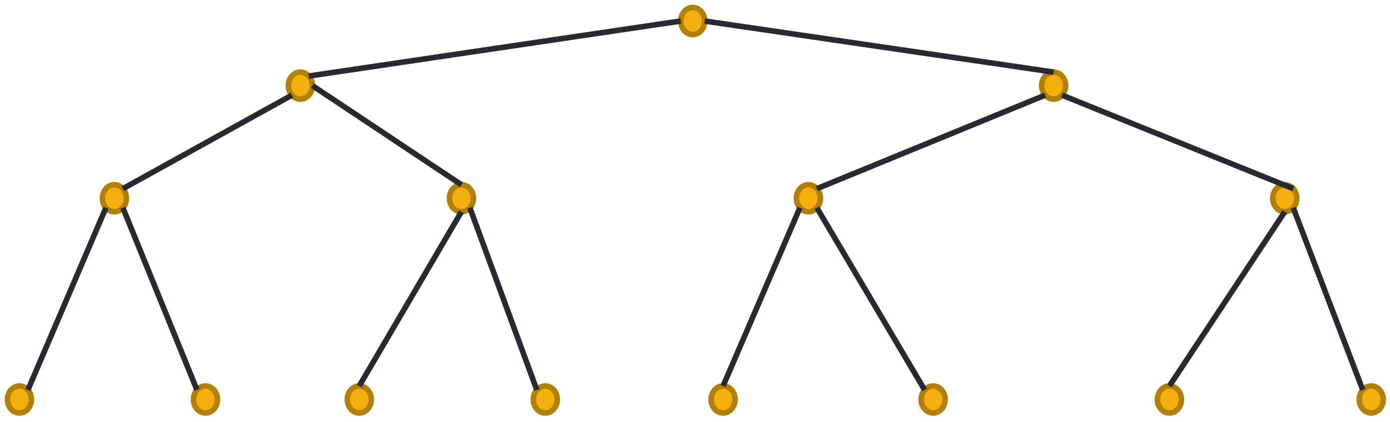
COL351: Slides for Lecture Component 20

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

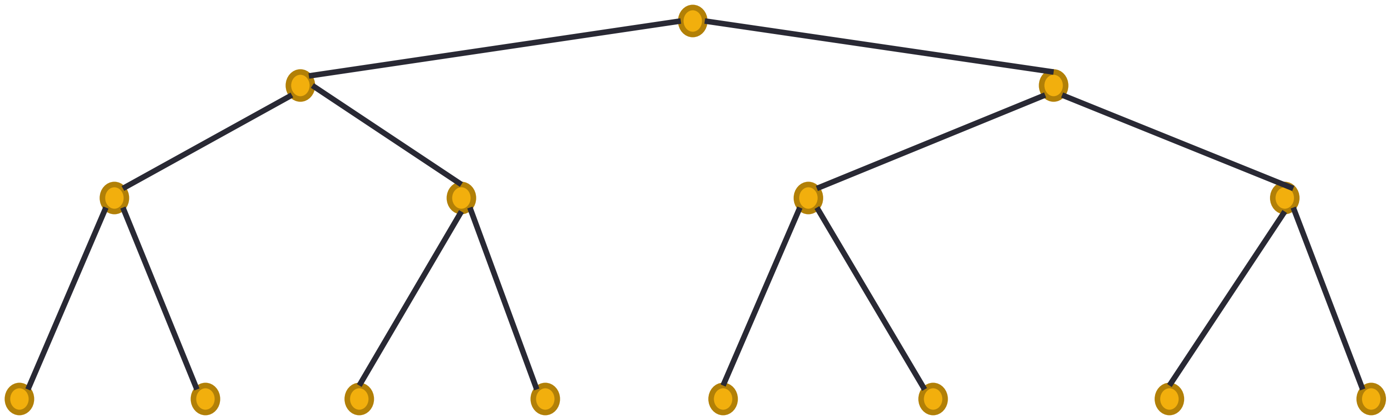
Divide and Conquer Trees

- Let’s say we have a full and balanced binary tree (all parents have two children and all leaves are on the bottom level.)



