 3$

$$= O(\log n)$$

These are unimportant.
Why?

Asymptotic Notation*

$\boldsymbol{O(g(n))}$

- At most within constant of g for large n
- {functions $f \mid \exists$ constants $c, n_0 > 0$ s.t. $\forall n > n_0, f(n) \leq c \cdot g(n)$}
- Set of functions that grow "in the same way" as or more *slowly* than g(n)

$\boldsymbol{\Omega(g(n))}$

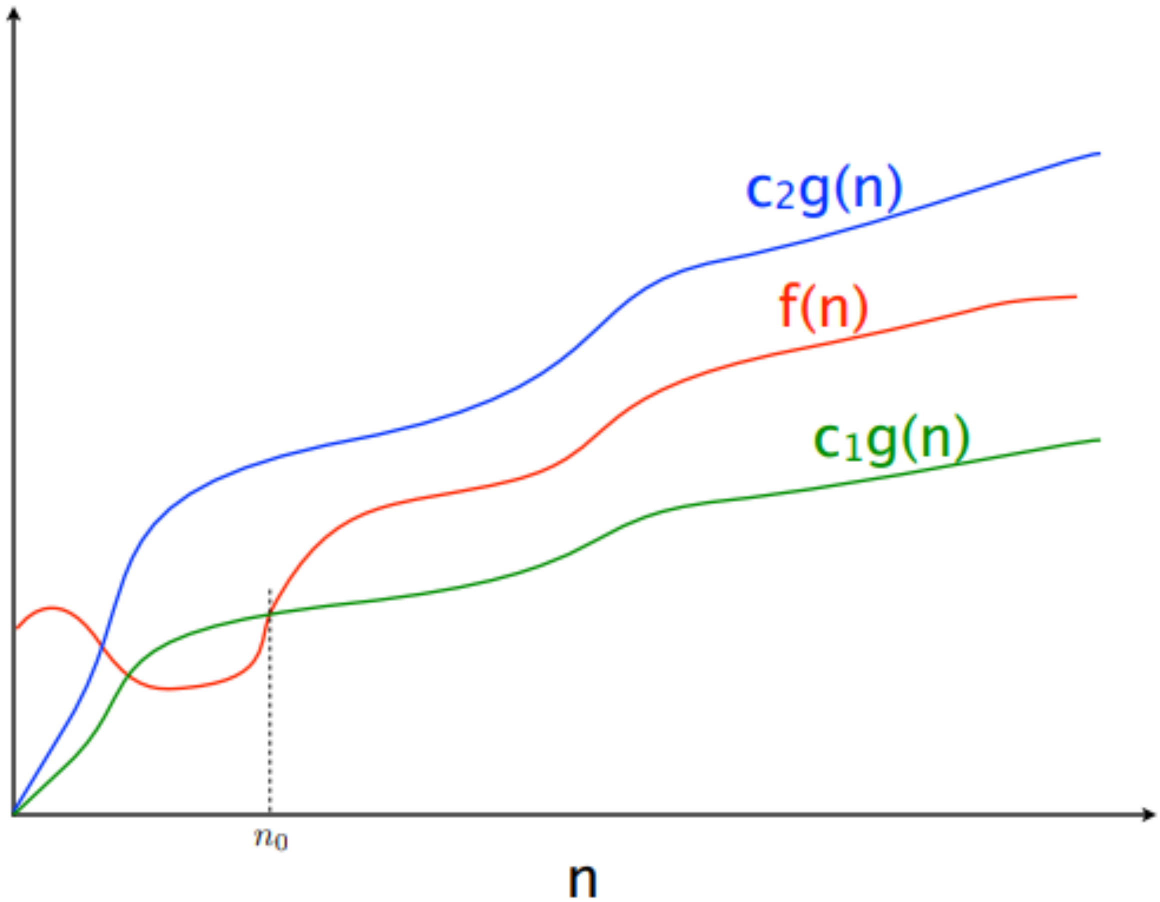
- At least within constant of g for large n
- {functions $f \mid \exists$ constants $c, n_0 > 0$s.t. $\forall n > n_0, f(n) \geq c \cdot g(n)$}
- Set of functions that grow "in the same way" as or more *quickly* than g(n)

