



# CS240 Algorithm Design and Analysis

## Lecture 3

### Greedy Algorithms (Cont.) Divide and Conquer

Quan Li  
Fall 2024  
2024.09.26



# Last Time – What you need to know



- Basic idea
  - A greedy algorithm always makes the choice that **looks best at the moment** and adds it to the current partial solution
  - Greedy algorithms don't always yield optimal solutions, but when they do, they're usually the **simplest and most efficient** algorithms available
  - Make the locally optimal choice at each step
- Algorithms
  - Interval Scheduling
    - Choose the job with the **earliest finish time**
  - Scheduling to Minimize Lateness
    - Choose the job with the **earliest deadline**





# Clustering





# Clustering



Photos, documents, micro-organisms



- **Clustering.** Given a set  $U$  of  $n$  objects labeled  $p_1, \dots, p_n$ , partition into clusters s.t. objects in different clusters are far apart.



e.g., a large number of corresponding pixels whose intensities differ over some threshold

- **Fundamental problem.** Divide into clusters so that points in different clusters are far apart.
  - Routing in mobile ad hoc networks
  - Identify patterns in gene expression
  - Document categorization for web search
  - Similarity searching in medical image databases
  - Skycat: cluster  $10^9$  sky objects into stars, quasars, galaxies

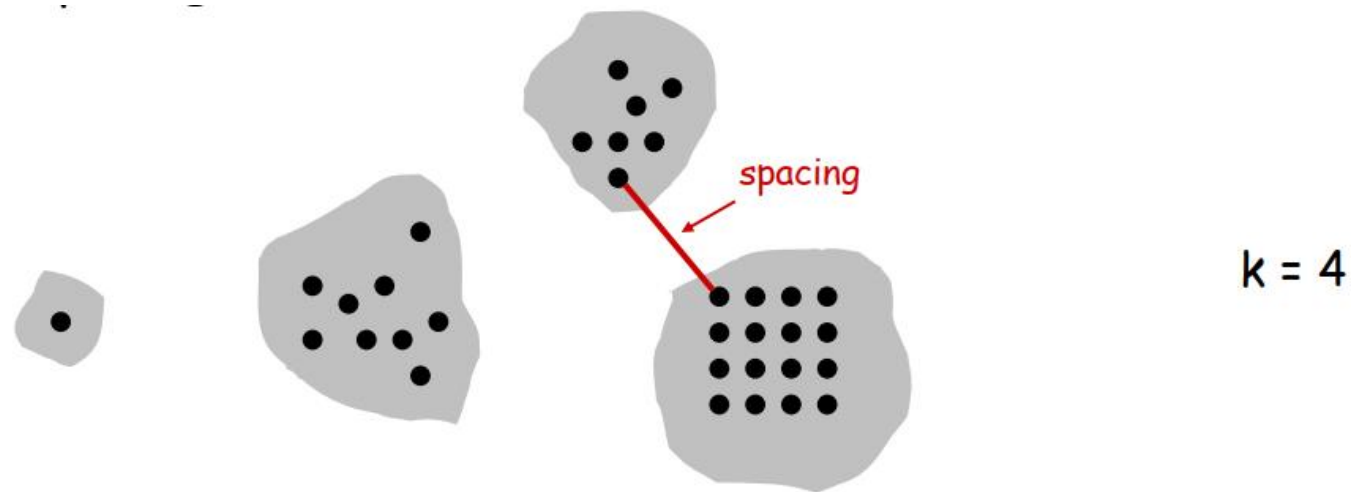




# Clustering of Maximum Spacing



- **k-clustering.** Divide objects into  $k$  non-empty groups
- **Distance function.** Assume it satisfies several natural properties
  - $d(p_i, p_j) = 0$  iff  $p_i = p_j$  (identity of indiscernible)
  - $d(p_i, p_j) \geq 0$  (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$  (symmetry)
- **Spacing.** Min distance between any pair of points in different clusters
- **Clustering of maximum spacing.** Given an integer  $k$ , find a  $k$ -clustering of maximum spacing

