# CSE 421 Winter 2025
# Lecture 9: Divide and Conquer

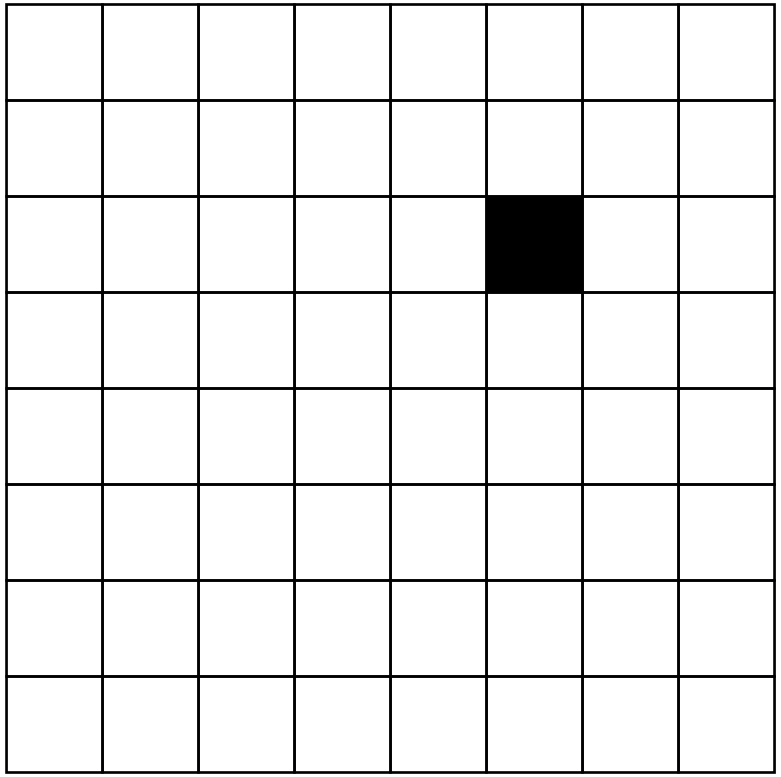
Nathan Brunelle

http://www.cs.uw.edu/421

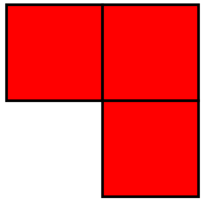
# Trominos Tiling

- Given an 8x8 grid with 1 cell missing, can we exactly cover it with “trominoes”?

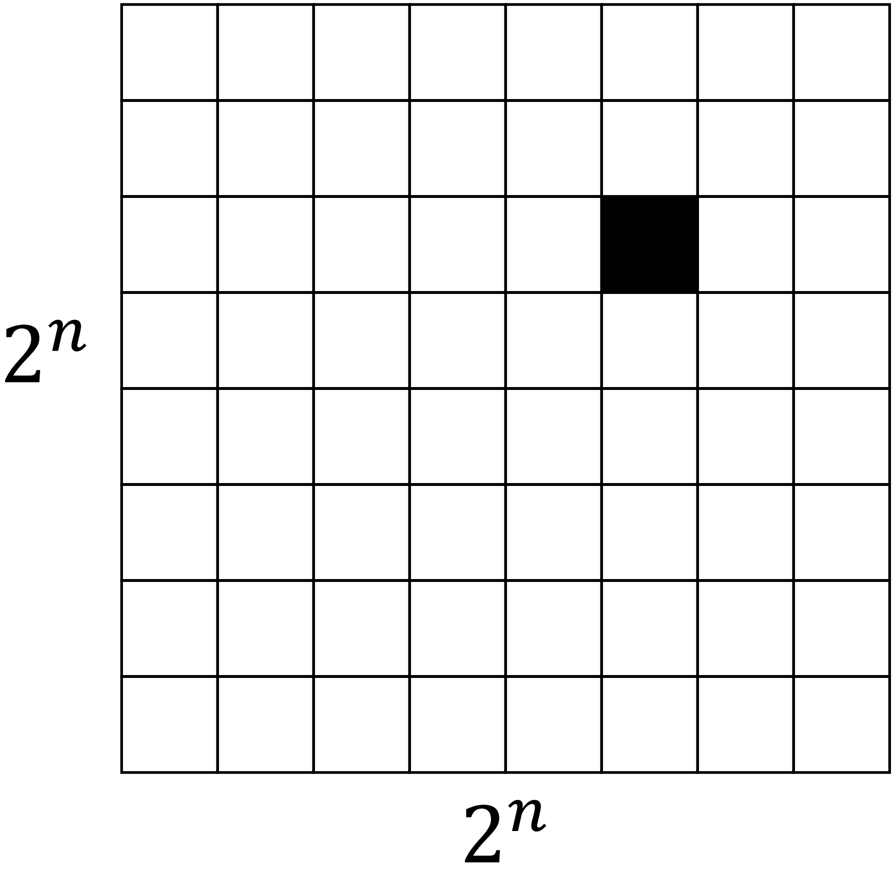
Can you cover this?



With these?

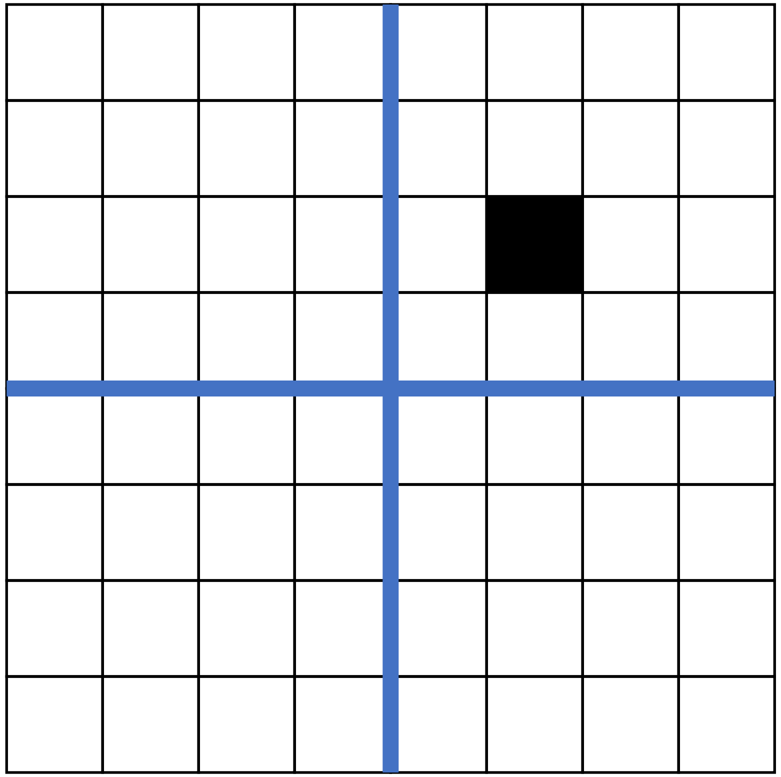


# Trominoes Puzzle Solution



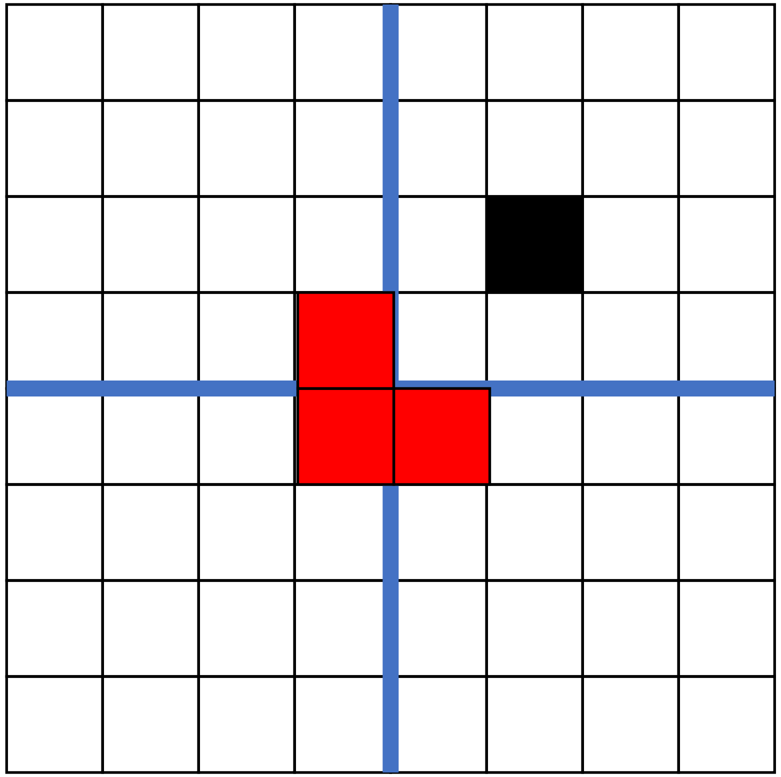

What about larger boards?