$\boldsymbol{\Theta(g(n))}$

- "Tightly" within constant of g for large n
- $\Omega(g(n)) \cap O(g(n))$
- Set of functions that grow "in the same way" as g(n)

*CLRS Chapter 3

Asymptotic Notation



$$f(n) = O(g(n))$$

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

$$f(n) = \Omega(g(n))$$

Asymptotic Bounds

The Sets big oh O(g), big theta Θ(g), big omega Ω(g) – remember these meanings:

- O(g): functions that grow **no faster** than g,
  or **asymptotic upper bound**
- Ω(g): functions that grow **at least as fast** as g,
  or **asymptotic lower bound**
- Θ(g): functions that grow **at the same rate** as g,
  or **asymptotic tight bound**

Asymptotic Notation Example

Direct Proof

Show: $n \log n \in O(n^2)$

Technique: Find $c, n_0 > 0$ s.t. $\forall n > n_0, n \log n \leq c \cdot n^2$

Proof: Let $c = 1, n_0 = 1$. Then,
$n_0 \log n_0 = (1) \log (1) = 0,$
$c\, n_0^2 = 1 \cdot 1^2 = 1,$
$0 \leq 1.$

$\forall n \geq 1, \log(n) < n \Rightarrow n \log n \leq n^2 \quad \square$

Asymptotic Notation Example

Show: $n^2 \notin O(n)$

Indirect Proof

Technique: Contradiction

Proof: Assume $n^2 \in O(n)$. Then $\exists c, n_0 > 0 \text{ s.t. } \forall n > n_0, n^2 \leq cn$
Some such constant c must exist. Can we derive it?

For all $n > n_0 > 0$, by our assumption, we know:
$cn \geq n^2$,
$c \geq n$.

Since c is dependent on n, it cannot be a constant.
Contradiction. Therefore $n^2 \notin O(n)$. □

Proof Techniques

Direct Proof ✓

- From the assumptions and definitions, directly derive the statement

Indirect Proof (Proof by Contradiction) ✓

- Assume the statement is true, then find a contradiction

Proof by Cases

Induction

More Asymptotic Notation

$\boldsymbol{o(g(n))}$

- Smaller than *any* constant factor of g for sufficiently large n
- {functions f : $\forall$ constants $c > 0$, $\exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)$}
- Set of functions that always grow more slowly than g(n)

Equivalently, ratio of $\frac{f(n)}{g(n)}$ is decreasing and tends towards 0:

$$f(n) \in o\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

More Asymptotic Notation

$\boldsymbol{o(g(n))}$

- Smaller than *any* constant factor of g for sufficiently large n
- $\{$functions $f : \forall$ constants $c > 0, \exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)\}$
- Set of functions that always grow more slowly than g(n)

$\boldsymbol{\omega(g(n))}$

- Greater than *any* constant factor of g for large n
- $\{$functions $f : \forall$ constants $c > 0, \exists n_0$ such that $\forall n > n_0, f(n) > c \cdot g(n)\}$
- Set of functions that always grow more quickly than g(n)

Equivalently, $f(n) \in \omega\big(g(n)\big) \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Another Asymptotic Notation Example

Show: $n \log n \in o(n^2)$

Direct Proof

Proof Technique: Show the statement directly, using either definition

- $\lim_{n \to \infty} \frac{n \log n}{n^2} = \lim_{n \to \infty} \frac{\log n}{n} = 0$ (why is this true?)

- Equivalently, for every constant $c > 0$, we can find an n_0 such that $\frac{\log n_0}{n_0} = c$. Then for all $n > n_0$, $n \log n < c\, n^2$ since $\frac{\log n}{n}$ is a decreasing function

  $\forall$ constants $c > 0$, $\exists n_0$ such that $\forall n > n_0, f(n) < c \cdot g(n)$

Summary: Using Limit Definition

Comparing f(n) and g(n) as n approaches infinity, calculate this:

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

If the result….