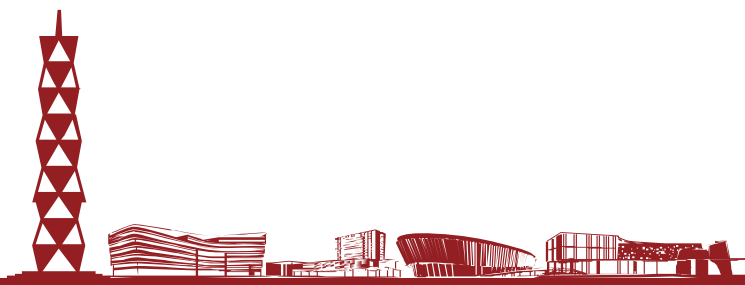




# Greedy Clustering Algorithm



- **Single-link k-clustering algorithm.**
  - Create  $n$  clusters, one for each object
  - Find the closest pair of objects such that each object is in a different cluster; add an edge between them and merge the two clusters
  - Repeat  $n-k$  times until there are exactly  $k$  clusters
- **Key observation.** This procedure is precisely Kruskal's algorithm (except we stop when there are  $k$  connected components)
- **Remark.** Equivalent to finding an MST and deleting the  $k-1$  most expensive edges

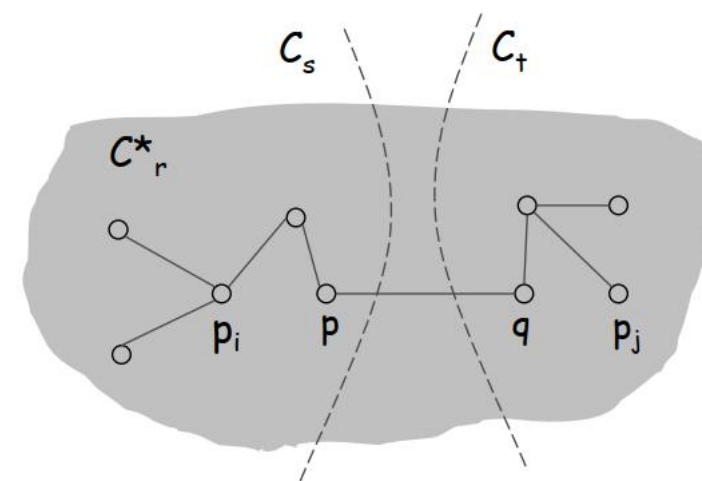




# Greedy Clustering Algorithm: Analysis

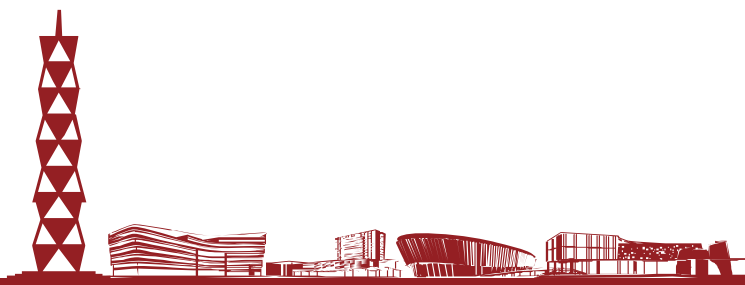


- **Theorem.** Let  $C^*$  denote the clustering  $C_1^*, \dots, C_k^*$  formed by deleting the  $k-1$  most expensive edges of a MST.  $C^*$  is a  $k$ -clustering of max spacing.
- Pf. Let  $C$  denote some other clustering  $C_1, \dots, C_k$ 
  - The spacing of  $C^*$  is the length  $d^*$  of the  $(k-1)^{\text{st}}$  most expensive edge in MST
  - Let  $p_i, p_j$  be in the same cluster in  $C^*$ , say  $C_r^*$ , but different clusters in  $C$ , say  $C_s$  and  $C_t$
  - Some edge  $(p, q)$  on  $p_i \text{ --- } p_j$  path in  $C_r^*$  spans two different clusters in  $C$ .
  - All edges on  $p_i \text{ --- } p_j$  path have length  $\leq d^*$  since Kruskal chose them
  - Spacing of  $C$  is  $\leq d^*$  since  $p$  and  $q$  are in different clusters