# Trominoes Puzzle Solution



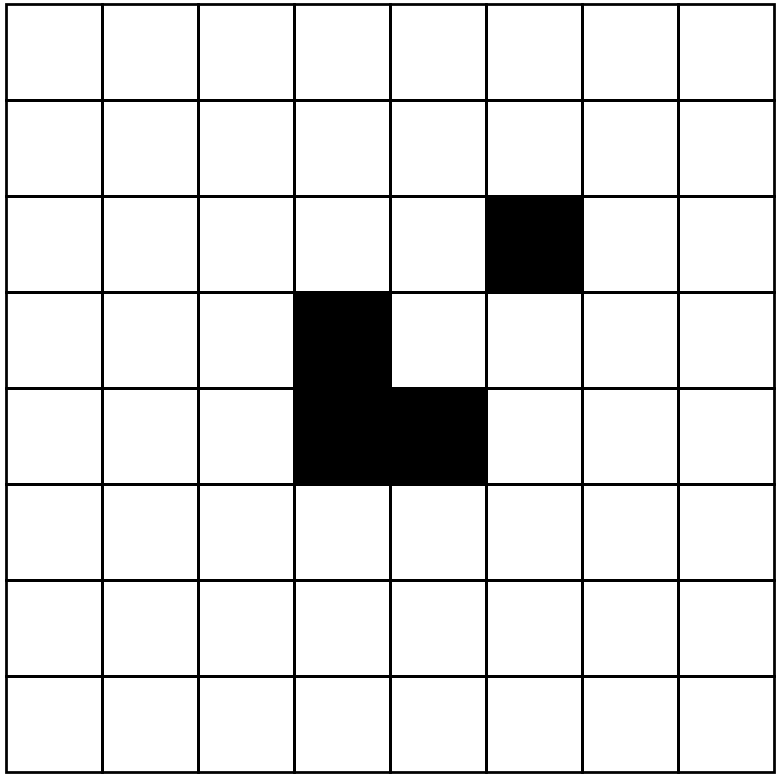
Divide the board into quadrants

# Trominoes Puzzle Solution



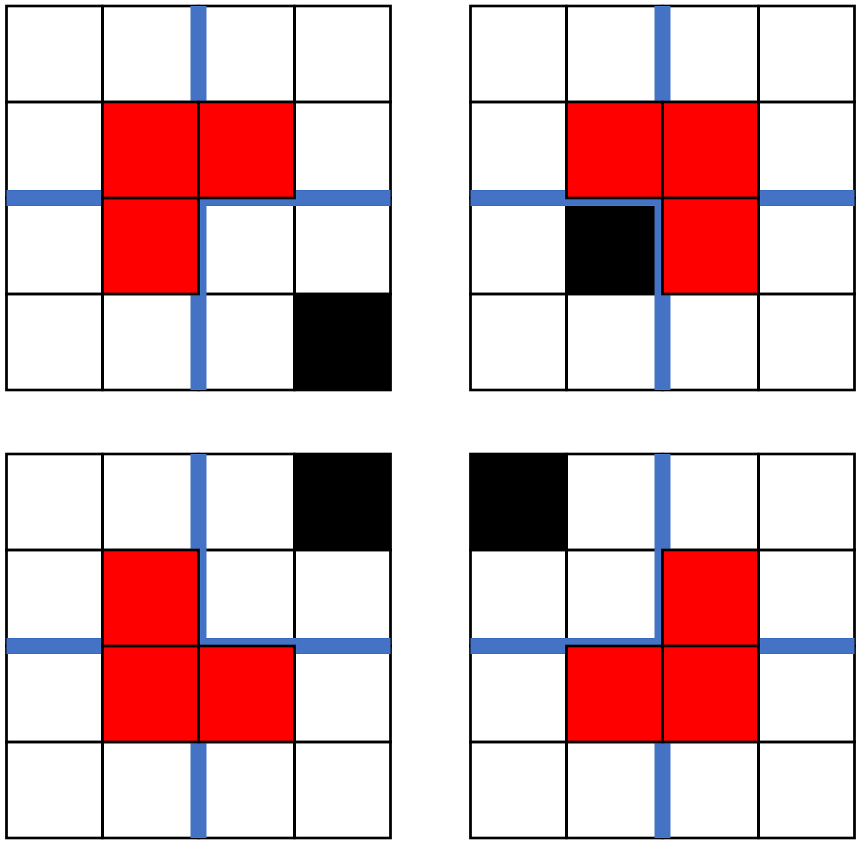
Place a tromino to occupy the three quadrants without the missing piece

# Trominoes Puzzle Solution



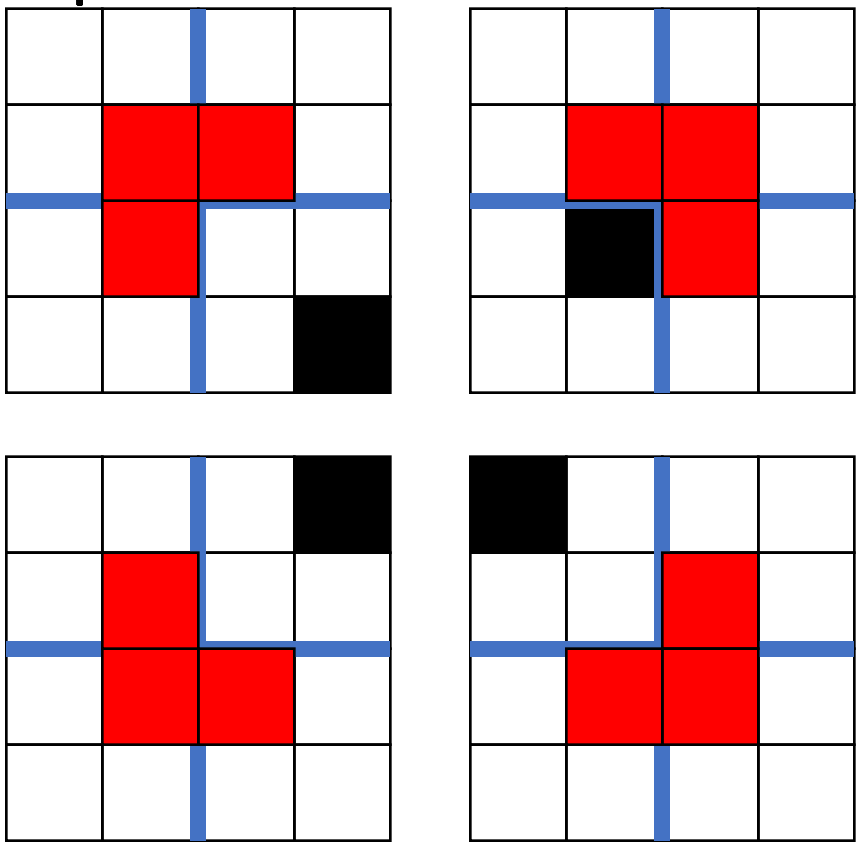
Each quadrant is now a smaller subproblem