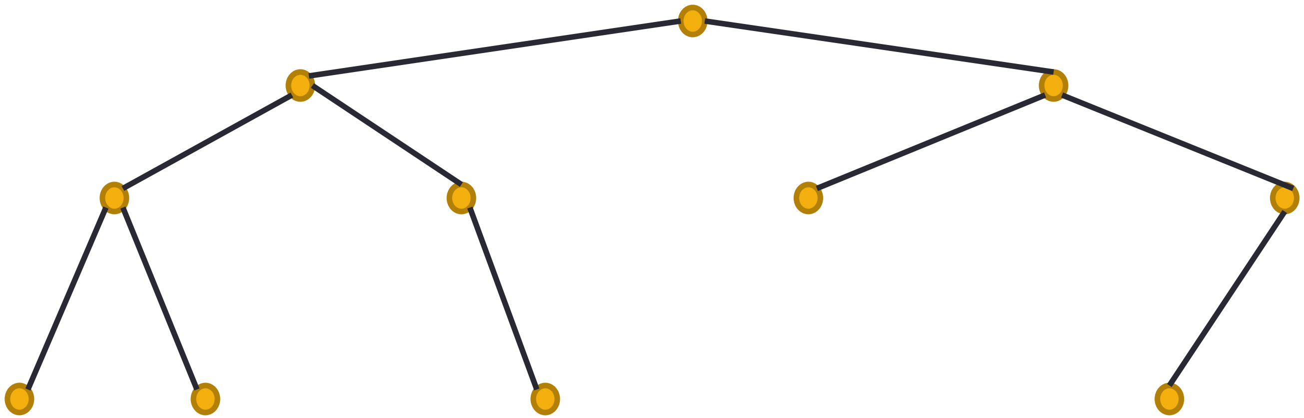
Divide and Conquer Trees

- Notice that each child's subtree is half of the problem so we get a nice divide and conquer structure.



Divide and Conquer Trees

- If the tree is uneven, we can still use the same strategy but we need to take a bit of care when calculating runtime.

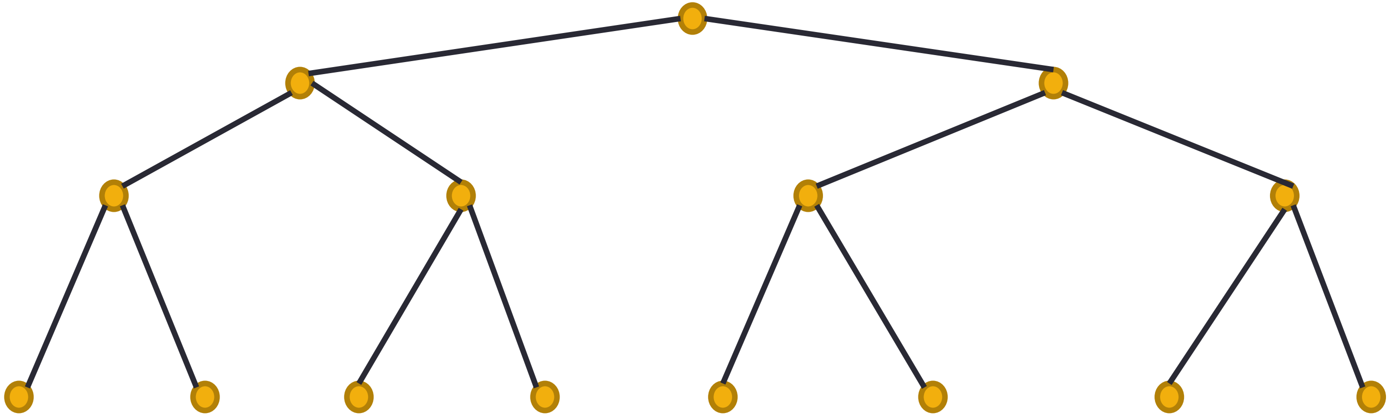


Least common ancestor

- Given a binary tree with n vertices, we wish to compute $LCA(x, y)$ for each pair of vertices x, y.
- $LCA(x, y)$ is the least common ancestor of x and y. Or in other words, the "youngest" common ancestor of x and y.
- For example, the LCA of me and my brother is our parent. The LCA of me and my uncle is my grandparent (his parent.) A vertex can be its own ancestor so the LCA of me and my father is my father.