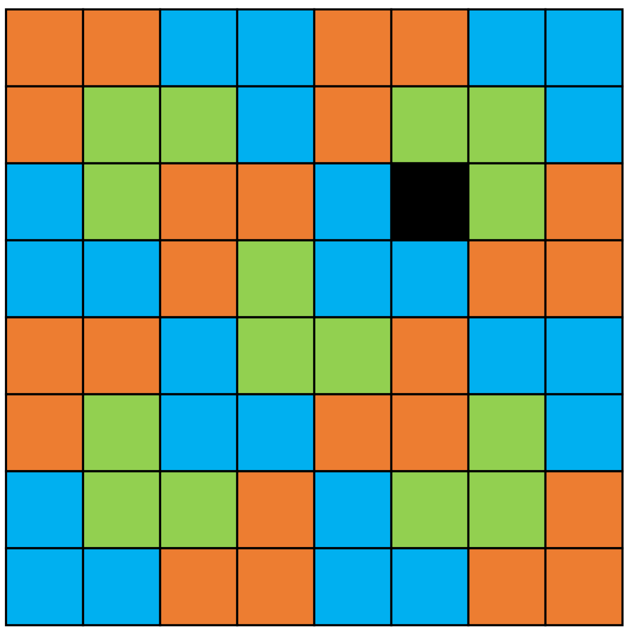
- $< \infty$, including the case in which the limit is 0 then
  $f \in O(g)$
- > 0, including the case in which the limit is ∞ then
  $f \in \Omega(g)$
- $= c$ and $0 < c < \infty$ then
  $f \in \Theta(g)$
- $= 0$ then $f \in o(g)$ read as "little oh of g"
- $= \infty$ then $f \in \omega(g)$ read as "little omega of g"

A Few Miscellaneous Things….

- Keep in mind that order classes are sets of functions.
  Computer scientists might be considered sloppy with our notation.
  We write: $f(n) = \Theta(g(n))$ when we mean: $f(n) \in \Theta(g(n))$
- Why have O(n) and Θ(n) and Ω(n) and o(n)? When do we use which?
  - Depends on what you want to communicate!
    Why so we have $\leq$ and $=$ and $\geq$ and $<$?
- What if algorithm has multiple parts with different order classes?
- lg n $\in$ o(n^{α}) for any $\alpha > 0$, including fractional powers
- $n^k \in o(c^n)$ for any k > 0 and any c > 1
  - powers of n grow more slowly than any exponential function c^n

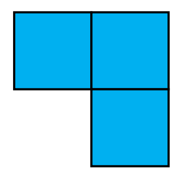
Back to Trominoes

A solution!



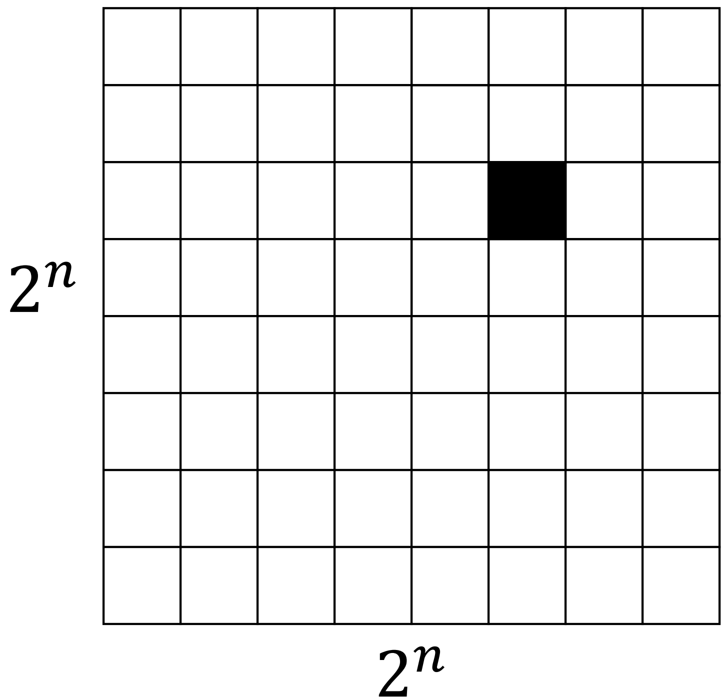
Can you cover an 8×8 grid with 1 square missing using “trominoes?”