# Trominoes Puzzle Solution

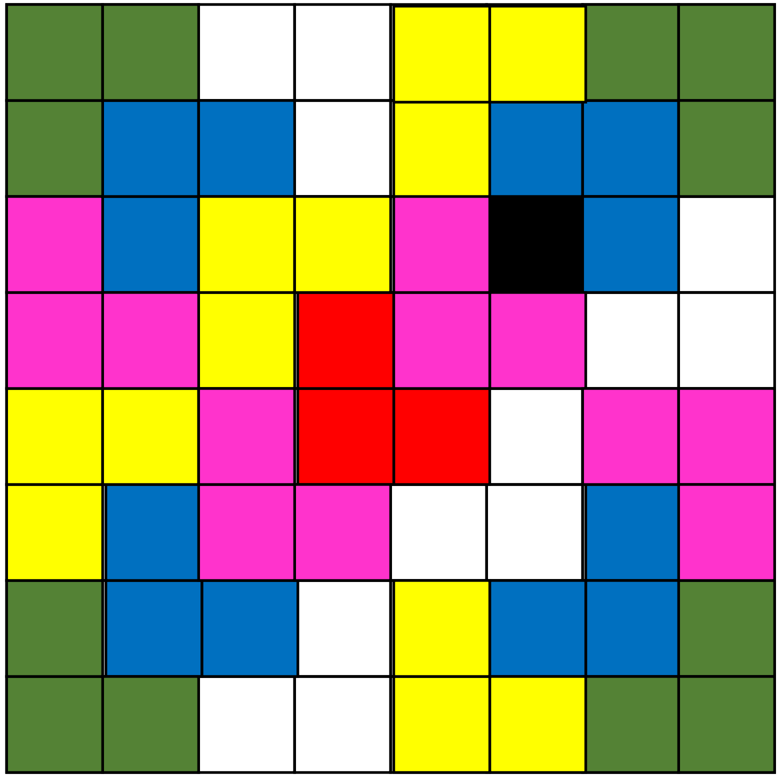


Solve **Recursively**

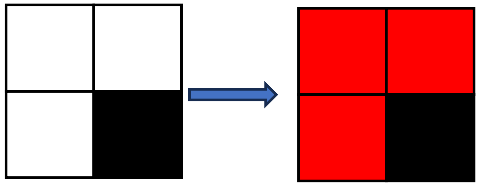
# Divide and Conquer



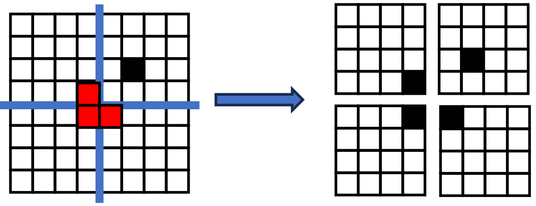
# Trominoes Puzzle Solution



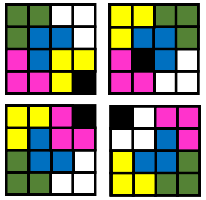
# Divide and Conquer (Trominoes)



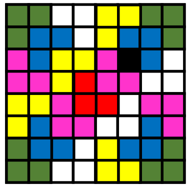
- **Base Case**:
  - For a $2 \times 2$ board, the empty cells will be exactly a tromino



- **Divide**:
  - Break of the board into quadrants of size $2^{n-1} \times 2^{n-1}$ each
  - Put a tromino at the intersection such that all quadrants have one occupied cell



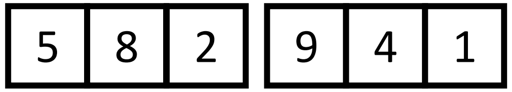
- **Conquer**:
  - Cover each quadrant



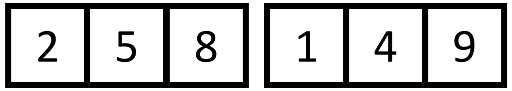
- **Combine**:
  - Reconnect quadrants

# Divide and Conquer (Merge Sort)

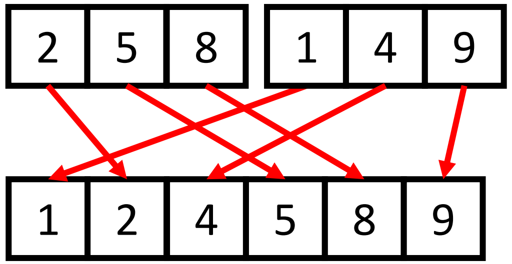

- **Base Case**:
  - If the list is of length 1 or 0, it’s already sorted, so just return it
  - (Alternative: when length is $\leq 15$, use insertion sort)




- **Divide**:
  - Split the list into two “sublists” of (roughly) equal length




- **Conquer**:
  - Sort both lists recursively




- **Combine**:
  - **Merge** sorted sublists into one sorted list

# Divide and Conquer (Running Time)

$T(c) = k$

$a = number\ of\ subproblems$

$\frac{n}{b} = size\ of\ each\ subproblem$

$f_d(n) =$ time to divide

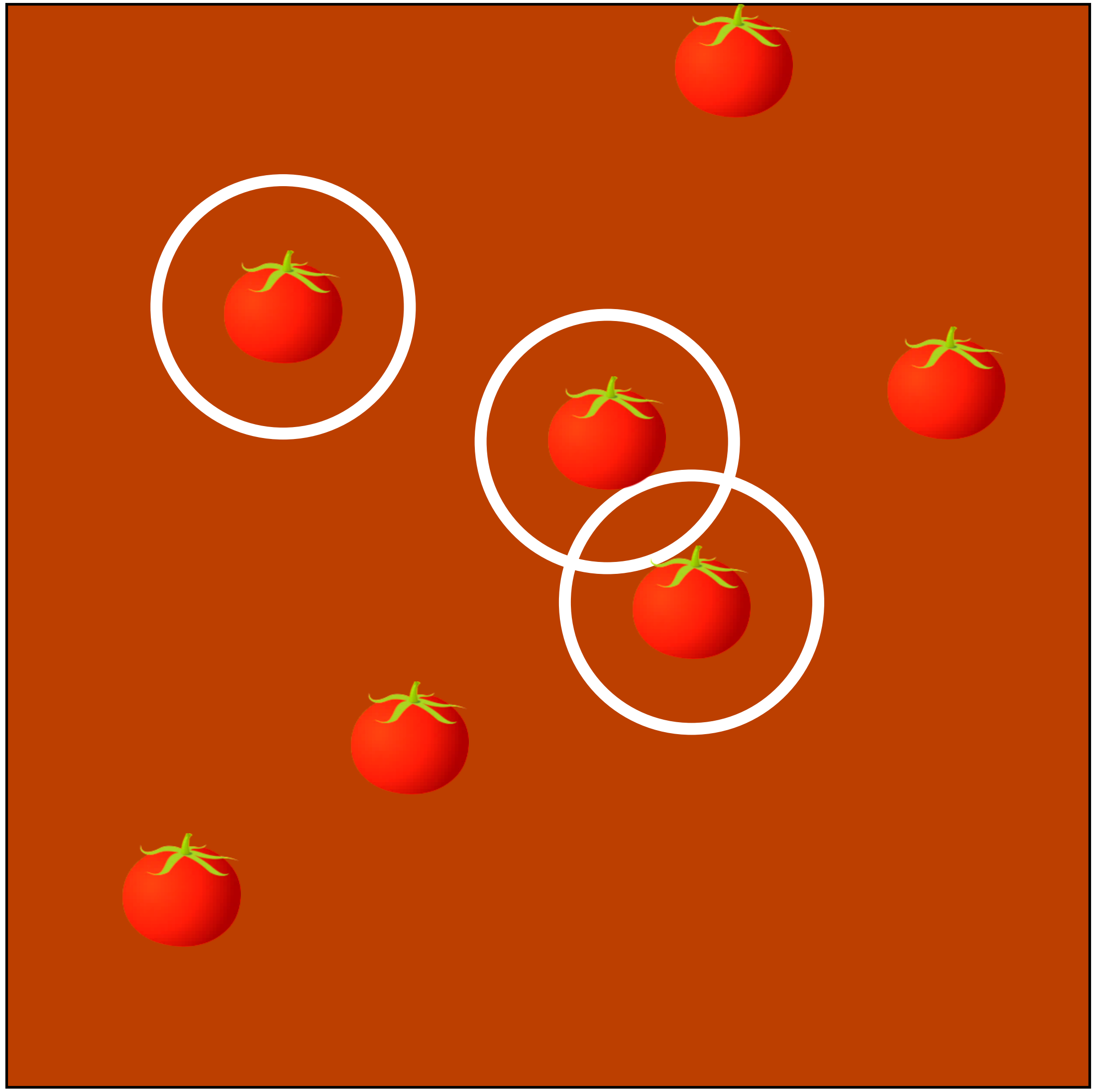
$a \cdot T\left(\frac{n}{b}\right)$

$f_c(n) =$time to combine

- **Base Case**:
  - When the problem size is small ($\leq c$), solve non-recursively
- **Divide**:
  - When problem size is large, identify 1 or more smaller versions of exactly the same problem
- **Conquer**:
  - Recursively solve each smaller subproblem
- **Combine**:
  - Use the subproblems' solutions to solve to the original