Least common ancestor

- What pairs of vertices will have the root r as their least common ancestor?



Least common ancestor

- What pairs of vertices will have the root r as their least common ancestor?
- For each vertex v, set $lca(v, r) = r$.
- For each pair of vertices u, v such that u is in the left subtree and v is in the right subtree, set $lca(u, v) = r$.
- Now what? Are we done?
- Recurse on the left and right subtrees!!!!!

Pseudocode

```
Def LCA(r):
    Lsubtree = explore(r.lc)
    Rsubtree = explore(r.rc)
    for all vertices u in Lsubtree:
        lca(u, r) = r
    for all vertices v in Rsubtree:
        lca(r, v) = r
    for all vertices u in Lsubtree:
        for all vertices v in Rsubtree:
            lca(u, v) = r
    LCA(r.lc)
    LCA(r.rc)
```

Pseudocode (runtime)

```
Def LCA(r):
  Lsubtree = explore(r.lc)
  Rsubtree = explore(r.rc)
  for all vertices u in Lsubtree:
    lca(u,r) = r
  for all vertices v in Rsubtree:
    lca(r,v) = r
  for all vertices u in Lsubtree:
    for all vertices v in Rsubtree:
      lca(u,v) = r
  LCA(r.lc)
  LCA(r.rc)
```

If the binary tree is balanced, then each recursive call is of size $\frac{n-1}{2}$ or roughly half.

How long does the non-recursive part take?

Pseudocode (runtime)

Def **LCA**(r):

Lsubtree = **explore**(r.lc)

Rsubtree = **explore**(r.rc)

for all vertices u in Lsubtree:

$lca(u, r) = r$

for all vertices v in Rsubtree:

$lca(r, v) = r$

for all vertices u in Lsubtree:

for all vertices v in Rsubtree:

$lca(u, v) = r$

LCA(r.lc)

LCA(r.rc)

If the binary tree is balanced, then each recursive call is of size $\frac{n-1}{2}$ or roughly half.

How long does the non-recursive part take?

$$T(n) = 2T\left(\frac{n-1}{2}\right) + O(\mathrm{n}^2)$$

Using the master theorem with a=2, b=2, d=2,

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

Pseudocode (runtime uneven)

Def **LCA**(r):

    Lsubtree = **explore**(r.lc)

    Rsubtree = **explore**(r.rc)

    for all vertices u in Lsubtree:

        $lca(u, r) = r$

    for all vertices v in Rsubtree:

        $lca(r, v) = r$

    for all vertices u in Lsubtree:

        for all vertices v in Rsubtree:

            $lca(u, v) = r$

    LCA(r.lc)

    LCA(r.rc)

If the binary tree is uneven then the runtime recurrence is

$$T(n) = T(L) + T(R) + O(LR)$$

Where L is the size of the left subrtree and R is the size of the right subtree.

What do you think the total runtime will be? Take a guess and we can check it!!!

Uneven DC runtime

- $T(n) = T(L) + T(\mathrm{R}) + \mathrm{O(LR)}$
- We guess that it would take $O(n^2)$. So let's try to prove this using induction.
- <u>Claim</u>: $T(n) \leq cn^2$ for all $n \geq 1$ and for some constant c that is bigger than $T(1)$ and bigger than the coefficient in the $O(LR)$ term.

Uneven DC runtime

- Base case. $T(1) < c(1^2)$. True by choice of c.
- Suppose that for some $n > 1$, $T(k) < ck^2$ for all k such that $1 \leq k < n$.
- Then

$$T(n) < T(L) + T(R) + cLR \leq cL^2 + cR^2 + cLR$$
$$< cL^2 + cR^2 + 2cLR = c(L + R)^2 = c(n - 1)^2 < cn^2$$

Make Heap

- Problem: Given a list of n elements, form a heap containing all elements.

