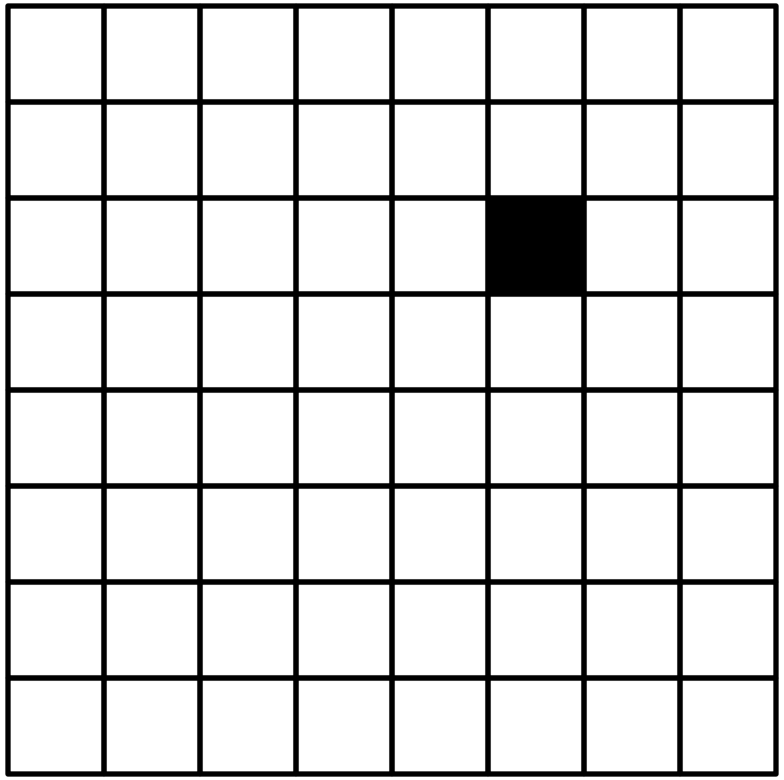
CS4102 Algorithms

Fall 2018

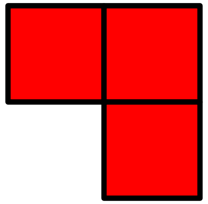
Warm up

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

Can you cover this?



With these?



Today’s Keywords

- Recursion
- Recurrences
- Asymptotic notation
- Divide and Conquer
- Trominos
- Merge Sort

CLRS Readings

- Chapters 3 & 4

Homeworks

- Hw0 due 11pm Wednesday, Sept 5
  - Submit 2 attachments (zip and pdf)
- Hw1 released Monday, Sept 3
  - Due 11pm Wednesday, Sept 12
  - Written (use Latex!)
  - Asymptotic notation
  - Recurrences
  - Divide and conquer

Attendance

- How many people are here today?
- Naïve algorithm
  1. Everyone stand
  2. Professor walks around counting people
  3. When counted, sit down
- Run time?
  - Class of n students
  - O(n)
- Other suggestions?

Better Attendance

1. Everyone Stand

2. Initialize your “count” to 1

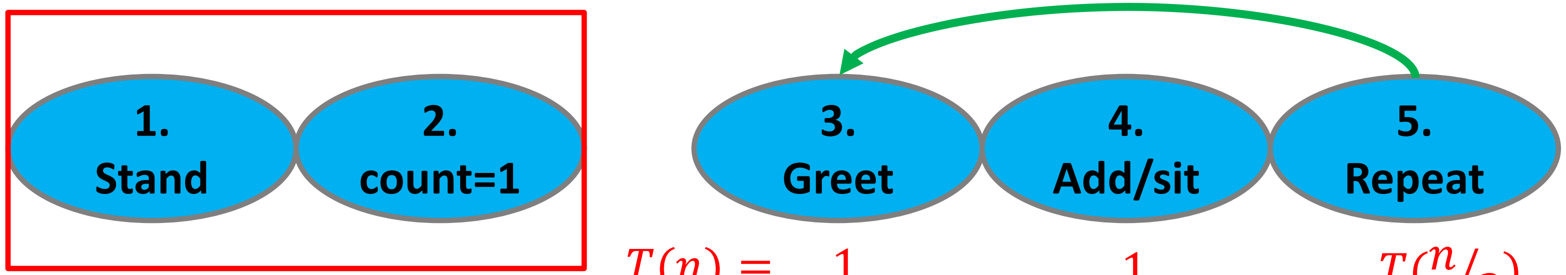
3. Greet a neighbor who is standing: share your name, full date of birth(pause if odd one out)