**Overall:** $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ **where** $f(n) = f_d(n) + f_c(n)$

# Closest Pair of Tomatoes



# Closest Pair of Points

**Given:**

- A sequence of $n$ points $p_1, \ldots, p_n$ with real coordinates in 2 dimensions ($\mathbb{R}^2$)

**Find:**

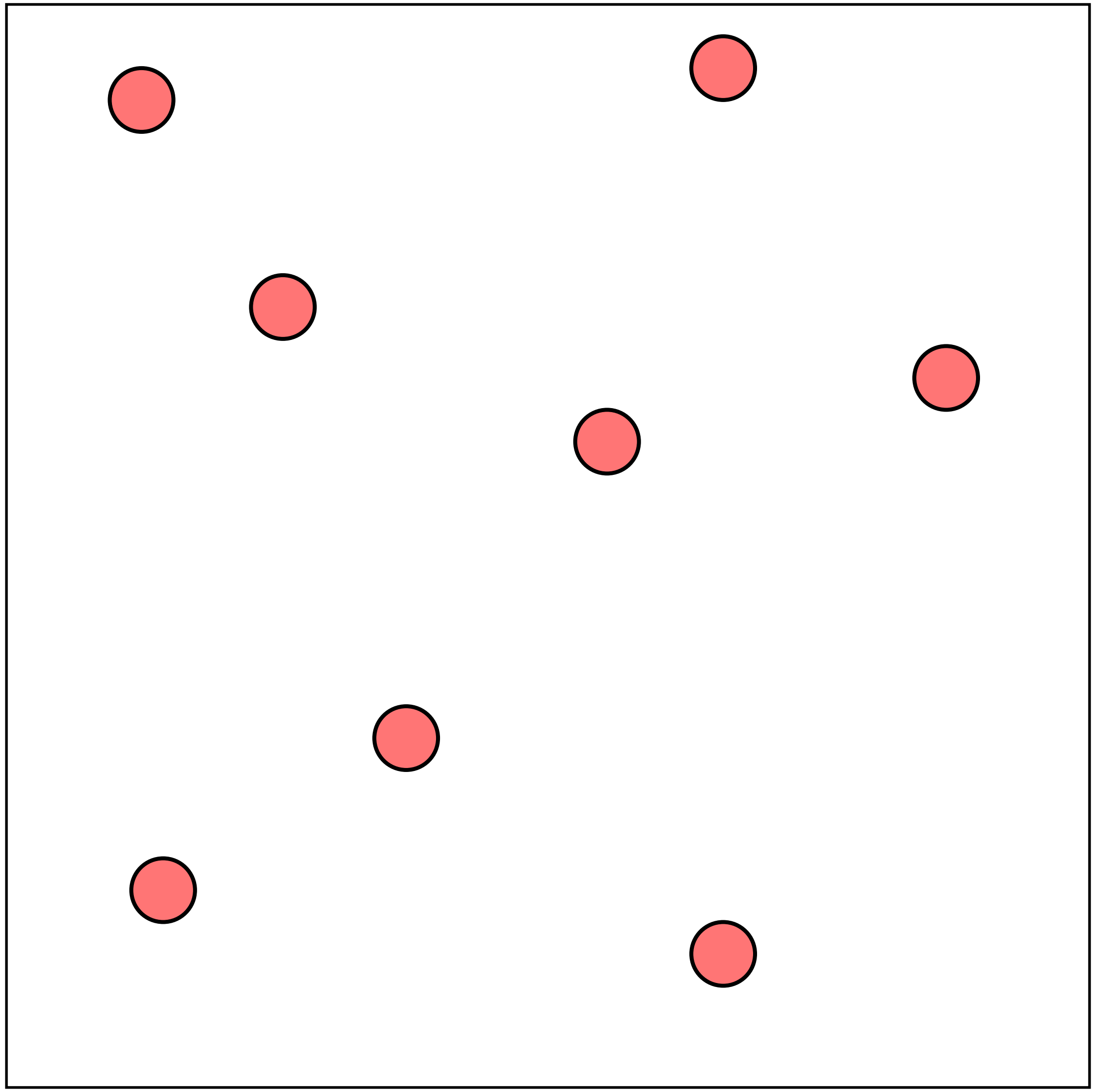
- A pair of points $p_i, p_j$ s.t. the Euclidean distance $d(p_i, p_j)$ is minimized

How about a $\Theta(n^2)$ algorithm?

- Try all possible pairs, keeping the smallest

Our goal:

- Use D&C to create a $\Theta(n \log n)$ algorithm

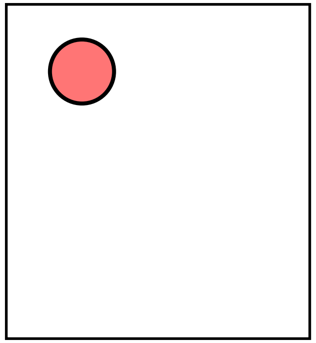


# Closest Pair of Point D&C Idea

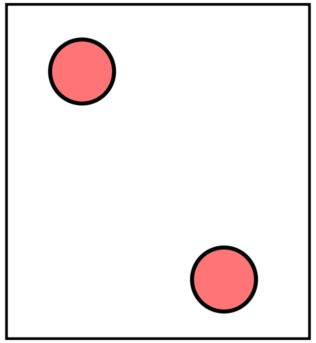
**To get $\Theta(n \log n)$, we will aim for $T(n) = 2T\left(\frac{n}{2}\right) + n$**

- **Base Case:**
  - If the number of points is small, do use a naïve solution
- **Divide:**
  - Otherwise partition the points into 2 subsets
  - Running time "budget" $O(n)$
- **Conquer:**
  - Find the closest pair of points in each subset
- **Combine:**
  - Use those closest pairs of points to find the closest overall
  - Running time "budget" $O(n)$