Divide and conquer strategy

- Assume $n = 2^k - 1$. (Add blank elements if needed)
- Divide the list into two lists of size $\frac{n-1}{2}$ and a left-over element
- Make heaps with both (in sub-trees of root)
- Put left-over element at root.
- “Trickle down” top element to reinstate heap property

Time analysis

- To solve one problem, we solve two problems of half the size, and then spend constant time per depth of the tree.

- $T(n) = \quad T(\quad) + O(\qquad)$

Time analysis

- To solve one problem, we solve two problems of half the size, and then spend constant time per depth of the tree.

- $T(n) = \quad 2\,T(\ n/2\ ) \ + O(\log n\ )$
- Doesn't fit master theorem.

Time analysis: sandwiching

- To solve one problem, we solve two problems of half the size, and then spend constant time per depth of the tree.

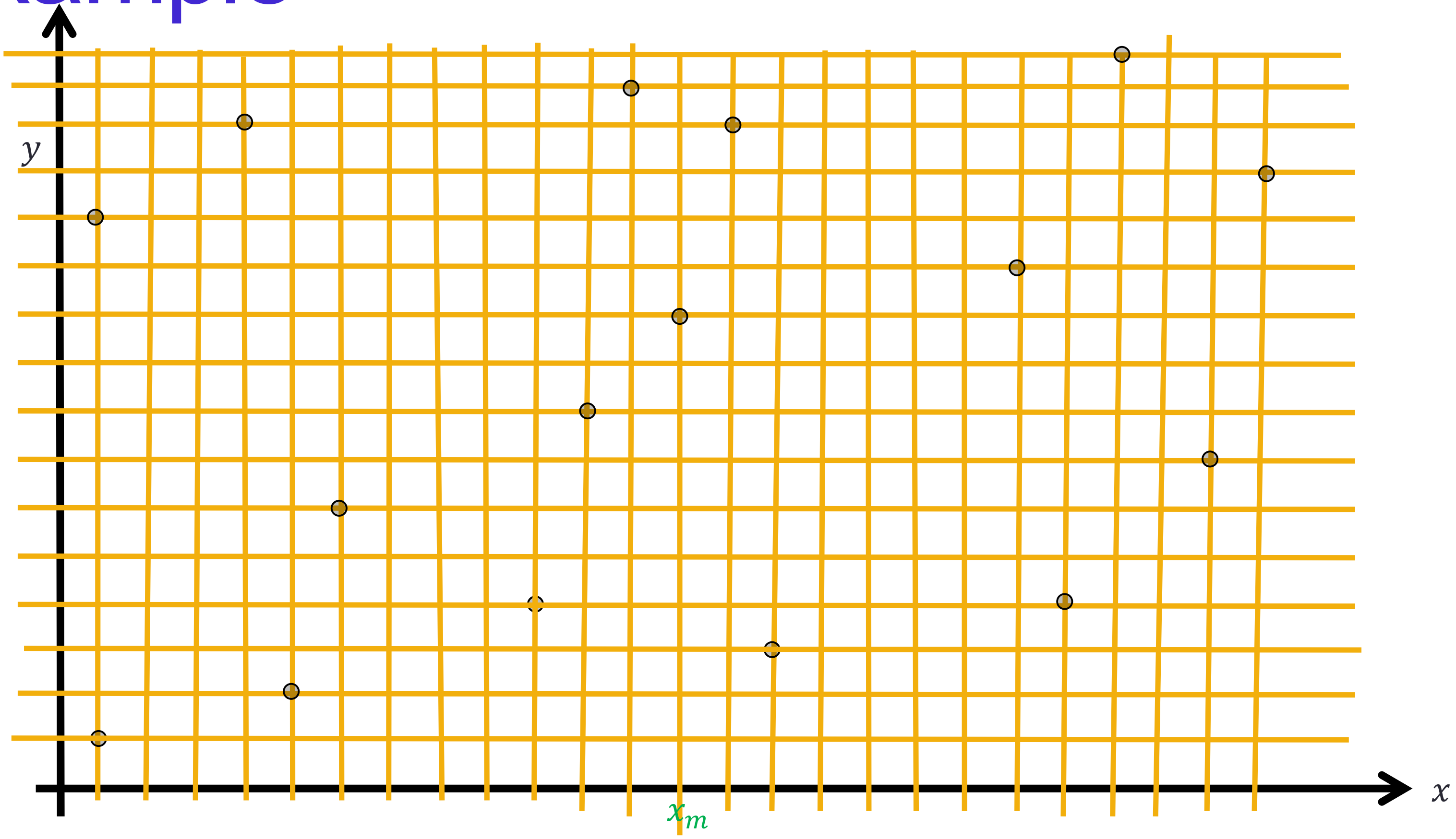
- T(n) = 2 T(n/2) + O(log n)
- Define L(n) =2 T(n/2) + O(1), H(n) = 2T(n/2) + $O\left(n^{\left\{\frac{1}{2}\right\}}\right)$
- L(n) < T(n) < H(n)
- Apply Master Theorem: Both L(n) and H(n) are O(n),
- So T(n) is O(n)