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 3

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

Attendance Algorithm Analysis



Constant Initialization

$T(n) = \quad 1 \qquad 1 \qquad T(^n/_2)$

$$T(n) = 1 + 1 + T(^n/_2)$$

$$T(1) = 3$$

How can we “solve” this?

Base case?

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

Why?

Let's solve the recurrence!

Special case: $n = 2^k$

$$T(1) = 3$$

$$\left.\begin{array}{l} T(n) = 2 + \cancel{T\left(n/2\right)} \\ \qquad\qquad 2 + \cancel{T\left(n/4\right)} \\ \qquad\qquad\quad 2 + \cancel{T\left(n/8\right)} \\ \qquad\qquad\qquad\quad \ddots \\ \qquad\qquad\qquad\qquad 3 \end{array}\right\} k$$

$$T(n) = 3 + \sum_{i=0}^{\log_2 n} 2 = 2\log_2 n + 3$$

What if $n \neq 2^k$?

- More people in the room → more time

  - $\forall\ 0 < n < m, T(n) < T(m)$

  - $T(n) \leq T\left(2^{\lceil \log_2 n \rceil}\right) = 2\,\lceil \log_2 n \rceil + 3 \quad = O(\log n)$

These are unimportant.
Why?

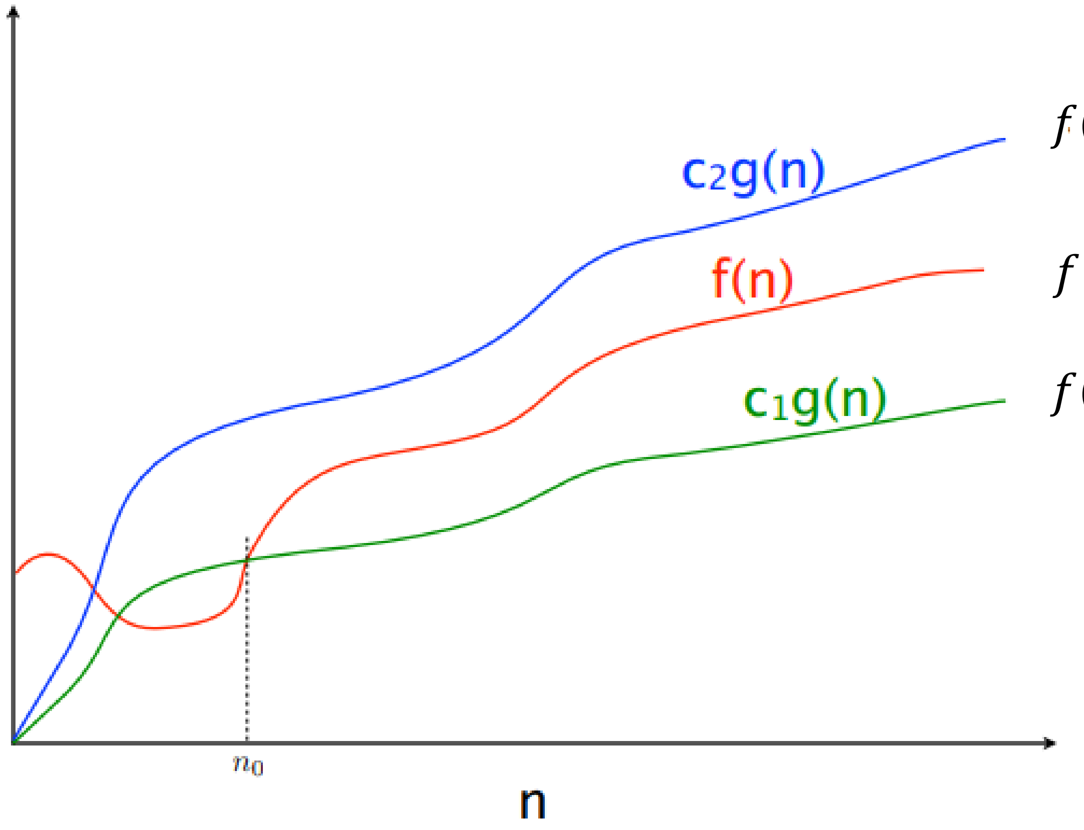
Asymptotic Notation*

- $O(g(n))$
  - At most within constant of g for large n
  - {functions $f \mid \exists$ constants $c > 0, n_0 > 0$ s.t. $\forall n > n_0, f(n) \leq c \cdot g(n)$}