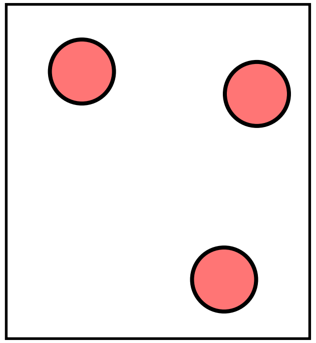
# Closest Pair: Base Cases



If $n = 1$
return $\infty$

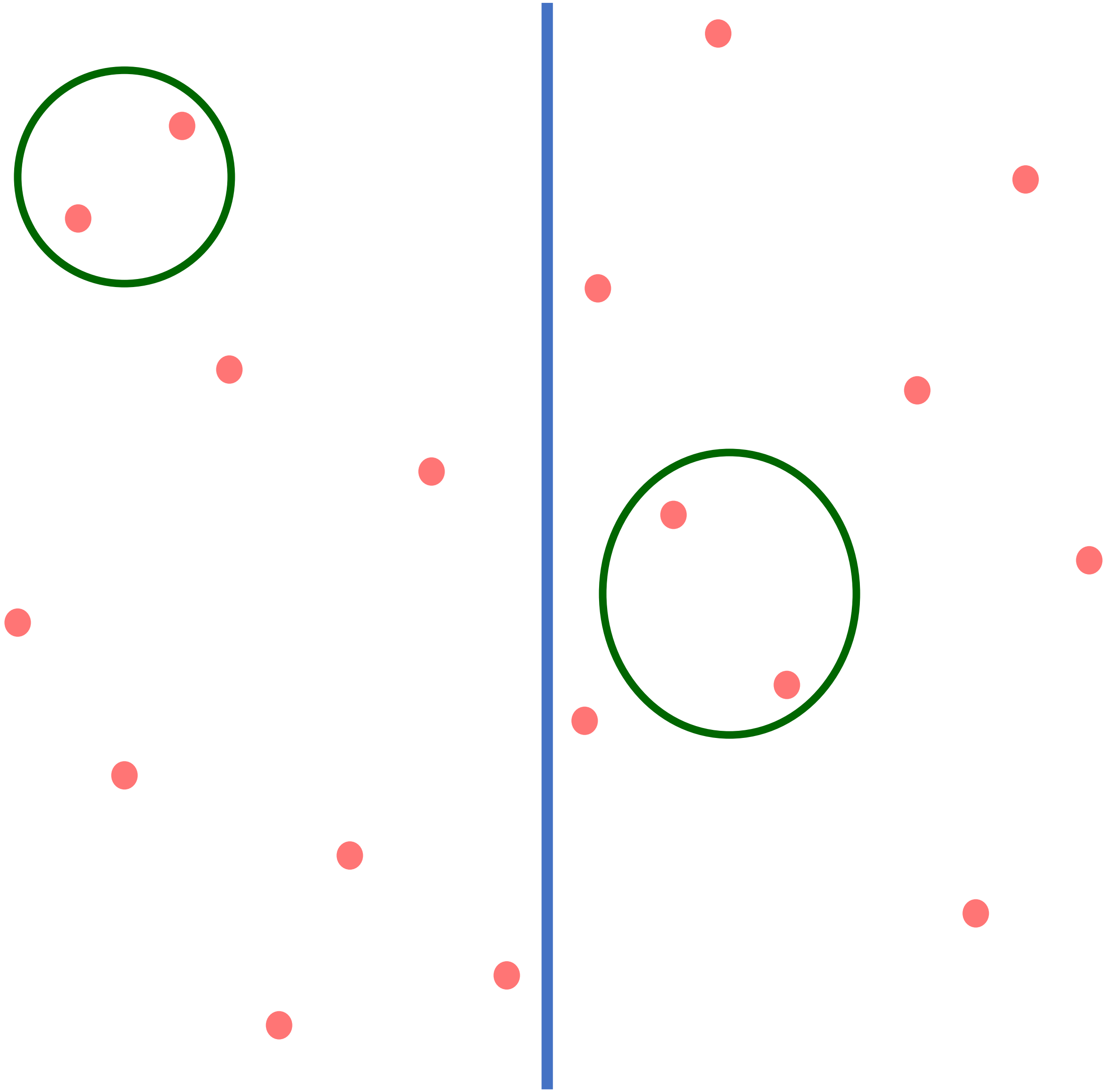


If $n = 2$
return the distance



If $n = 3$
check all 3 pairs
return the closest

# Closest Pair: First Idea



**Divide:**

- Split using **median** $x$-coordinate
- each subpart has size $n/2$.

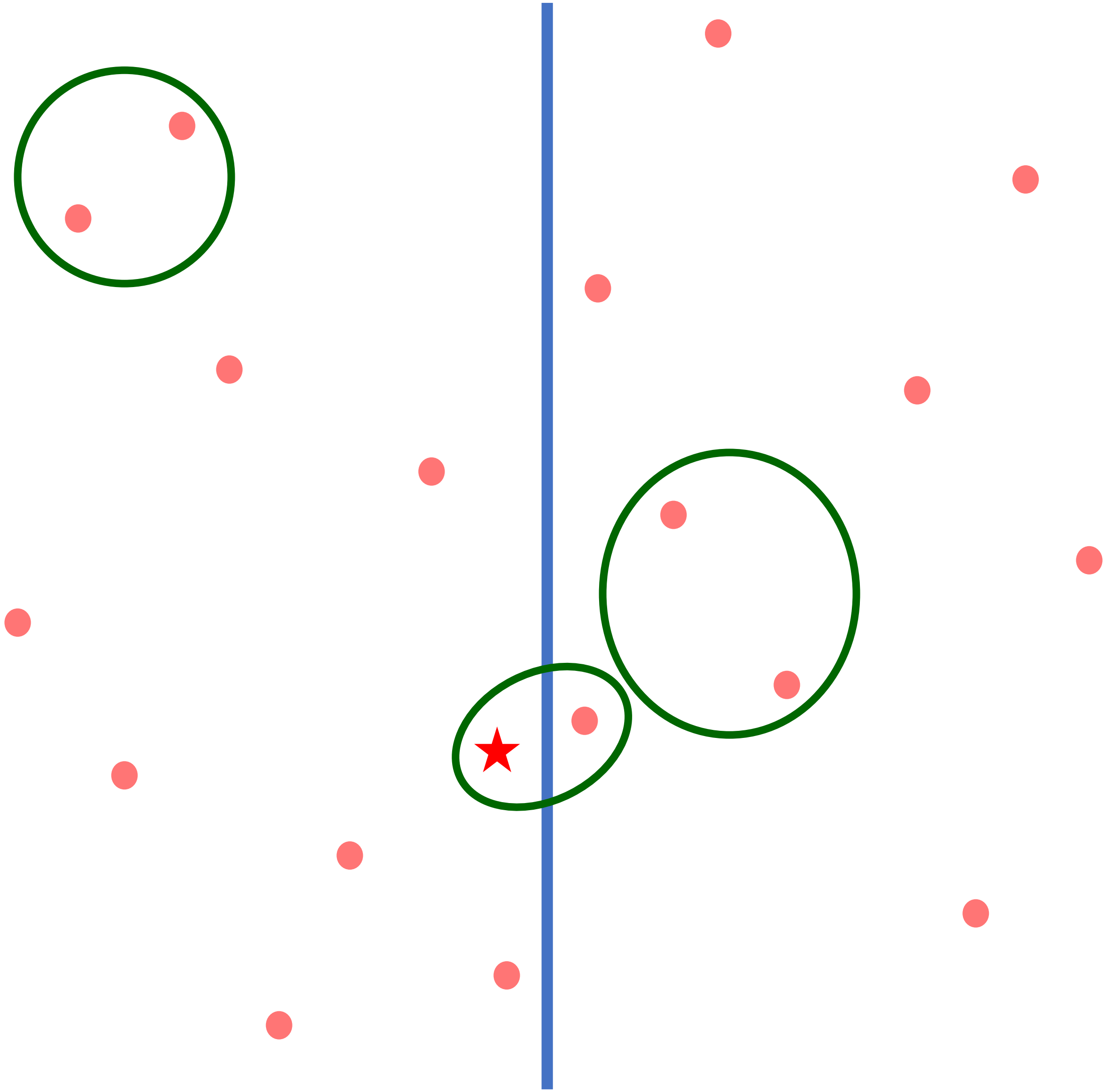
**Conquer:**

- Solve both size $n/2$ subproblems
- We now have the closest pair from the left and from the right

**Combine:**

- Return the closer of the left pair and the right pair

# Closest Pair: First Idea - Problem



**Divide:**

- Split using **median** $x$-coordinate
- each subpart has size $n/2$.