minimum distance

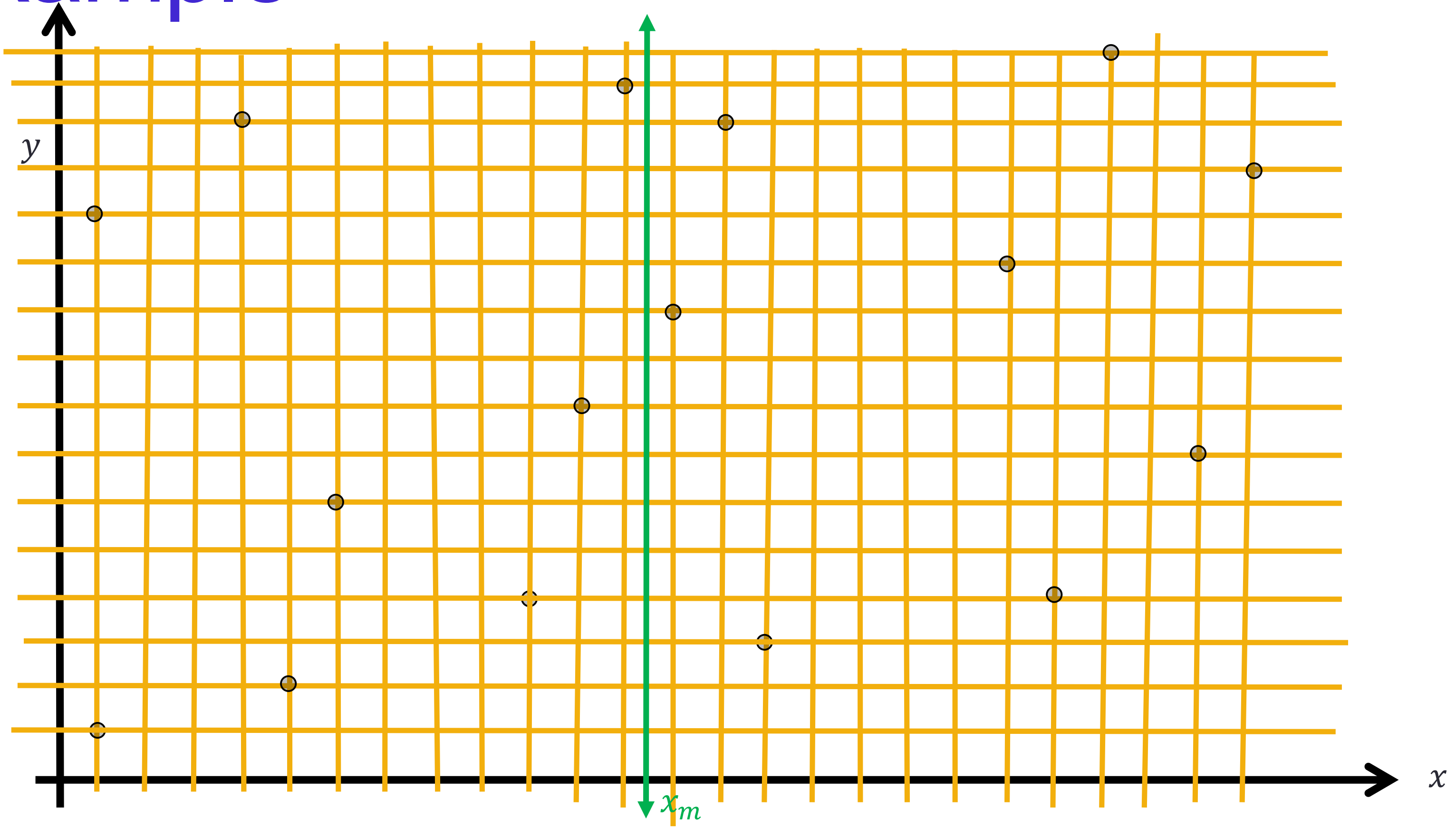
- Given a list of coordinates, $[(x_1, y_1), \ldots, (x_n, y_n)]$, find the distance between the closest pair.

