

# Master Theorem

- How do you solve a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

We will use the master theorem.

# Summation Lemma

Consider the summation

$$\sum_{k=0}^n r^k$$

It behaves differently for different values of  $r$ .

# Summation Lemma

Consider the summation

$$\sum_{k=0}^n r^k$$

It behaves differently for different values of  $r$ .

If  $r < 1$  then this sum converges. This means that the sum is bounded above by some constant  $c$ . Therefore

$$\text{if } r < 1, \quad \text{then } \sum_{k=0}^n r^k < c \text{ for all } n \text{ so } \sum_{k=0}^n r^k \in O(1)$$

# Summation Lemma

Consider the summation

$$\sum_{k=0}^n r^k$$

It behaves differently for different values of  $r$ .

If  $r = 1$  then this sum is just summing 1 over and over  $n$  times. Therefore

$$\text{if } r = 1, \quad \text{then } \sum_{k=0}^n r^k = \sum_{k=0}^n 1 = n + 1 \in O(n)$$

# Summation Lemma

Consider the summation

$$\sum_{k=0}^n r^k$$

It behaves differently for different values of  $r$ .

If  $r > 1$  then this sum is exponential with base  $r$ .

$$\text{if } r > 1, \text{ then } \sum_{k=0}^n r^k < cr^n \text{ for all } n, \quad \text{so } \sum_{k=0}^n r^k \in O(r^n) \quad \left( c > \frac{r}{r-1} \right)$$

# Summation Lemma

Consider the summation

$$\sum_{k=0}^n r^k$$

It behaves differently for different values of  $r$ .

$$\sum_{k=0}^n r^k \in \begin{cases} O(1) & \text{if } r < 1 \\ O(n) & \text{if } r = 1 \\ O(r^n) & \text{if } r > 1 \end{cases}$$

# Master Theorem

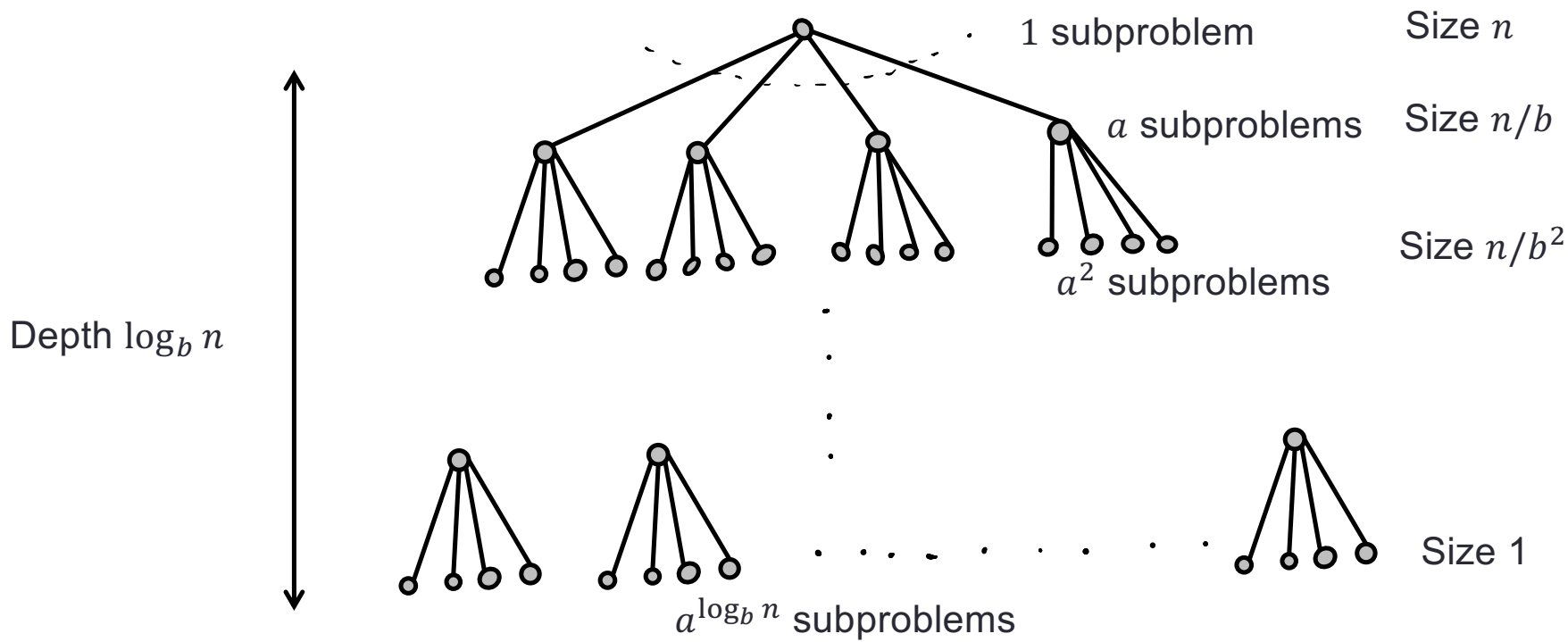
Master Theorem: If  $T(n) = aT(n/b) + O(n^d)$  for some constants  $a > 0, b > 1, d \geq 0$ ,