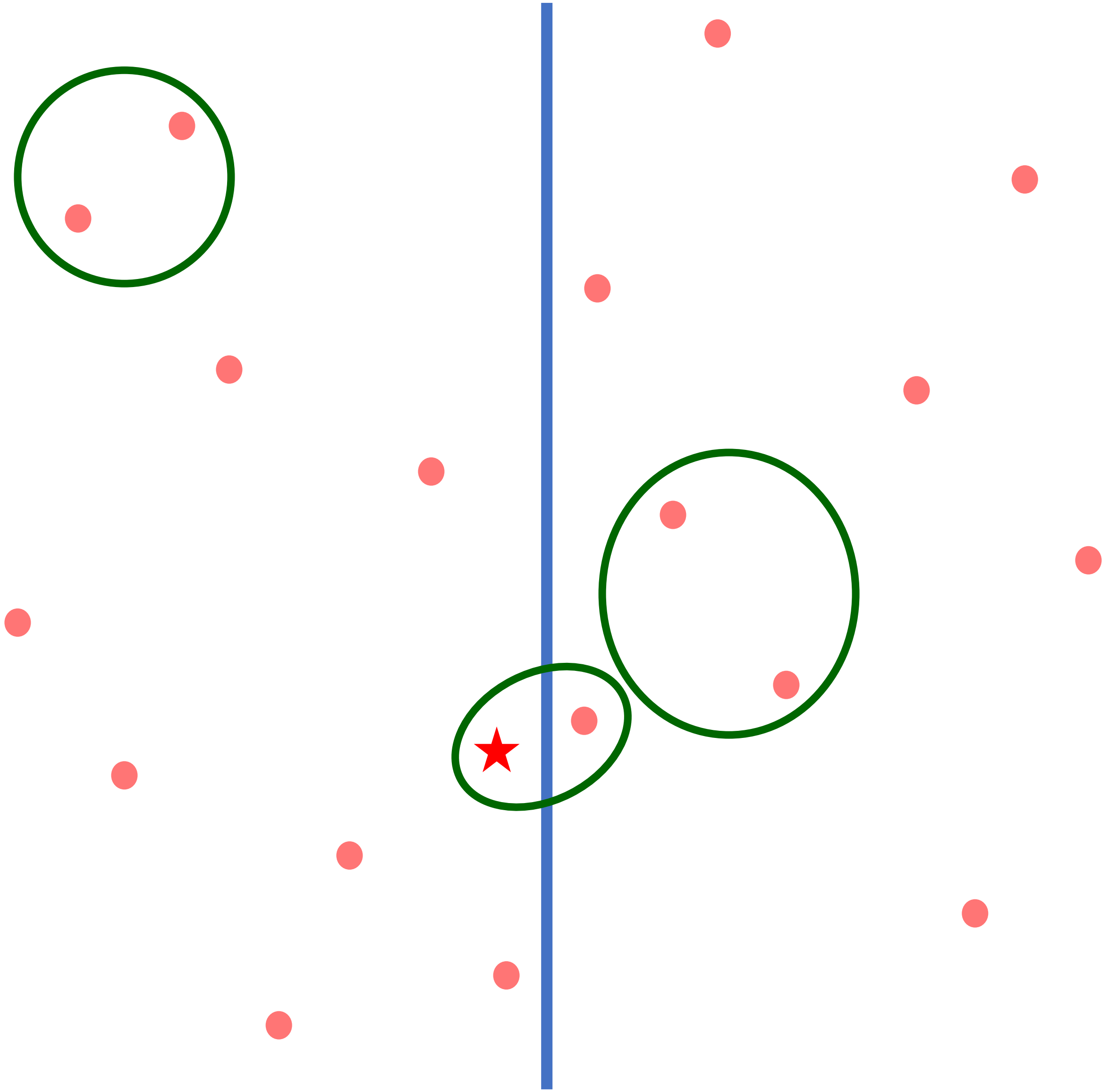
**Conquer:**

- Solve both size $n/2$ subproblems
- We now have the closest pair from the left and from the right

**Combine:**

- Find the closest pair crossing the middle
- Return the closest of the left, right, and crossing pairs

# Finding the Closest Crossing Pair – 1st Idea



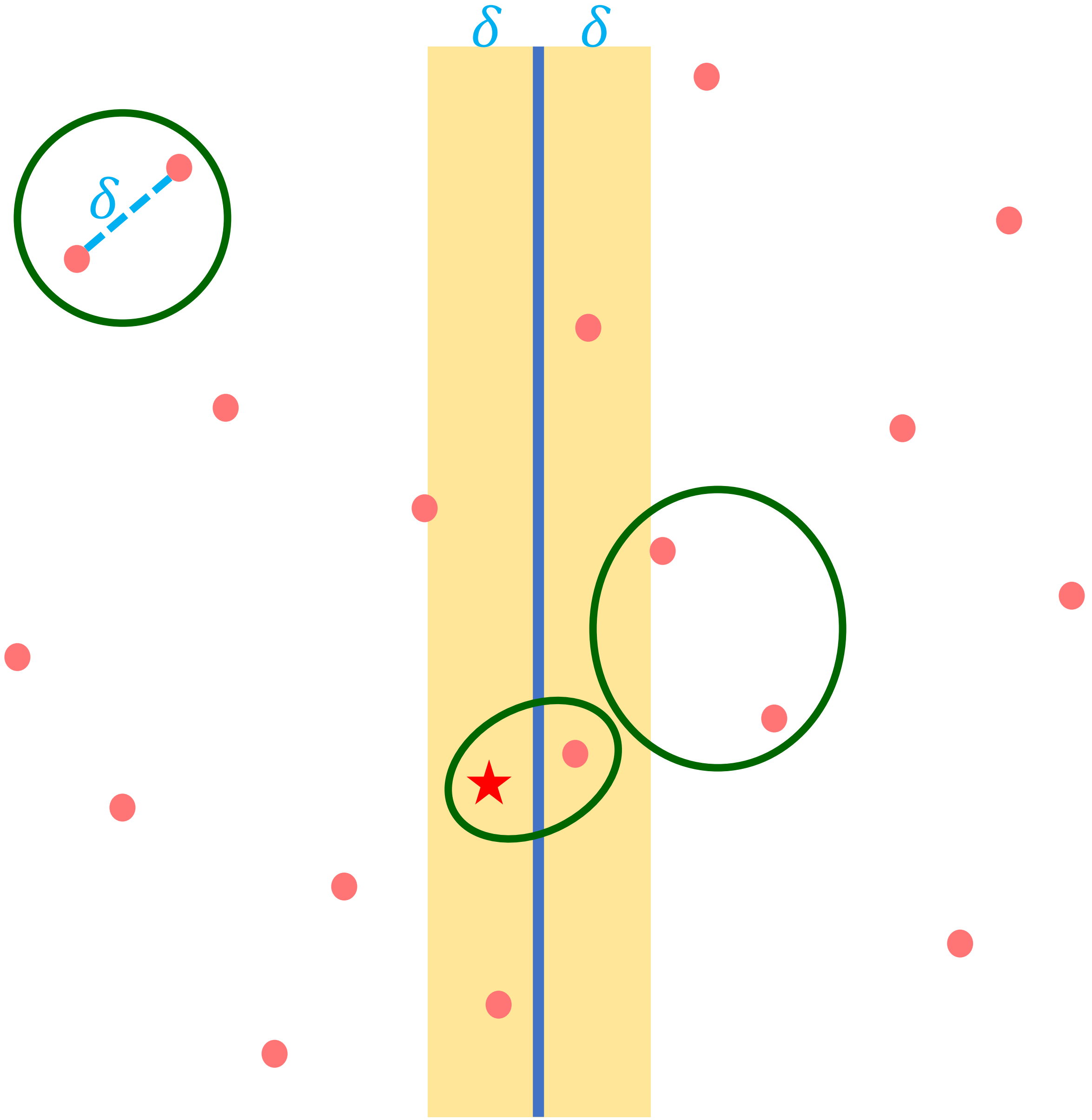
**Combine:**
- Find the closest pair crossing the middle
- Return the closest of the left, right, and crossing pairs

**Procedure:**
- For each point on the left, find its closest point on the right
- Save the closest seen as the crossing pair