What about a 4x4 grid? 2x2? ☺



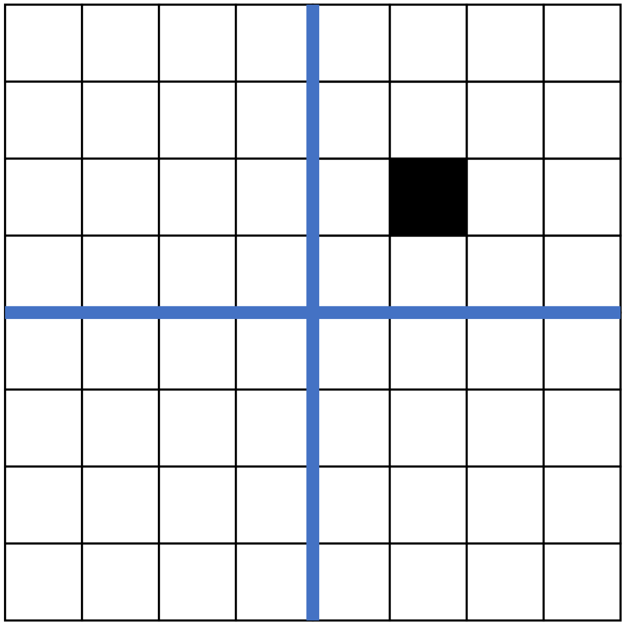
Tromino

Trominoes Puzzle Solution



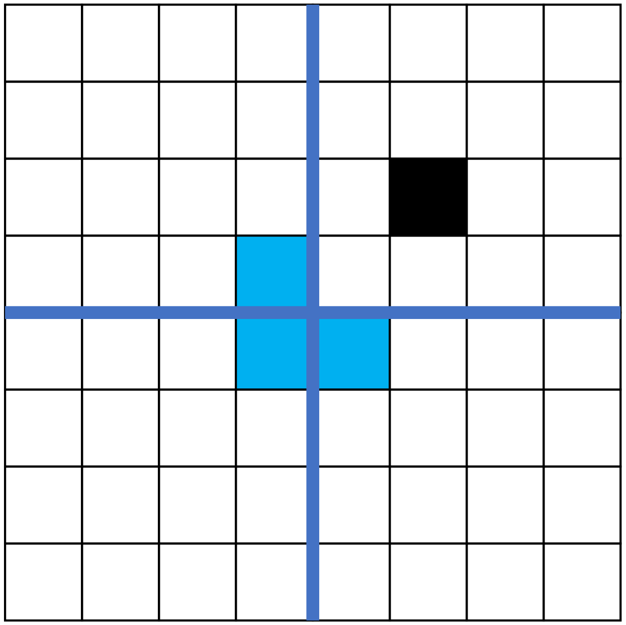
What about larger boards?