- $\Omega(g(n))$
  - At least within constant of g for large n
  - {functions $f \mid \exists$ constants $c > 0, n_0 > 0$ s.t. $\forall n > n_0, f(n) \geq c \cdot g(n)$}

- $\Theta\big(g(n)\big)$
  - "Tightly" within constant of g for large n
  - $\Omega\big(g(n)\big) \cap O(g(n))$

*CLRS Chapter 3



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

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

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

Asymptotic Notation Example

- To Show: $n \log n \in O(n^2)$
  - Find $c, n_0 > 0$ s.t. $\forall n > n_0, n \log n \leq c \cdot n^2$
  - Let $c = 1, n_0 = 1$
  - $(1)\log(1) = 0, 1 \cdot 1^2 = 1$
  - $\forall n \geq 1, \log(n) < n \Rightarrow n \log n \leq n^2$

Asymptotic Notation

- $o(g(n))$
  - Below *any* constant of g for large n
  - $\{$functions $f | \forall$ constants $c > 0, \exists n_0$ s.t. $\forall n > n_0, f(n) < c \cdot g(n)\}$

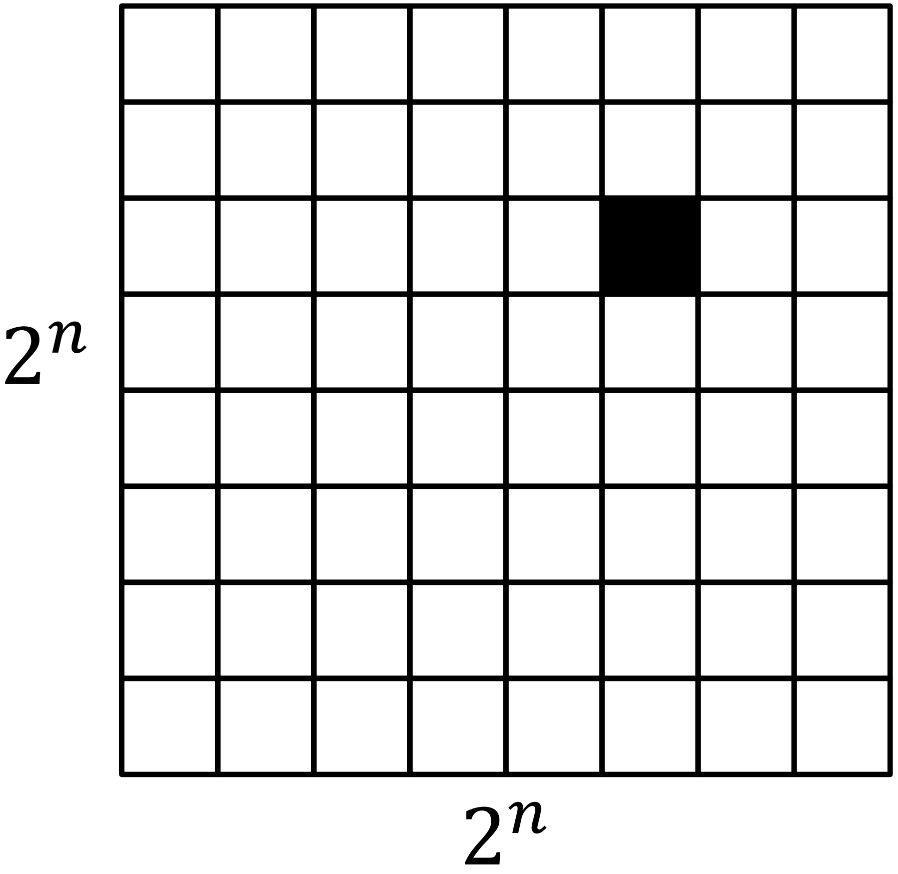
- $\omega(g(n))$
  - Above *any* constant of g for large n
  - $\{$functions $f | \forall$ constants $c > 0, \exists n_0$ s.t. $\forall n > n_0, f(n) > c \cdot g(n)\}$