Then

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

# Master Theorem: Solving the recurrence

$$T(n) = aT(n/b) + O(n^d)$$





# Master Theorem: Solving the recurrence

After  $k$  levels, there are  $a^k$  subproblems, each of size  $n/b^k$ .

So, during the  $k$ th level of recursion, the time complexity is

$$\begin{aligned} O\left(\left(\frac{n}{b^k}\right)^d\right) a^k &= O\left(a^k \left(\frac{n}{b^k}\right)^d\right) \\ &= O\left(n^d \left(\frac{a}{b^d}\right)^k\right) \end{aligned}$$

# Master Theorem: Solving the recurrence

After  $k$  levels, there are  $a^k$  subproblems, each of size  $n/b^k$ .

$$\begin{aligned} \text{So, during the } k\text{th level, the time complexity is } & O\left(\left(\frac{n}{b^k}\right)^d\right) a^k = O\left(a^k \left(\frac{n}{b^k}\right)^d\right) \\ & = O\left(n^d \left(\frac{a}{b^d}\right)^k\right) \end{aligned}$$

After  $\log_b n$  levels, the subproblem size is reduced to 1, which usually is the size of the base case.

So the entire algorithm is a sum of each level.

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

# Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 1:  $a < b^d$

Then we have that  $\frac{a}{b^d} < 1$  and the series converges to a constant so

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

# Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 2:  $a = b^d$

Then we have that  $\frac{a}{b^d} = 1$  and so each term is equal to 1

$$T(n) = O(n^d \log_b n)$$

# Master Theorem: Proof

$$T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Case 2:  $a > b^d$

Then the summation is exponential and grows proportional to its last term

$\left(\frac{a}{b^d}\right)^{\log_b n}$  so

$$T(n) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O(n^{\log_b a})$$

# Master Theorem

Theorem: If  $T(n) = aT(n/b) + O(n^d)$  for some constants  $a > 0, b > 1, d \geq 0$ ,

Then

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Top-heavy

Steady-state

Bottom-heavy

# Master Theorem Applied to Multiply

The recursion for the runtime of Multiply is

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

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So we have that  $a=4$ ,  $b=2$ , and  $d=1$ . In this case,  $a > b^d$  so

$$T(n) \in O(n^{\log_2 4}) = O(n^2)$$

Not any improvement of grade-school method.

# Master Theorem Applied to MultiplyKS

The recursion for the runtime of Multiply is

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

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So we have that  $a=3$ ,  $b=2$ , and  $d=1$ . In this case,  $a > b^d$  so

$$T(n) \in O(n^{\log_2 3}) = O(n^{1.58})$$

An improvement on grade-school method!!!!!!



# Poll: What is the fastest known integer multiplication time?

- $O(n^{\log 3})$
- $O(n \log n (\log(\log n))^2)$
- $O(n \log n 2^{\{\log^* n\}})$
- $O(n \log n)$
- $O(n)$

# Poll: What is the fastest known integer multiplication time? All have/will be correct

- $O(n^{\log 3})$  **Kuratsuba**
- $O(n \log n \log \log n)$  **Schonhage-Strassen**, 1971
- $O(n \log n 2^{\{c \log^* n\}})$  **Furer**, 2007
- $O(n \log n)$  **Harvey and van der Hoeven**, 2019
- $O(n)$ , you, tomorrow?