Trominoes Puzzle Solution



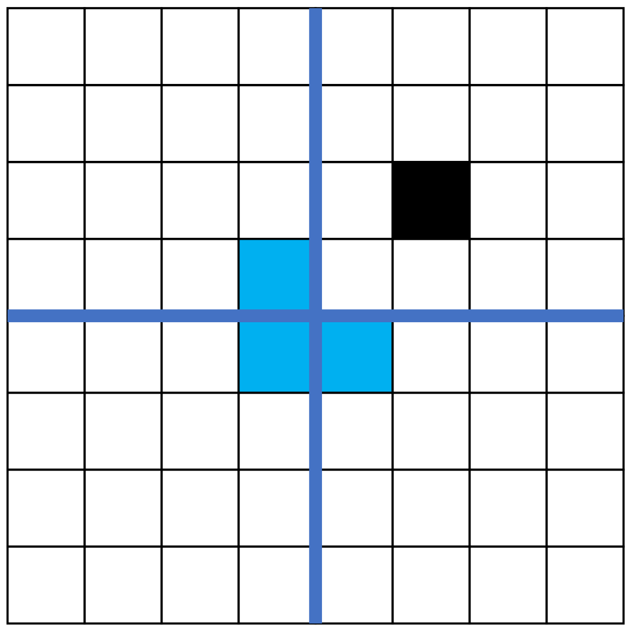
Divide the board into quadrants

Trominoes Puzzle Solution



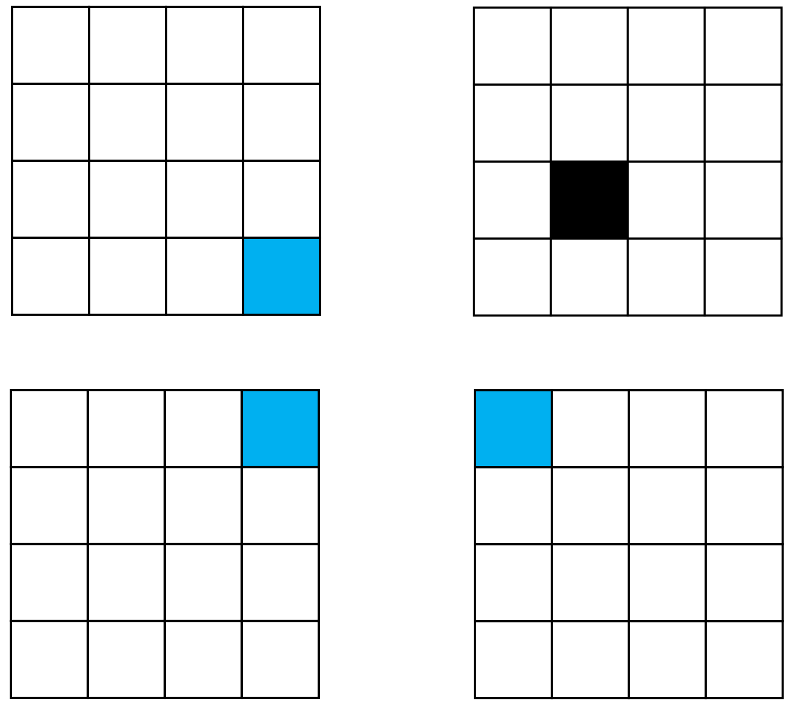
Place a tromino to occupy the three quadrants without the missing piece

Trominoes Puzzle Solution



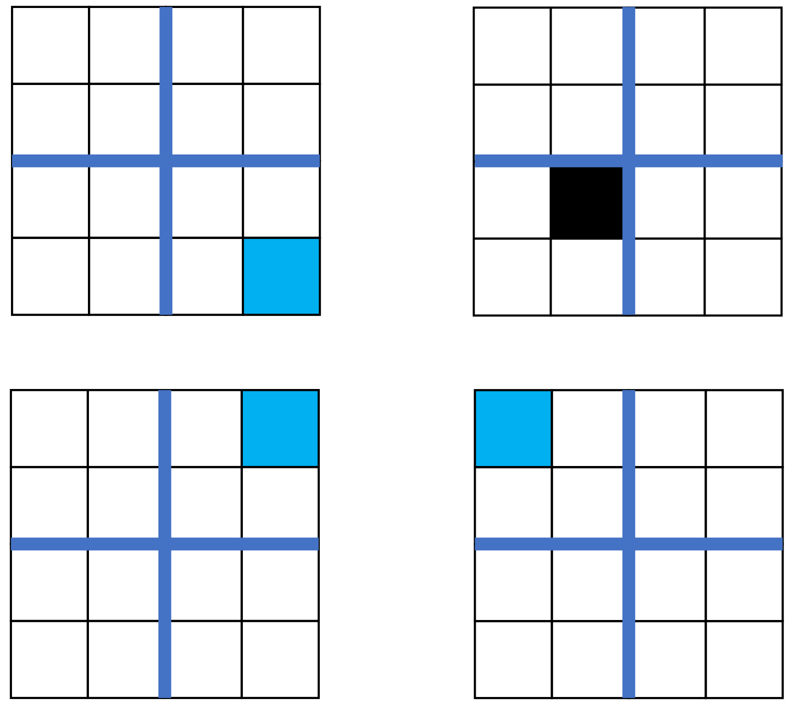
Place a tromino to occupy the three quadrants without the missing piece