- $\theta\big(g(n)\big)$?
  - $o(g(n)) \cap \omega(g(n)) = \emptyset$

Asymptotic Notation Example

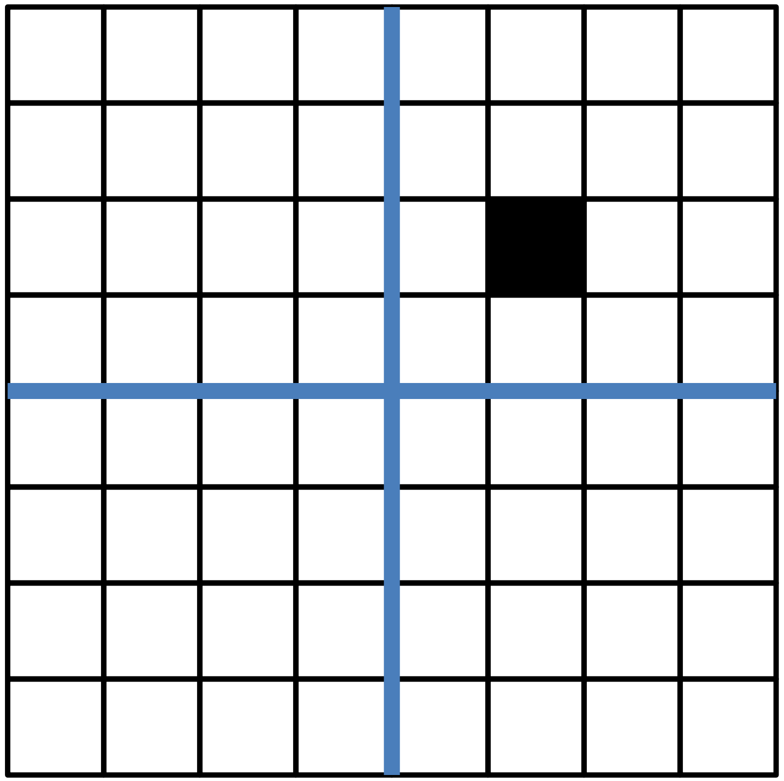
- $o(g(n)) = \{\text{functions } f \mid \forall \text{ constants } c, \exists n_0 \text{ s.t. } \forall n > n_0, f(n) < c \cdot g(n)\}$
- To Show: $n \log n \in o(n^2)$
  - given any c find a $n_0 > 0$ s.t. $\forall n > n_0, n \log n < c \cdot n^2$
  - Find a value of n in terms of c: $n \log n < c \cdot n^2$
  - $n \log n < c \cdot n^2$
  - $\log n < c \cdot n$
  - For a given c, select any value of n such that $\frac{\log n}{n} < c$