- Brute force solution?
- min = 0
- for i from 1 to n-1:
  - for j from i+1 to n:
    - if min > distance$((x_i, y_i), (x_j, y_j))$
- return min

Example



Example



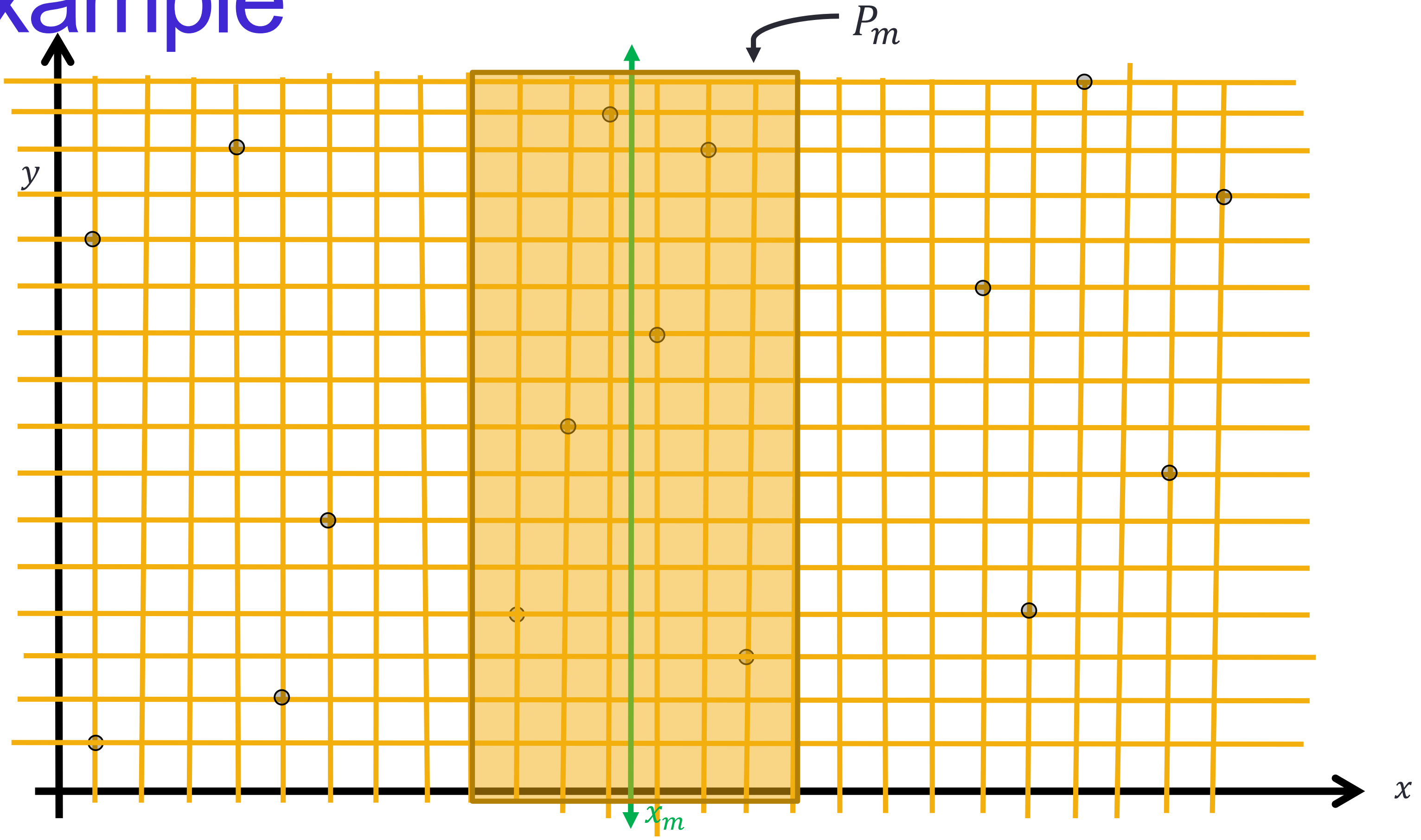
Divide and conquer

- Partition the points by x, according to whether they are to the left or right of the median
- Recursively find the minimum distance points on the two sides.
- Need to compare to the smallest "cross distance" between a point on the left and a point on the right
- Only need to look at "close" points

Combine

- How will we use this information to find the distance of the closest pair in the whole set?
- We must consider if there is a closest pair where one point is in the left half and one is in the right half.
- How do we do this?
- Let $d = \min(d_L, d_R)$ and compare only the points (x_i, y_i) such that $x_m - d \leq x_i$ and $x_i \leq x_m + d$.

Example



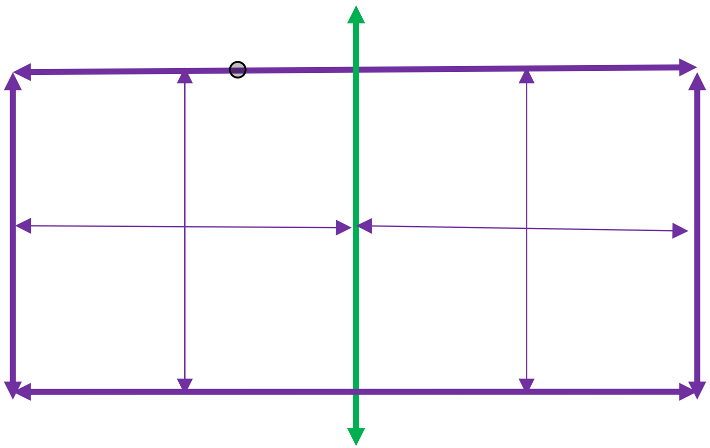
Combine

- How will we use this information to find the distance of the closest pair in the whole set?
- We must consider if there is a closest pair where one point is in the left half and one is in the right half.
- How do we do this?
- Let $d = \min(d_L, d_R)$ and compare only the points (x_i, y_i) such that $x_m - d \leq x_i$ and $x_i \leq x_m + d$.

- Worst case, how many points could this be?

Combine step

- Given a point $(x, y) \in P_m$, let's look in a $2d \times d$ rectangle with that point at its upper boundary:



- There could not be more than 8 points total because if we divide the rectangle into 8 $\frac{d}{2} \times \frac{d}{2}$ squares then there can never be more than one point per square.
- Why???

Combine step

- So instead of comparing (x, y) with every other point in P_m we only have to compare it with at most a constant **c** points lower than it (smaller y)
- To gain quick access to these points, let's sort the points in P_m by y values.
- The points above must be in the **c** points before our current point in this sorted list

- Now, if there are k vertices in P_m we have to sort the vertices in $O(k\log k)$ time and make at most ck comparisons in $O(k)$ time for a total combine step of $O(k \log k)$.

- But we said in the worst case, there are n vertices in P_m and so worst case, the combine step takes $O(n \log n)$ time.

Time analysis

- But we said in the worst case, there are n vertices in P_m and so worst case, the combine step takes $O(n \log n)$ time.

- Runtime recursion:

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

This is T(n) = O(n (log n)^2)

Pre-processing : Sort by both x and y, keep pointers between sorted lists Maintain sorting in recursive calls reduces to T(n) =2 T(n/2) +O(n), so T(n) is O(n log n)