# Optimal Caching







# Optimal Offline Caching



- **Caching**

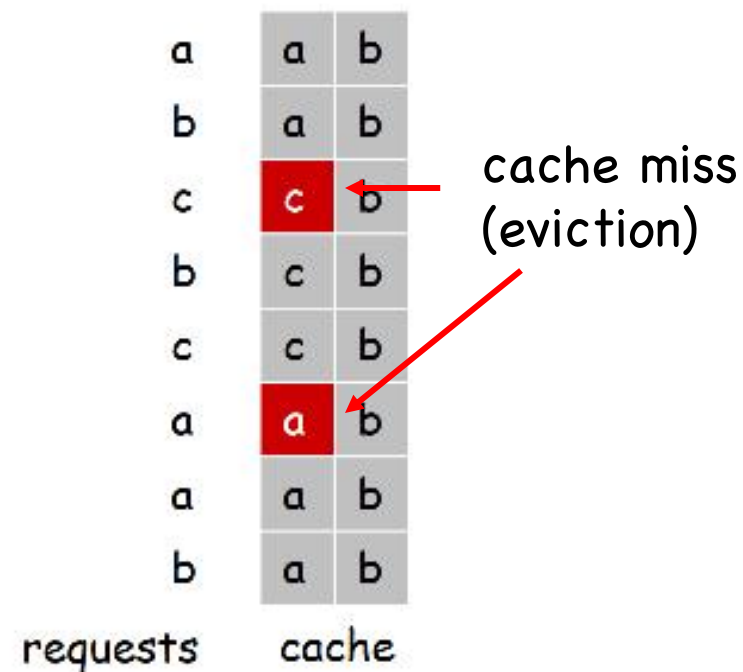
- Cache with capacity to store  $k$  items
- Sequence of  $m$  item requests  $d_1, d_2, \dots, d_m$
- **Cache hit:** item already in cache when requested
- **Cache miss:** item not already in cache when requested; must bring requested item into cache, and evict some existing item, if full

- **Applications.** CPU, RAM, hard drive, web, browser, ...

- **Goal.** Eviction schedule that minimizes number of evictions

- Ex:  $k = 2$ , initial cache =  $ab$ , requests:  $a, b, c, b, c, a, a, b$

- **Optimal eviction schedule:** 2 evictions



- **Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future

future queries:    g a b c e d a b b a c d e a f a d e f g h ...

↑  
cache miss

↑  
eject this one

- **Theorem.** FF is optimal eviction schedule.
- **Pf.** Algorithm and theorem are intuitive; proof is subtle





# Quiz



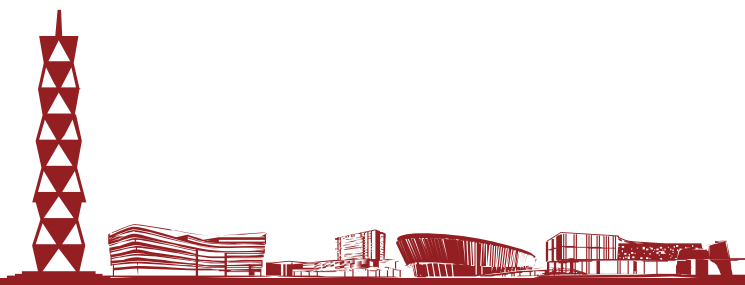
- Which item will be evicted next using farthest-in-future schedule?

- A
- B
- C
- D
- E

cache

r e q u e s t	⋮	.	.	.	.
	B	D	B	Y	A
	C	D	B	C	A
	E	D	E	C	A
	F	?	?	?	?
	C				
	D				
	A				
	E				
	A				
	C				
	⋮				

← cache miss  
(which item to eject?)