Trominoes Puzzle Solution



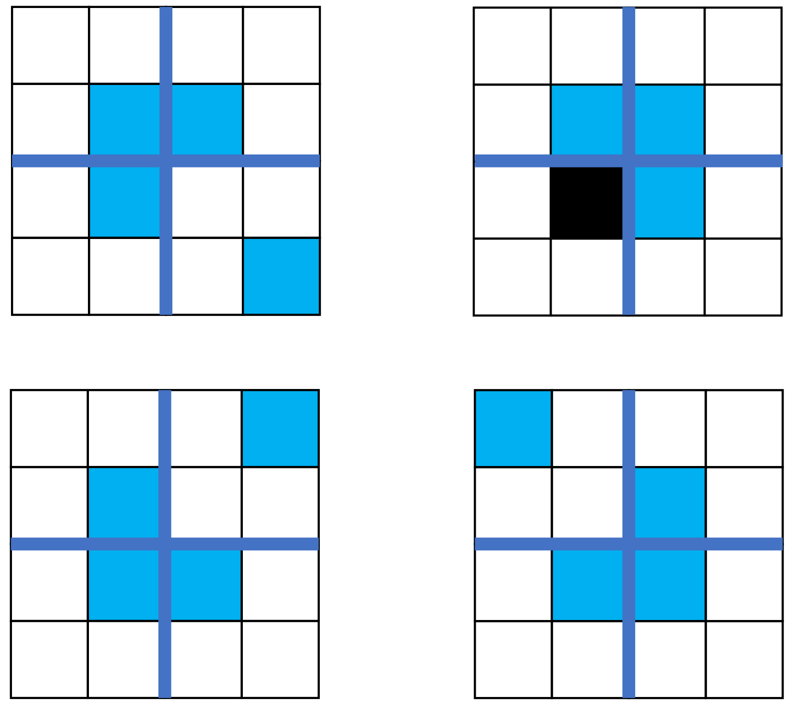
Observe: Each quadrant is now a smaller subproblem!

Trominoes Puzzle Solution



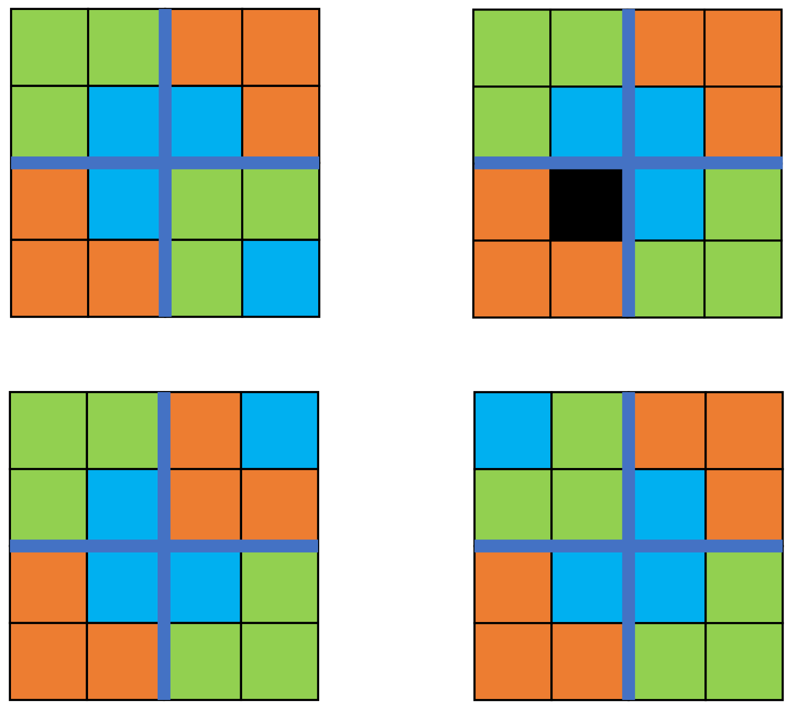
Solve **Recursively**

Trominoes Puzzle Solution



Solve **Recursively**

Trominoes Puzzle Solution



Our first algorithmic technique!