Trominoes Puzzle Solution



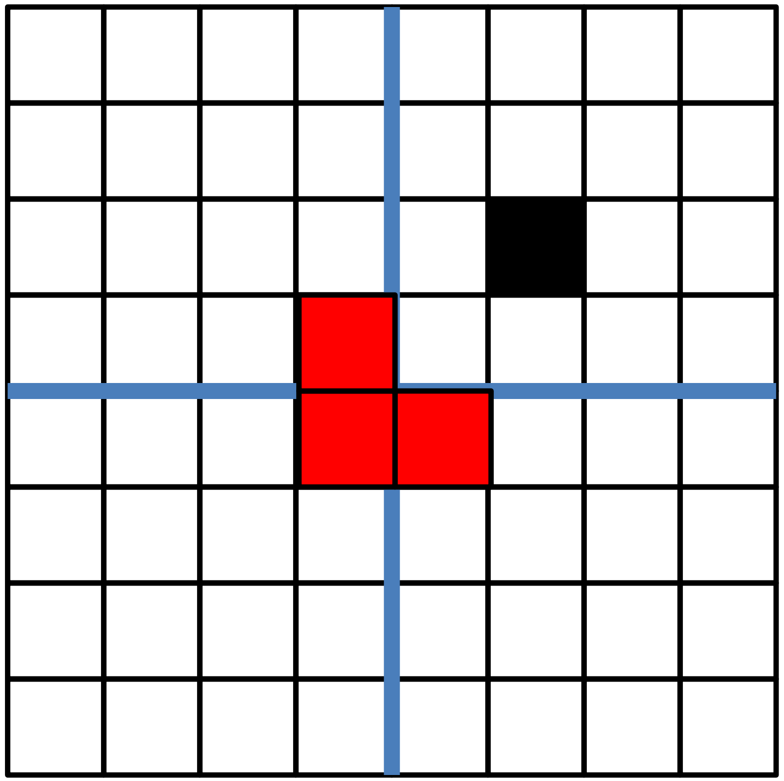
What about larger boards?