# Can we do better than $n^{1.58}$ ?

- Could any multiplication algorithm have a faster asymptotic runtime than  $\Theta(n^{1.58})$ ?
- Any ideas?????

# Can we do better than $n^{1.58}$ ?

- What if instead of splitting the number in half, we split it into thirds.



# Can we do better than $n^{1.58}$ ?

- What if instead of splitting the number in half, we split it into thirds.

- $x = 2^{2n/3}x_L + 2^{n/3}x_M + x_R$

- $y = 2^{2n/3}y_L + 2^{n/3}y_M + y_R$

# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$

# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$   
 $= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf$

9 multiplications means 9 recursive calls.

Each multiplication is 1/3 the size of the original.

# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$   
 $= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf$

9 multiplications means 9 recursive calls.

Each multiplication is 1/3 the size of the original.

$$T(n) = 9T\left(\frac{n}{3}\right) + O(n)$$



# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$   
 $= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf$

$$T(n) = 9T\left(\frac{n}{3}\right) + O(n)$$

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$$a=9$$

$$b=3$$

$$d=1$$

$$9 > 3^1$$

$$T(n) = O(n^{\log_3 9})$$

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

# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$   
 $= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf$
- There is a way to reduce from 9 multiplications down to just 5!!!
- Then the recursion becomes
- $T(n) = 5T(n/3) + O(n)$
- So by the master theorem

# Multiplying trinomials

- $(ax^2 + bx + c)(dx^2 + ex + f)$   
 $= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf$
- There is a way to reduce from 9 multiplications down to just 5!!!
- Then the recursion becomes
- $T(n) = 5T(n/3) + O(n)$
- So by the master theorem  $T(n) = O(n^{\log_3 5}) = O(n^{1.43})$

# Dividing into k subproblems

- What happens if we divide into k subproblems each of size  $n/k$ .
- $(a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \dots a_1x + a_0)(b_{k-1}x^{k-1} + b_{k-2}x^{k-2} + \dots b_1x + b_0)$
- How many terms are there? (multiplications.)

# Dividing into k subproblems

- What happens if we divide into k subproblems each of size n/k.
- $(a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \dots + a_1x + a_0)(b_{k-1}x^{k-1} + b_{k-2}x^{k-2} + \dots + b_1x + b_0)$
- How many terms are there? (multiplications.)
- There are  $k^2$  multiplications. The recursion is

$$T(n) = k^2 T\left(\frac{n}{k}\right) + O(n) \dots \dots \dots a = k^2, b = k, d = 1$$

$$T(n) = O(n^{\log_k k^2}) = O(n^2)$$

# Cook-Toom algorithm

- In fact, if you split up your number into  $k$  equally sized parts, then you can combine them with  $2k-1$  multiplications instead of the  $k^2$  individual multiplications.
- This means that you can get an algorithm that runs in
- $T(n) = (2k - 1)T(n/k) + O(n)$

# Cook-Toom algorithm

- In fact, if you split up your number into  $k$  equally sized parts, then you can combine them with  $2k-1$  multiplications instead of the  $k^2$  individual multiplications.
- This means that you can get an algorithm that runs in
- $T(n) = (2k - 1)T(n/k) + O(n)$
- $T(n) = O\left(n^{\frac{\log(2k-1)}{\log k}}\right)$  time!!!!