# Reduced Eviction Schedules



- **Def.** A **reduced** schedule is a schedule that only inserts an item  $d$  into the cache in a step in which  $d$  is requested and  $d$  is not already in the cache
- **Intuition.** Can transform an unreduced schedule into a reduced one with no more evictions

- $x$  enters cache without a request
- $d$  enters cache without a request
- $b$  enters cache without a request
- $c$  enters cache without a request
- $x$  enters cache without a request

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

a reduced schedule





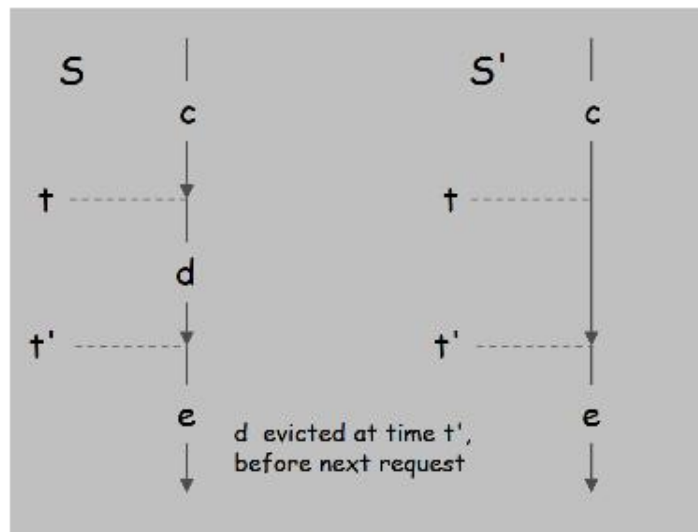
# Reduced Eviction Schedules



- **Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more evictions
- **Pf.** (by induction on number of unreduced items) ← doesn't enter cache at requested time

Suppose  $S$  brings  $d$  into the cache at time  $t$ , without a request, let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache

- **Case 1:  $d$  evicted at time  $t'$ , before next request for  $d$**



Case 1

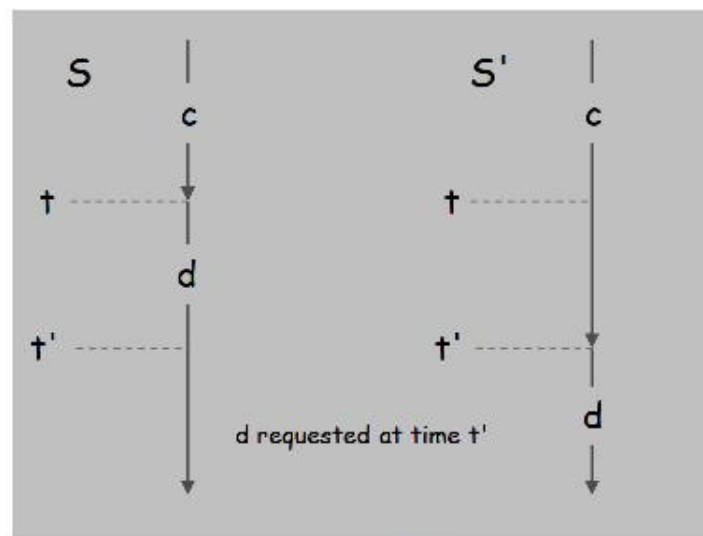




# Reduced Eviction Schedules



- **Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more evictions
- **Pf.** (by induction on number of unreduced items) ← doesn't enter cache at requested time
- Suppose  $S$  brings  $d$  into the cache at time  $t$ , without a request
- Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache
- Case 1:  $d$  evicted at time  $t'$ , before next request for  $d$
- **Case 2:  $d$  requested at time  $t'$  before  $d$  is evicted**



Case 2





# Farthest-In-Future: Analysis



- **Lemma.** Let  $S$  be a reduced schedule that makes the same schedule as  $S_{FF}$  through the first  $j$  requests. Then there is a reduced schedule  $S'$  that makes the same schedule as  $S_{FF}$  through the first  $j+1$  requests, and incurs no more eviction that  $S$  does
- **Pf.**
  - Consider  $(j+1)^{st}$  request  $d = d_{j+1}$
  - Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before request  $j+1$
  - Case 1: ( $d$  is already in the cache).  $S' = S$  satisfies invariant
  - Case 2: ( $d$  is not in the cache and  $S$  and  $S_{FF}$  evict the same element).  
 $S' = S$  satisfies invariant