Trominoes Puzzle Solution



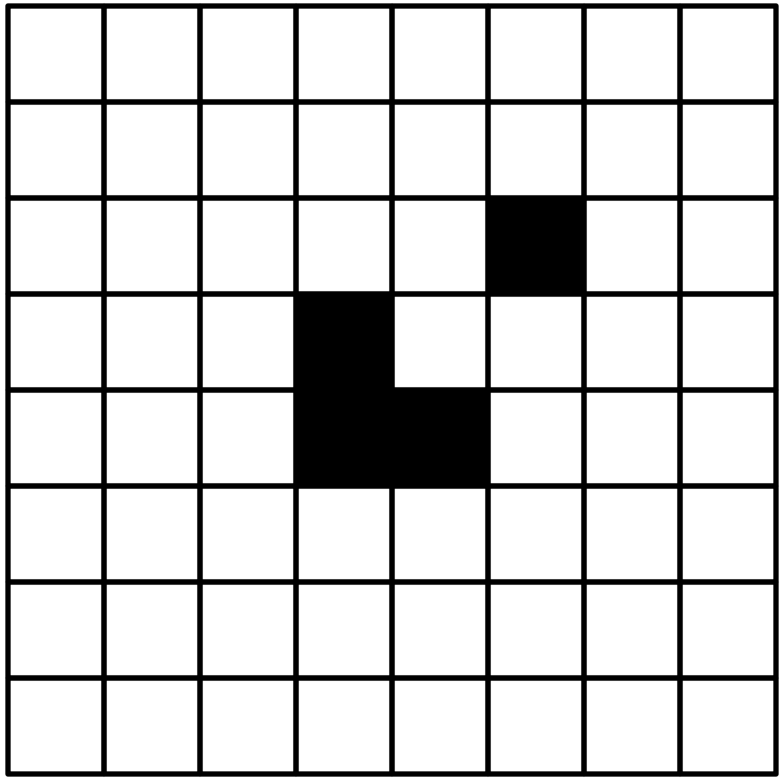
Divide the board into quadrants

Trominoes Puzzle Solution



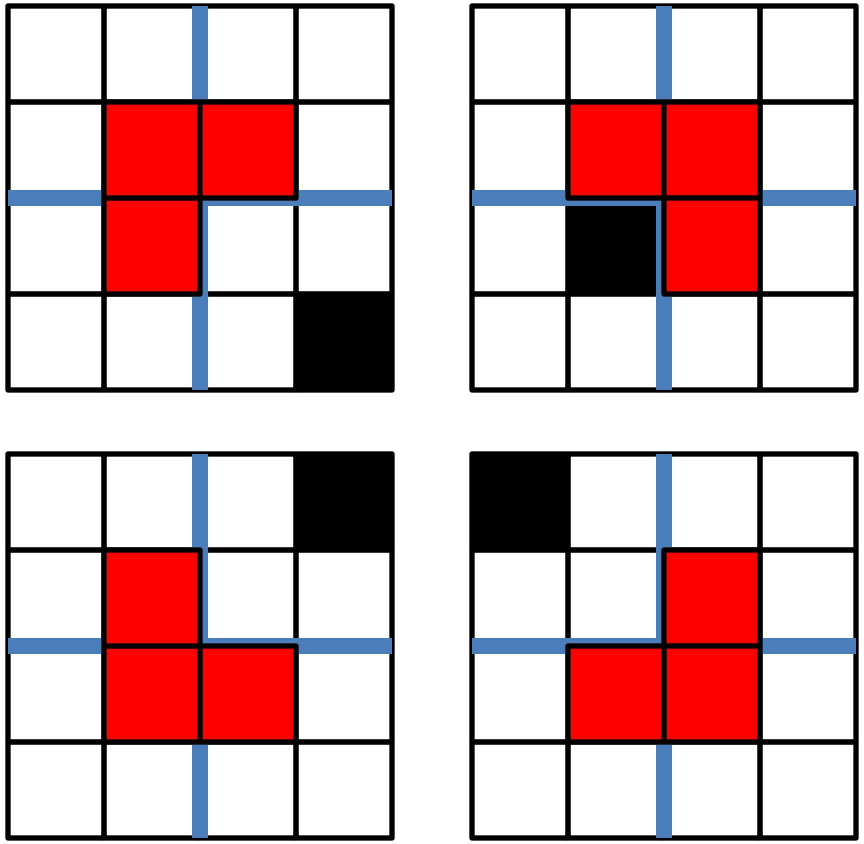
Place a tromino to occupy the three quadrants without the missing piece

Trominoes Puzzle Solution



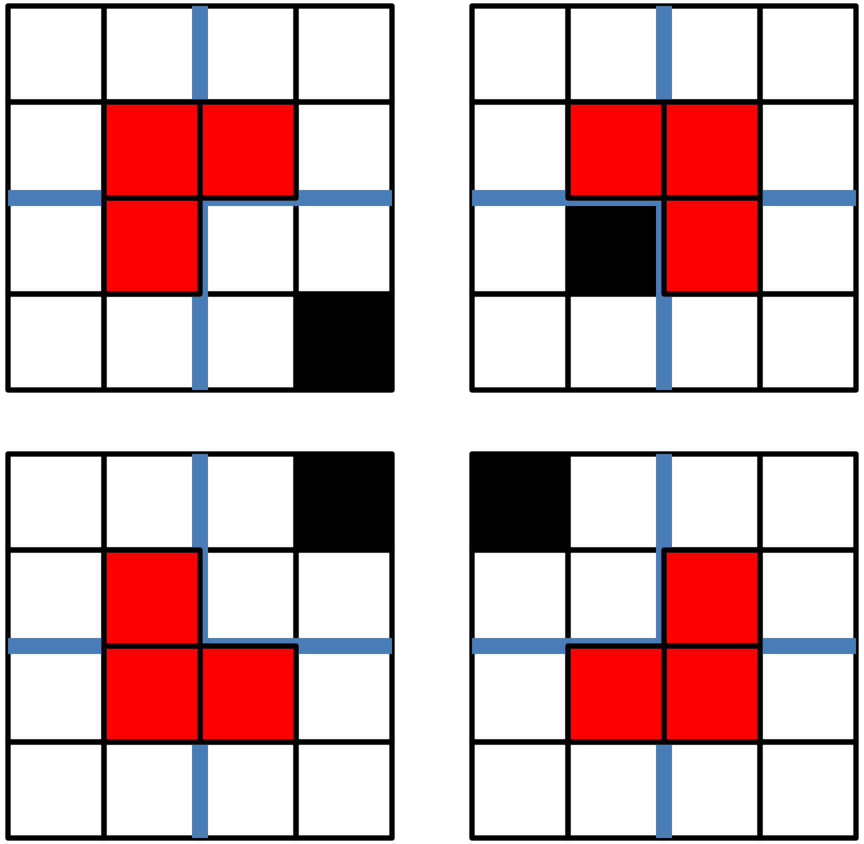
Each quadrant is now a smaller subproblem