# Cook-Toom algorithm

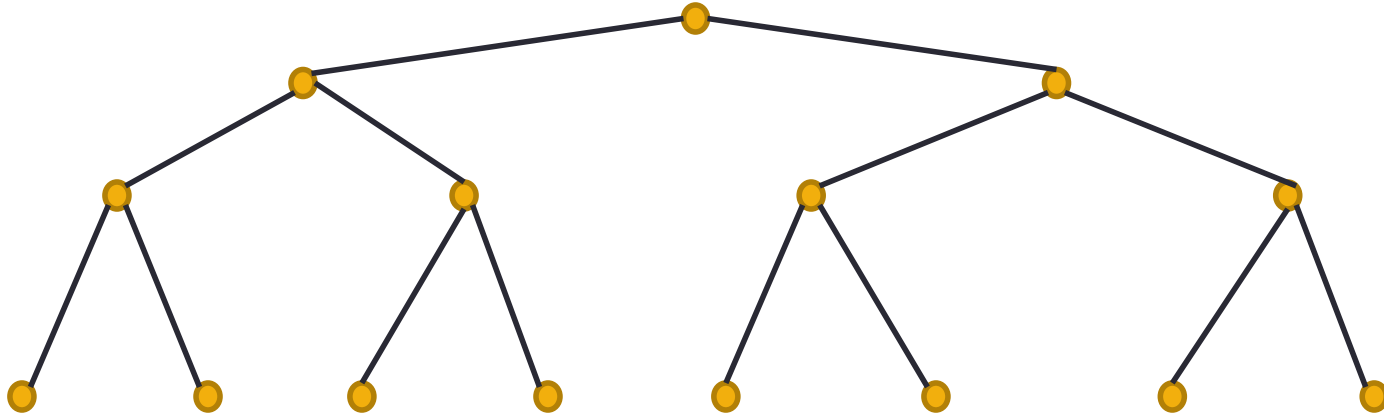
$$T(n) = (2k - 1)T(n/k) + O(n)$$

- $T(n) = O\left(n^{\frac{\log 2k-1}{\log k}}\right)$  time.
- So we can have a near-linear time algorithm if we take  $k$  to be sufficiently large. The  $O(n)$  term in the recursion takes a lot of time the bigger  $k$  gets. So is it worth it to make  $k$  very large?