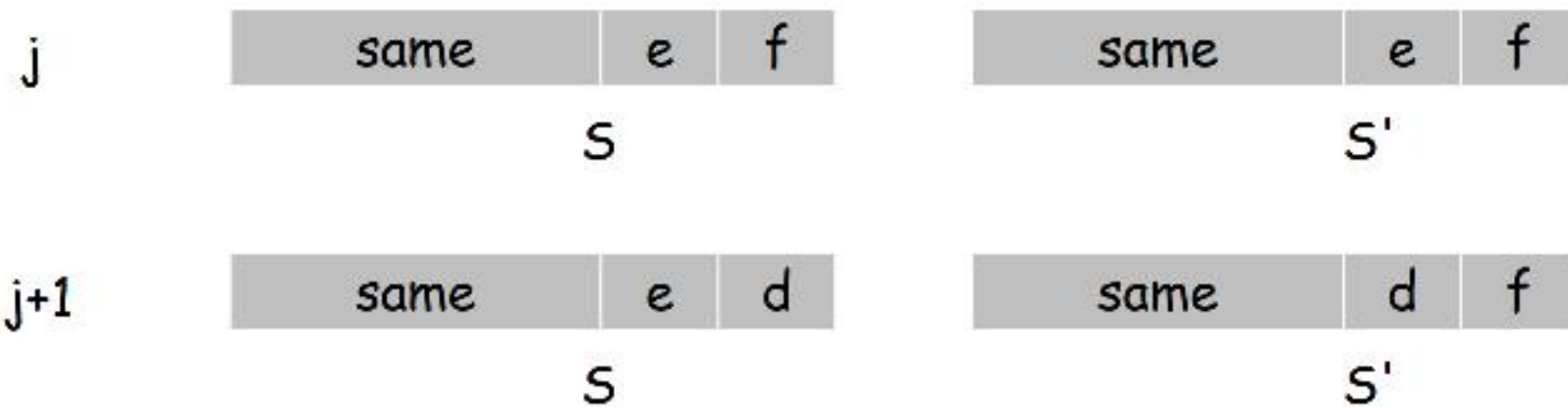


# Farthest-In-Future: Analysis



- **Pf. (continued)**

- Case 3: ( $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ )
  - Begin construction of  $S'$  from  $S$  by evicting  $e$  instead of  $f$



- Now  $S'$  agrees with  $S_{FF}$  on first  $j+1$  requests
- From request  $j+2$  onward, we make  $S'$  the same as  $S$ , but this becomes impossible when  $e$  or  $f$  is involved







# Farthest-In-Future: Analysis



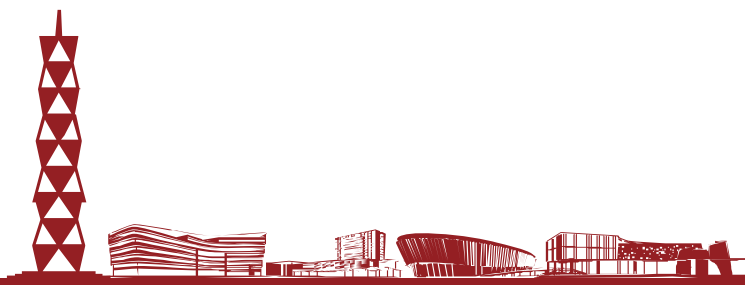
- **Pf. (continued)**

- Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$

must involve  $e$  or  $f$  (or both)



Case 3a:  $g = e$ . Cannot happen with Farthest-In-Future since there must be a request for  $f$  before  $e$