Divide and Conquer

[CLRS Chapter 4]

Divide:

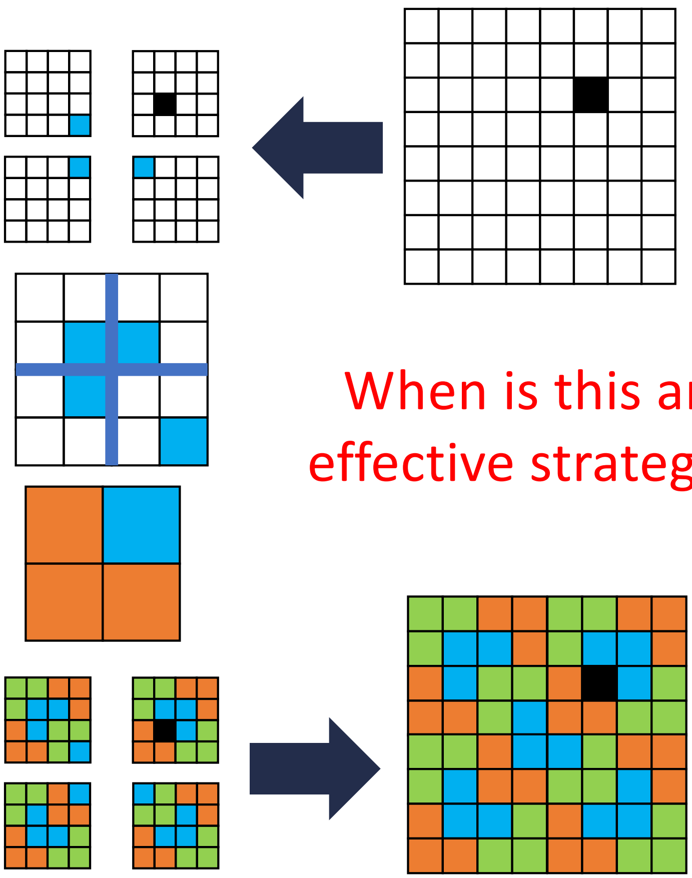
- Break the problem into multiple subproblems, each smaller instances of the original

Conquer:

- If the suproblems are "large":
  - Solve each subproblem recursively
- If the subproblems are "small":
  - Solve them directly (base case)

Combine:

- Merge solutions to subproblems to obtain solution for original problem



When is this an effective strategy?

Analyzing Divide and Conquer

1. Break into smaller subproblems
2. Use recurrence relation to express recursive running time
3. Use asymptotic notation to simplify

Divide: $D(n)$ time

Conquer: Recurse on smaller problems of size $s_1, \ldots, s_k$

Combine: $C(n)$ time

Recurrence:

- $T(n) = D(n) + \sum_{i \in [k]} T(s_i) + C(n)$

So… You've come up with a clever Divide and Conquer Algorithm!

Is it efficient compared to other solutions?

You have its T(n). But you what order class does that belong to?

$T(n) \in \Theta(???)$

Goal: Reduce recurrence to closed form.

There are several techniques!

Some easier than others (but can't always be used)

Techniques

Tree get a picture of recursion

Guess/Check guess and use induction to prove

"Cookbook" MAGIC!