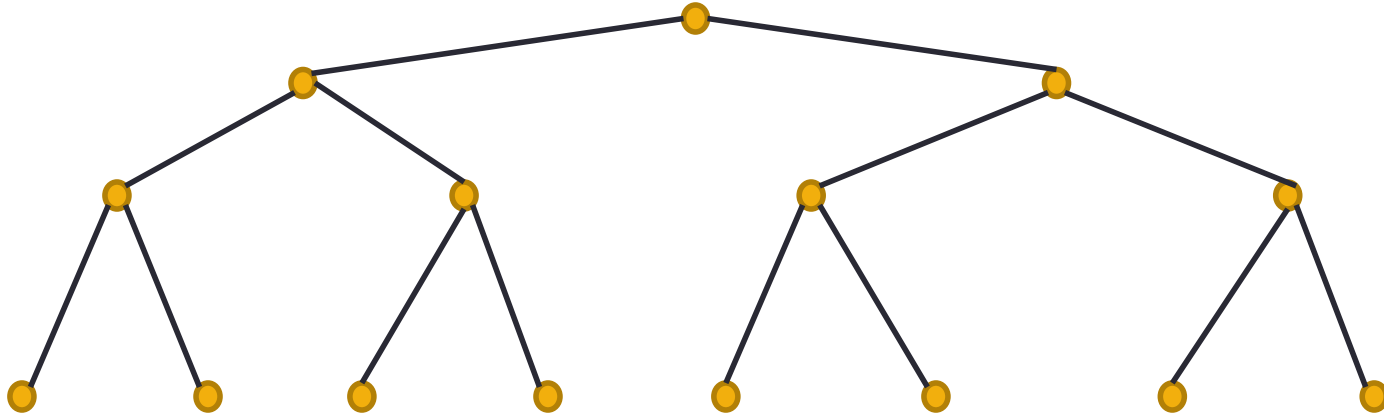
# Divide and Conquer Trees

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



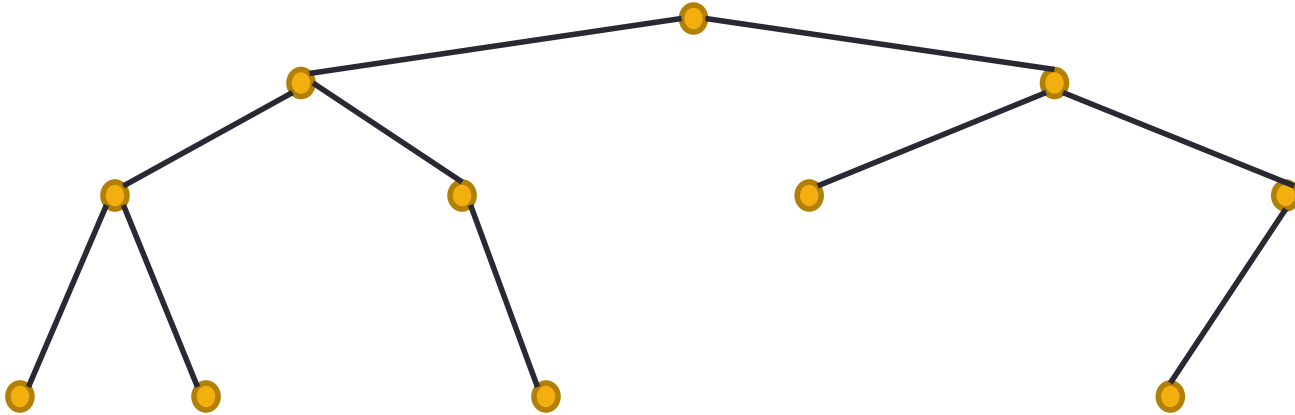
# Divide and Conquer Trees

- 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.

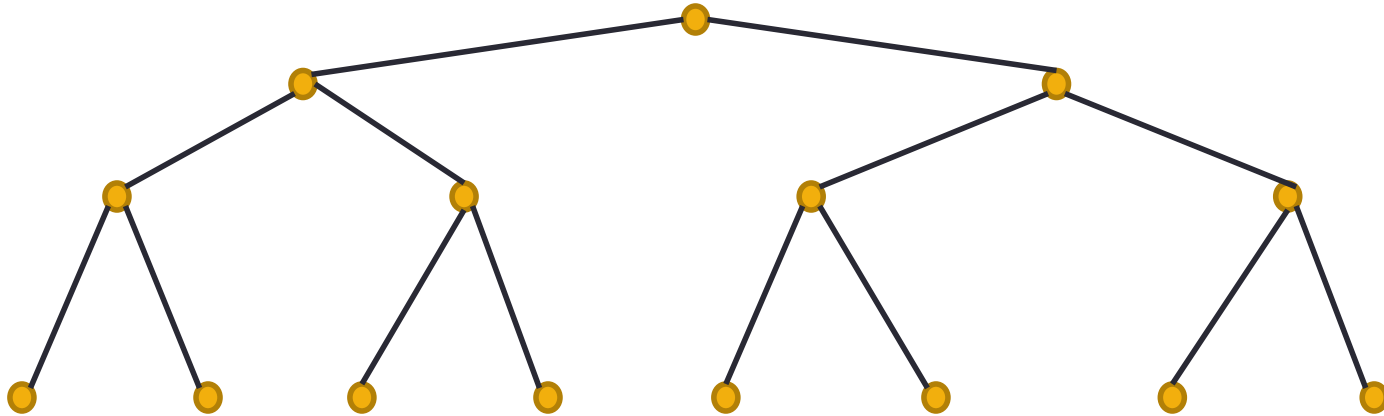


# Least common ancestor

- 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?

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

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 subtree 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

- $T(n) = T(L) + T(R) + O(LR)$
- We guess that it would take  $O(n^2)$ . So let's try to prove this using induction.
- Claim:  $T(n) \leq cn^2$  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

$$\begin{aligned} T(n) &< T(L) + T(R) + cLR \leq cL^2 + cR^2 + cLR \\ &< cL^2 + cR^2 + 2cLR = c(L + R)^2 = c(n - 1)^2 < cn^2 \end{aligned}$$

# 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.
- $T(n) = 2T(n/2) + O(1)$

# Time analysis

- To solve one problem, we solve two problems of half the size, and then spend constant time per depth of the tree.
- $T(n) = 2 T( n/2 ) + O(\log n )$
- Doesn't fit master theorem.