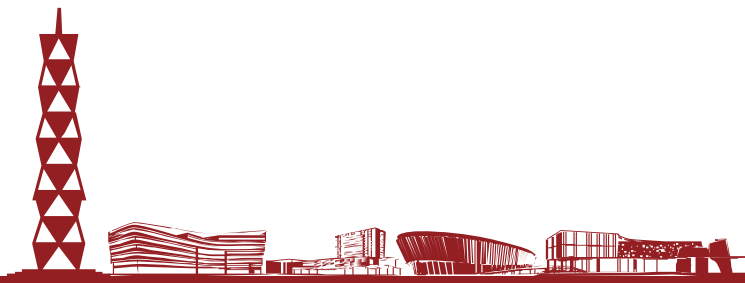
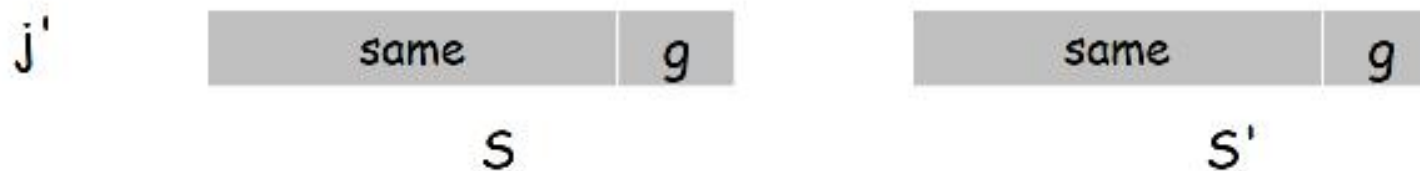
# Farthest-In-Future: Analysis



- Pf. (continued)
- Let  $j'$  be the first time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$



- Case 3b:  $g \neq e, f$ .  $S$  must evict  $e$ .
- Make  $S'$  evict  $f$ ; now  $S$  and  $S'$  have the same cache.

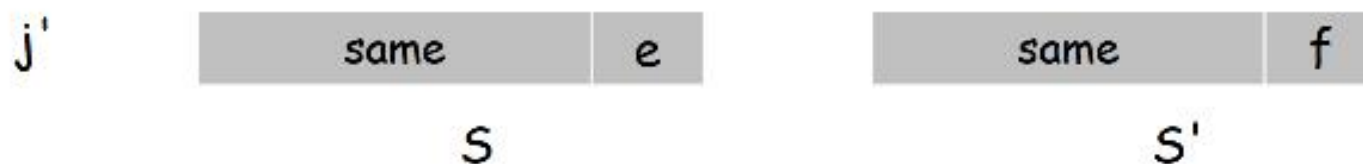




# Farthest-In-Future: Analysis



- Pf. (continued)
- Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$



- Case 3c:  $g = f$ . Element  $f$  cannot be in cache of  $S$ , so let  $e'$  be the element that  $S$  evicts.
  - If  $e' = e$ ,  $S'$  accesses  $f$  from cache; now  $S$  and  $S'$  have same cache
  - If  $e' \neq e$ ,  $S'$  evicts  $e'$  and brings  $e$  into the cache; now  $S$  and  $S'$  have the same cache.  $S'$  is no longer reduced, but can be transformed into a reduced schedule with
    - a) agrees with  $S_{FF}$  through step  $j + 1$
    - b) has no more evictions than  $S$





# Farthest-In-Future: Analysis



- **Theorem.** FF is optimal eviction algorithm
- **Pf.** (by induction on number of requests  $j$ )
- Base case (trivial):
- There exists an optimal reduced schedule  $S$  that makes the same schedule as  $S_{FF}$  through the first **0** requests
- Inductive step (implied by the lemma):
- If there exists an optimal reduced schedule  $S$  that agrees with  $S_{FF}$  through the first  **$j$**  requests, then there exists an optimal reduced schedule  $S'$  that agrees with  $S_{FF}$  through the first  **$j+1$**  requests