Substitution substitute in to simplify

Merge Sort

Divide:

- Break n-element list into two lists of $^{n}/_{2}$ elements

Conquer:

- If $n > 1$:
  - Sort each sublist recursively
- If $n = 1$:
  - List is already sorted (base case)

Combine:

- Merge together sorted sublists into one sorted list

Merge

Combine: Merge sorted sublists into one sorted list

We have:

- 2 sorted lists (L_1, L_2)
- 1 output list (L_{out})

While (L_1 and L_2 not empty):
    If $L_1[0] \leq L_2[0]$:
        L_{out}.append(L_1.pop())
    Else:
        L_{out}.append(L_2.pop())
L_{out}.append(L_1)
L_{out}.append(L_2)

$O(n)$

Analyzing Merge Sort

1. Break into smaller subproblems
2. Use recurrence relation to express recursive running time
3. Use asymptotic notation to simplify

Divide: 0 comparisons

Conquer: recurse on 2 small subproblems, size $\frac{n}{2}$

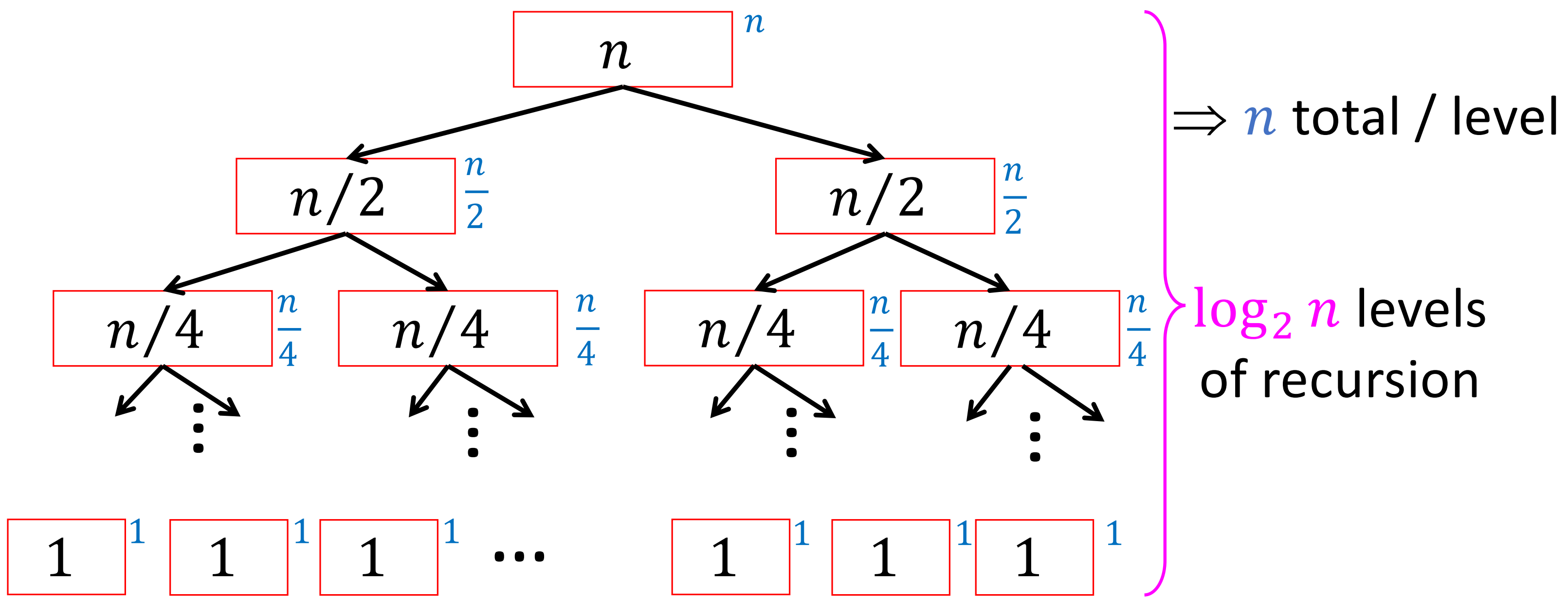
Combine: n comparisons

Recurrence: $T(n) = 2\,T\left(\frac{n}{2}\right) + n$

Practice: solve by substitution (like we did for ”attendance”)

Tree method

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



$$T(n) = \sum_{i=1}^{\log_2 n} n = n \log_2 n$$