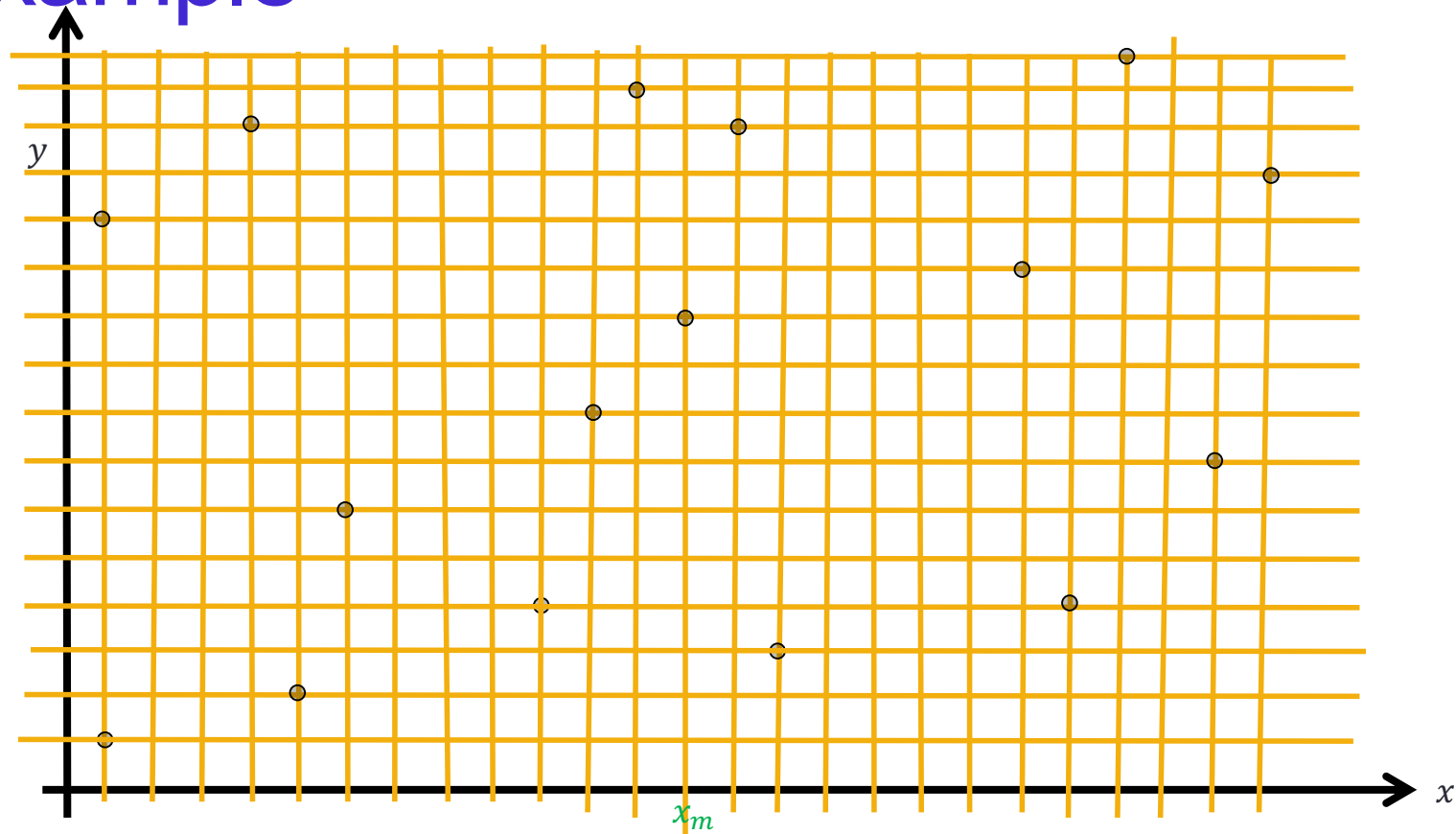
# Time analysis: sandwiching

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

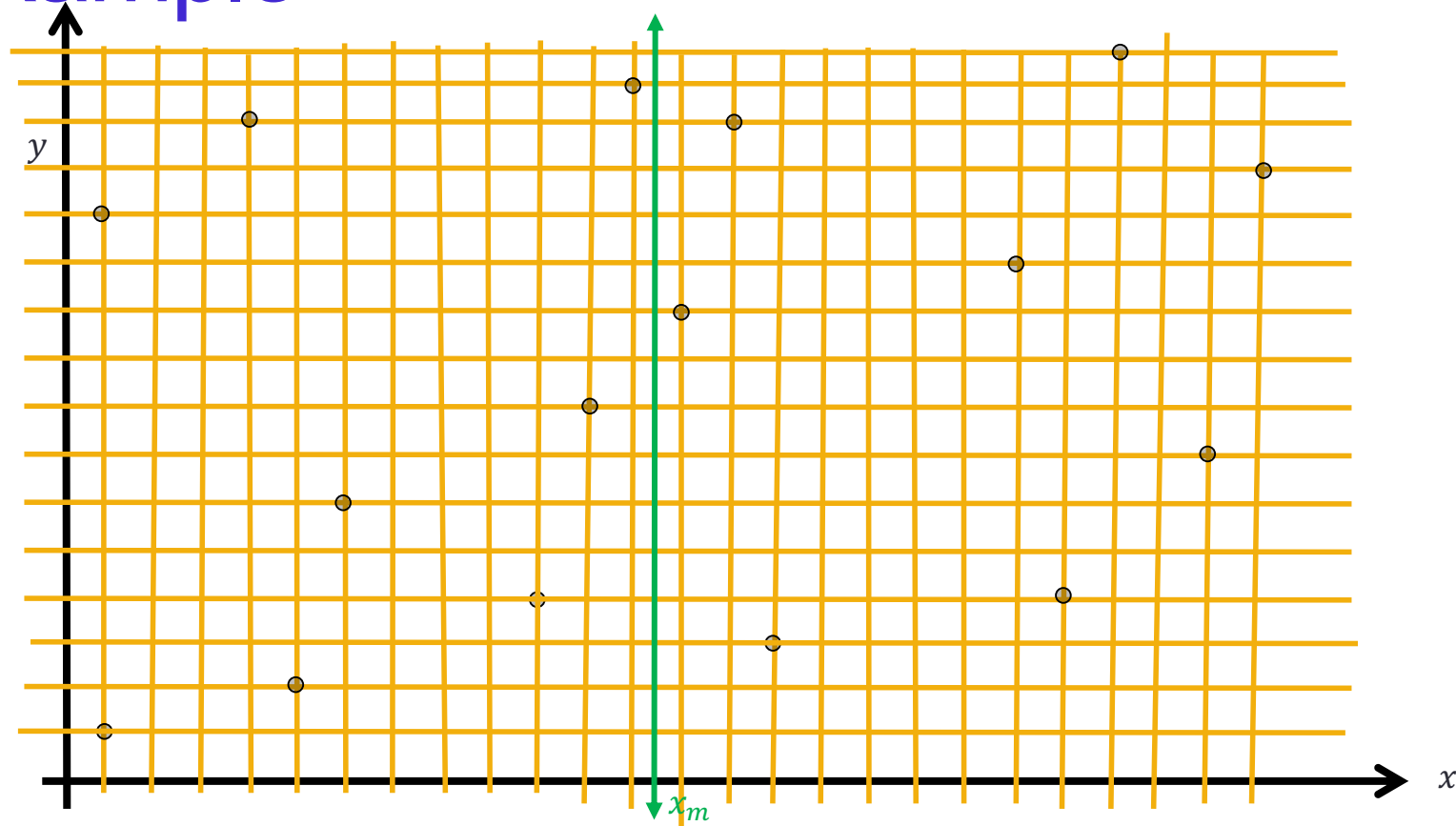
# minimum distance

- Given a list of coordinates,  $[(x_1, y_1), \dots, (x_n, y_n)]$ , find the distance between the closest pair.
- Brute force solution?
- $\text{min} = 0$
- for  $i$  from 1 to  $n-1$ :
  - for  $j$  from  $i+1$  to  $n$ :
    - if  $\text{min} > \text{distance}((x_i, y_i), (x_j, y_j))$
- return  $\text{min}$