**Problem?**

Running time is $\left(\frac{n}{2}\right)^2$

# Finding the Closest Crossing Pair – 2nd Idea

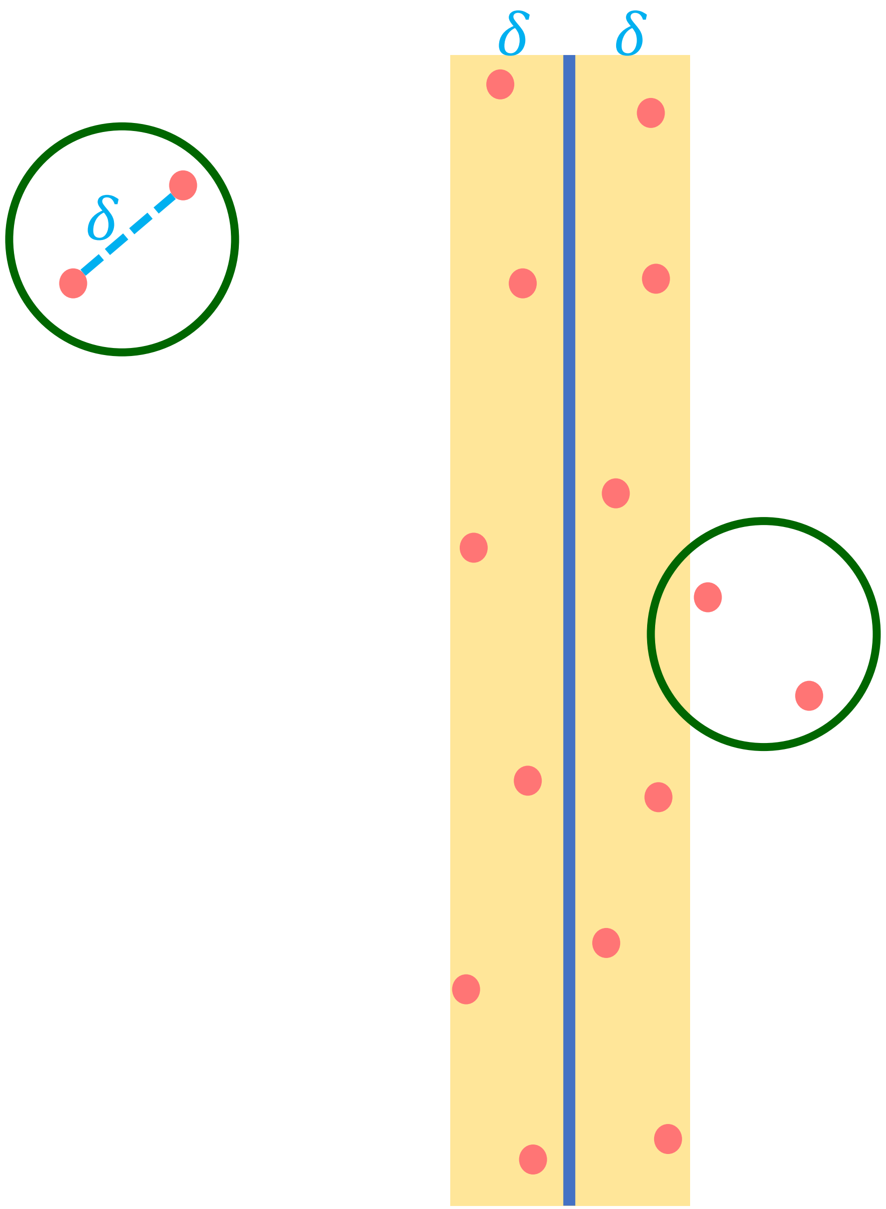



**Combine:**
- Find the closest pair crossing the middle
- Return the closest of the left, right, and crossing pairs

**Observation:**
- We only care about crossing pairs that might be closer than left and right
- Ignore points too far from the divide

**Procedure:**
- Let $\delta$ be the closest distance from left and right
- For each point on the left that's within $\delta$ of the divide, find its closest match from among points within $\delta$ on the right

# Problem with the 2nd Idea



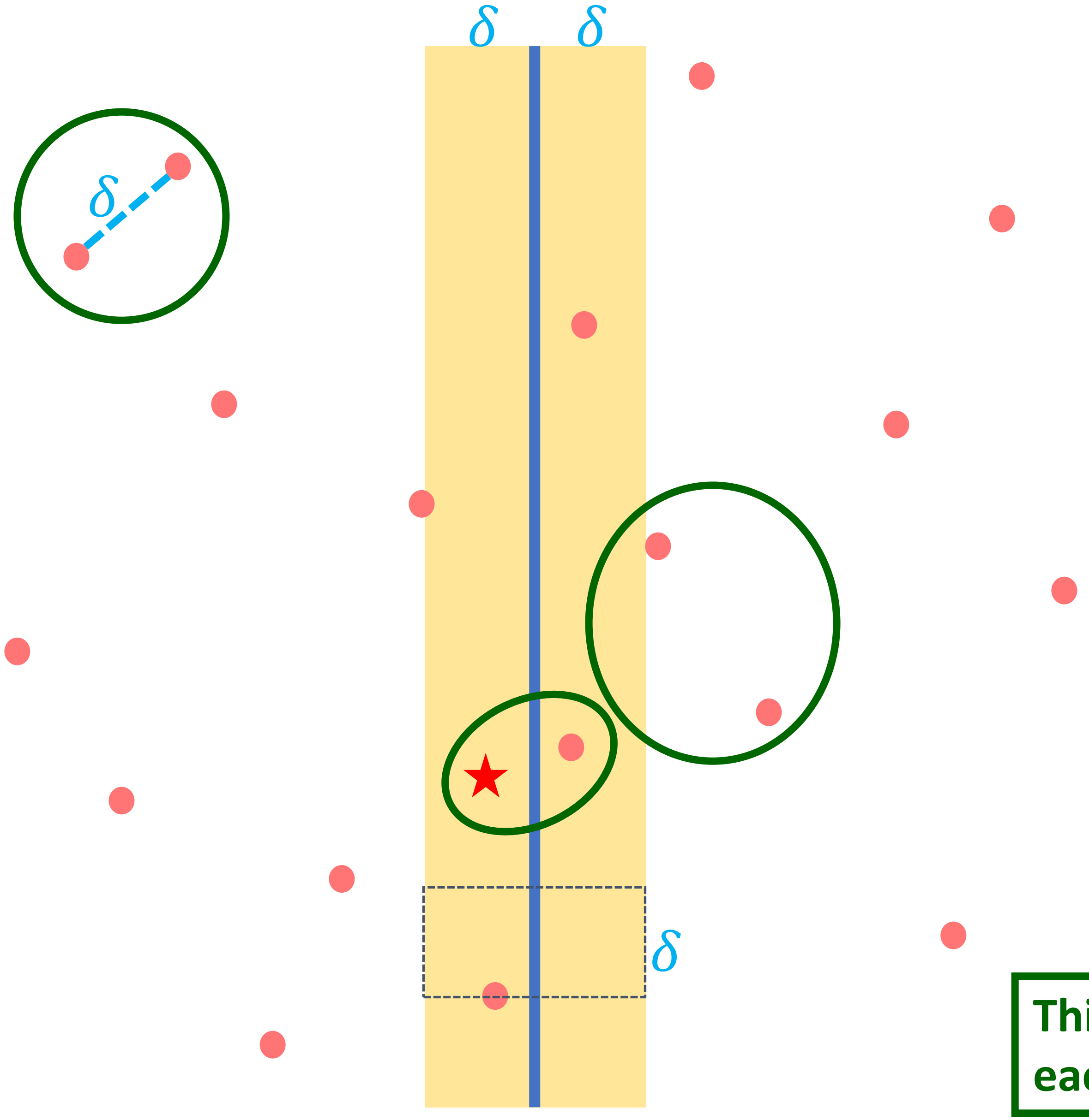

**Combine:**
- Find the closest pair crossing the middle
- Return the closest of the left, right, and crossing pairs

**Observation:**
- We only care about crossing pairs that might be closer than left and right
- Ignore points too far from the divide

**Problem:**
- We could still exceed our budget!

**Solution:**
- Re-apply the observation vertically!
- We only need to consider points within $\delta$ above the current point as well!