Trominoes Puzzle Solution



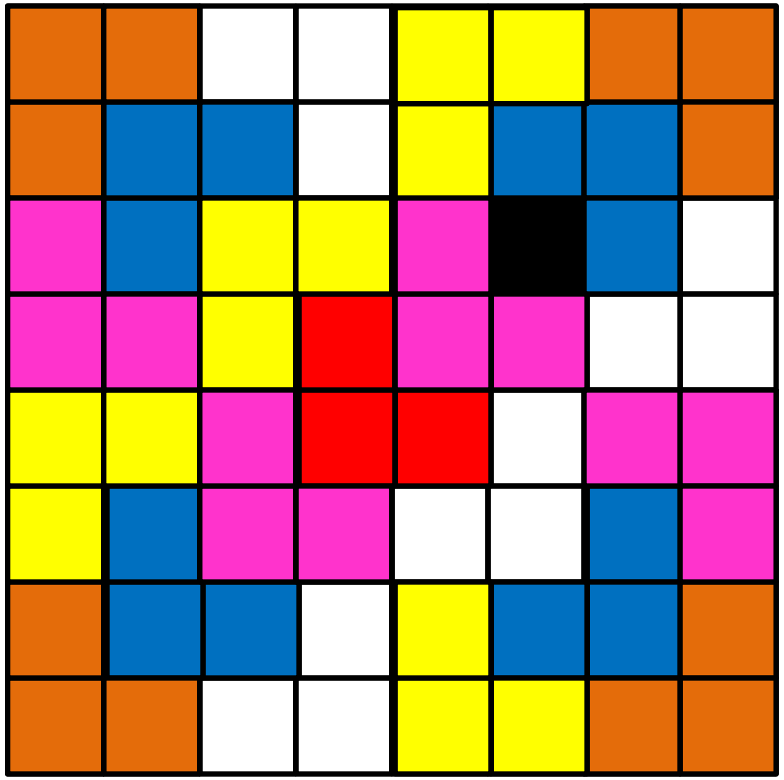
Solve **Recursively**

Divide and Conquer



Our first algorithmic technique!

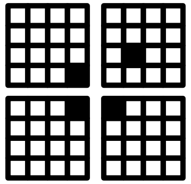
Trominoes Puzzle Solution



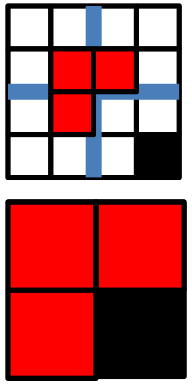
Divide and Conquer*

When is this a good strategy?