# Example



# Example



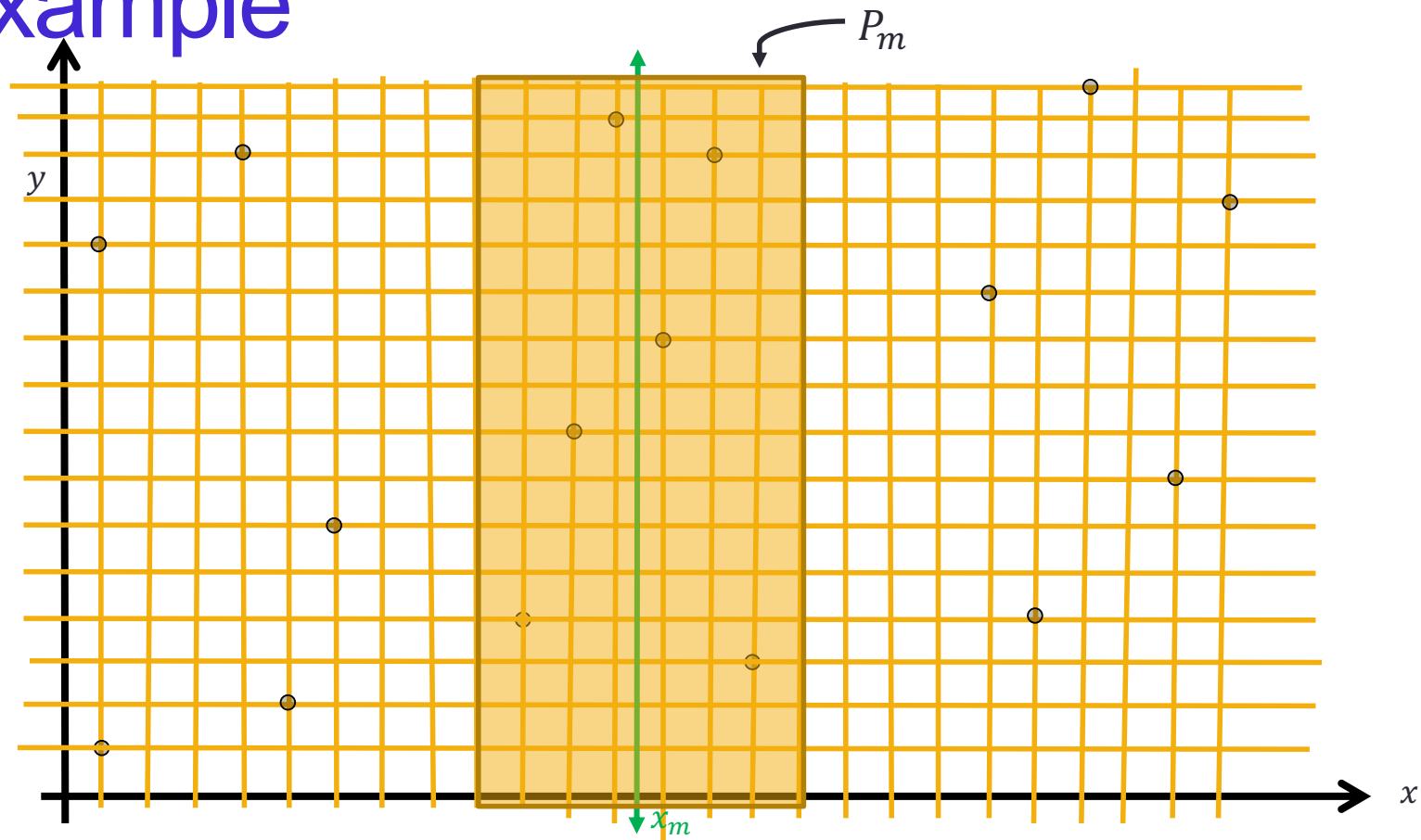
# Divide and conquer

- 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



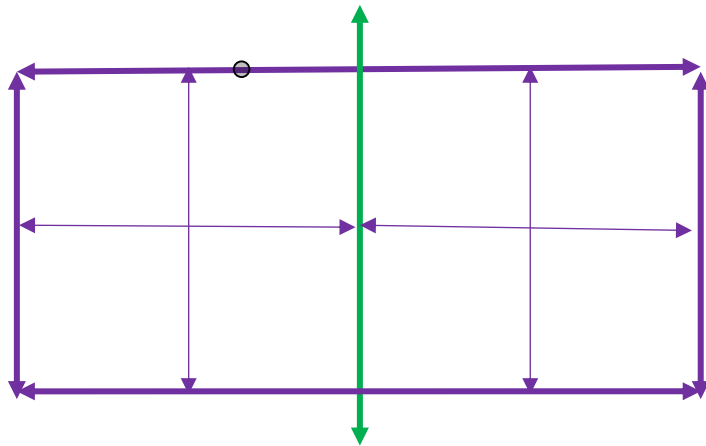
# Combine

- 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  $O(n \log n)$