# Finding the Closest Crossing Pair – 3rd Idea




**Combine:**

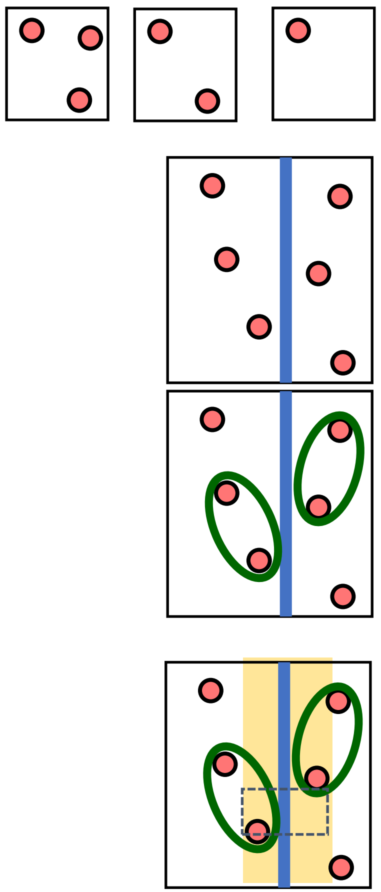
- Find the closest pair crossing the middle
- Return the closest of the left, right, and crossing pairs

**Procedure:**

- Let $\delta$ be the closest distance from left and right
- From bottom to top, for each point $p_l$ on the left that's within $\delta$ of the divide on the left:
  - compare it to each point on the right that is within $\delta$ of the divide and no more than $\delta$ above $p_l$

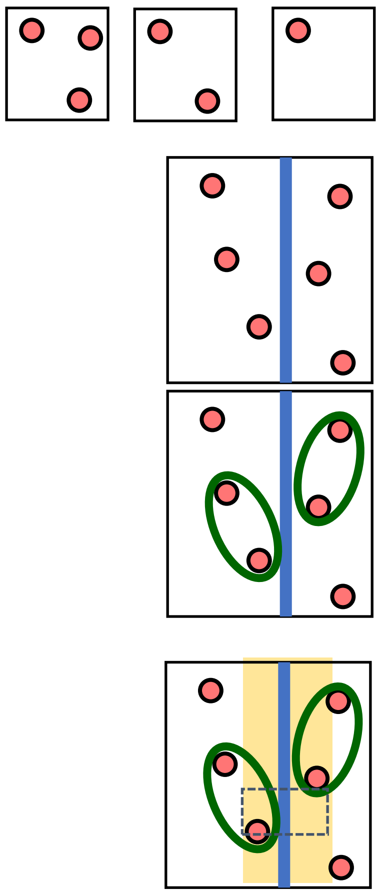
**This will only fit within our budget if we compare each $p_l$ to a constant number of other points**

# Divide and Conquer (Closest Pair of Points)



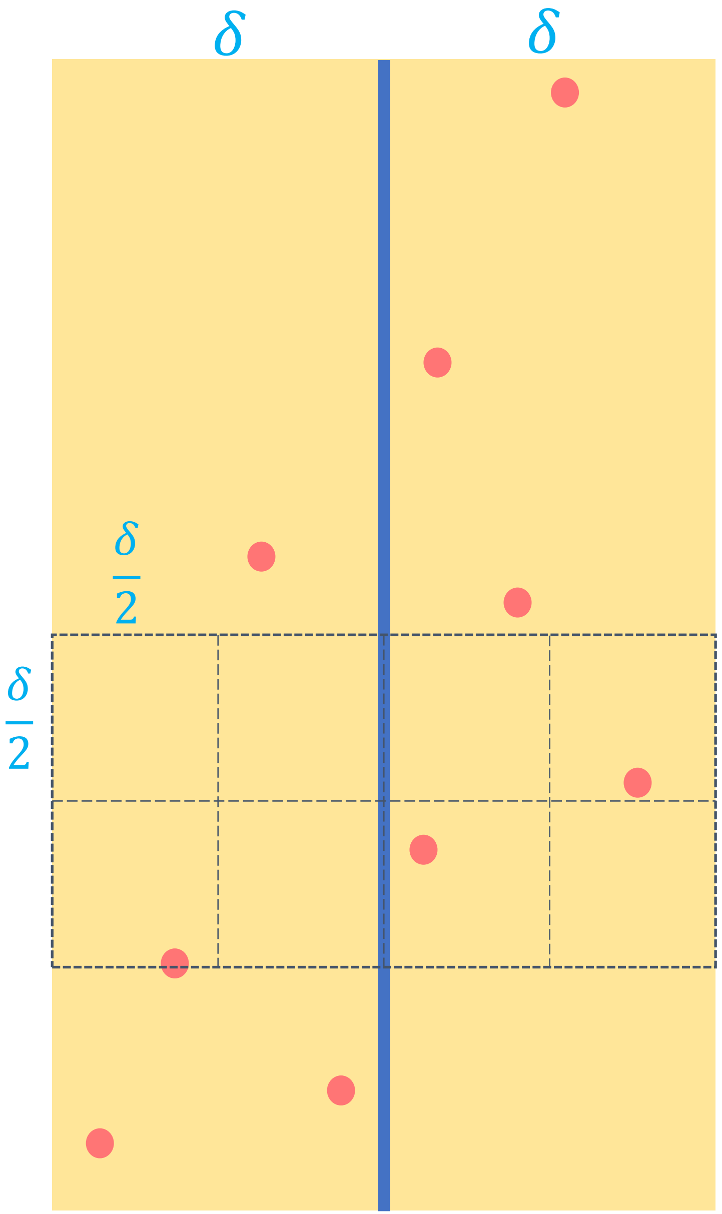
- **Preprocessing:**
  - Sort the points by $x$ coordinate (call this list $L_x$)
  - Make a copy of the points and sort by $y$ coordinate (call this list $L_y$)
- **Base Case:**
  - If there's 1 point then return ∞, If there's 2 or 3 points, solve naively
- **Divide:**
  - Find the median $x$ coordinate
  - Partition $L_x$ and $L_y$ into the points on the left vs. right of the median
- **Conquer:**
  - Recursively find the closest pair from among the left and right of the median
- **Combine:**
  - Let $\delta$ be the closest from the left and the right solutions
  - Filter $L_y$ to include only the points within $\delta$ of the median $x$
  - For each point $p$ still in $L_y$:
    - For each point within $\delta$ of $p$ vertically:
      - Compare $p$ with that point and save if the distance is less than $\delta$
  - Return minimum of the saved pair and the one used for $\delta$

# Surprisingly, This works!



- **Preprocessing:**
  - Sort the points by $x$ coordinate (call this list $L_x$)
  - Make a copy of the points and sort by $y$ coordinate (call this list $L_y$)
- **Base Case**:
  - If there's 1 point then return $\infty$, If there's 2 or 3 points, solve naively
- **Divide**:
  - Find the median $x$ coordinate
  - Partition $L_x$ and $L_y$ into the points on the left vs. right of the median
- **Conquer**:
  - Recursively find the closest pair from among the left and right of the median
- **Combine**:
  - Let $\delta$ be the closest from the left and the right solutions
  - Filter $L_y$ to include only the points within $\delta$ of the median $x$
  - For each point $p$ still in $L_y$:
    - **For the next 7 points vertically:**
      - Compare $p$ with that point and save if the distance is less than $\delta$
  - Return minimum of the saved pair and the one used for $\delta$

# Why is 7 enough?

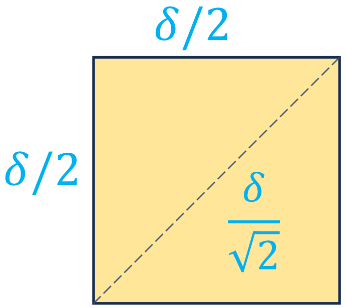



**Claim:**

- For any point $p$ in the "strip", the 8th point above it is guaranteed to be more than $\delta$ away.

**Proof:**

- Consider a grid of $\frac{\delta}{2} \times \frac{\delta}{2}$ squares starting from $p$
- Any two points within the same square are at most $\frac{\delta}{\sqrt{2}}$ apart.




- Because $\sqrt{2} > 1$, we know that $\frac{\delta}{\sqrt{2}} < \delta$
- Therefore, there is at most one point per square
- Besides the one which contains $p$ there are only 7 other squares within range $\delta$

# Full Algorithm

```
ClosestPair(L):
   L_x = L sorted by x coordinate
   L_y = L sorted by y coordinate
   return ClosestPairRec(L_x, L_y)
ClosestPairRec(L_x, L_y):
   # Base cases omitted
   m = median x coordinate
   P_x1 = the points from L_x to the left of the median
   P_y1 = the points from L_y to the left of the median
   P_x2 = the points from L_x to the right of the median
   P_y2 = the points from L_y to the right of the median
   a_1 = ClosestPair(P_x1, P_y1)
   a_2 = ClosestPair(P_x2, P_y2)
   a = closer of a_1 and a_2
   δ = distance(a)
   for each p in L_y:
      if p's x coordinate is more than δ from m:
         remove p from L_y
   for each p in L_y:
      for each of the next 7 points q in L_y:
         if distance(p, q):
            a = (p, q)
   return a
```