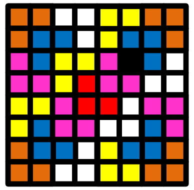
- **Divide**:



  – 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 together solutions to subproblems

*CLRS Chapter 4

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 small problems, size s
- **Combine**: $\mathrm{C}(n)$ time
- **Recurrence:**
  - $T(n) = D(n) + \sum T(s) + C(n)$

Recurrence Solving Techniques

Tree

Guess/Check

“Cookbook”

Substitution

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)

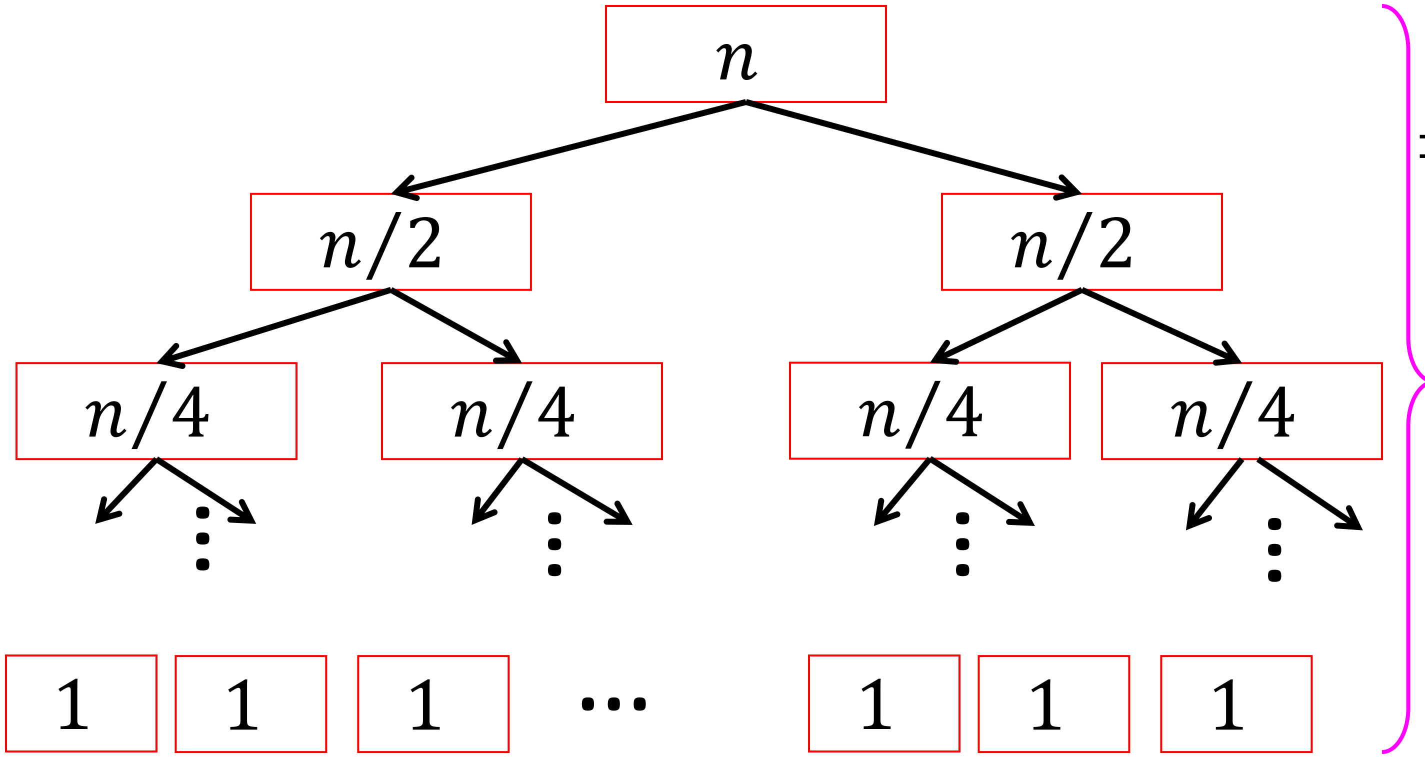
Analyzing Merge Sort

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 problems, size $\frac{n}{2}$
- **Combine**: n comparisons
- **Recurrence:**
  - $T(n) = 2T(\frac{n}{2}) + n$

Tree method

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



$\Rightarrow$ n total / level

$\log_2 n$ levels of recursion

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