# Caching Perspective



- **Online vs. offline algorithms**
  - Offline: full sequence of requests is known as priori
  - Online (reality): requests are not known in advance
  - Caching is among most fundamental online problems in CS
- **LIFO.** Evict page brought in most recently
- **LRU.** Evict page whose most recent access was earliest
- **Theorem.** FF is optimal offline eviction algorithm.
  - Provides basis for understanding and analyzing online algorithms
  - LRU is  $k$ -competitive.
  - LIFO is arbitrarily bad





# Greedy Algorithms: Summary



- **Basic idea**

- Make the locally optimal choice at each step

- **Algorithms**

- Interval Scheduling
  - Choose the job with the earliest finish time
- Scheduling to Minimize Lateness
  - Choose the job with the earliest deadline
- Clustering
  - Single-link k-clustering
- Optimal Caching
  - Evict item that is requested farthest in future

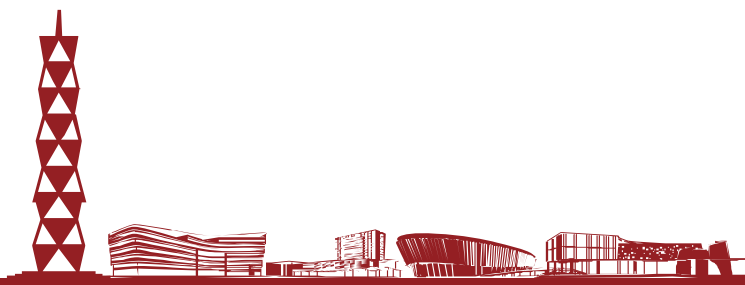
- **Proof skills**

- **Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's
- **Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality





# Divide and Conquer





# Divide-and-Conquer



- **Divide-and-conquer**

- Break up problem into several parts
- Solve each part recursively
- Combine solutions to sub-problems into overall solution

- **Most common usage**

Break up problem of size  $n$  into **two** equal parts of size  $\frac{1}{2}n$  in **linear time**

Solve two parts recursively

Combine two solutions into overall solution in **linear time**

- **Consequence**

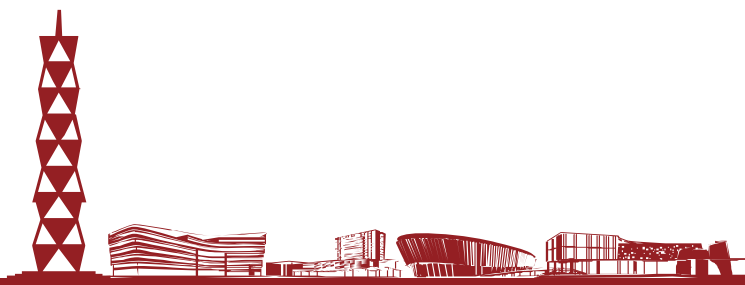
Divide-and-conquer:  $\Theta(n \log n)$







# Mergesort (Revisit)





# Mergesort



- **Mergesort**

- Divide array into two halves
- Recursively sort each half
- Merge two halves to make sorted whole

A	L	G	O	R	I	T	H	M	S
---	---	---	---	---	---	---	---	---	---

A	L	G	O	R
---	---	---	---	---

I	T	H	M	S
---	---	---	---	---

divide

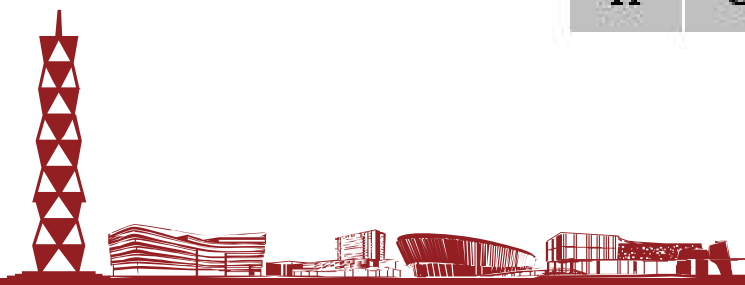
A	G	L	O	R
---	---	---	---	---

H	I	M	S	T
---	---	---	---	---

sort

A	G	H	I	L	M	O	R	S	T
---	---	---	---	---	---	---	---	---	---

merge

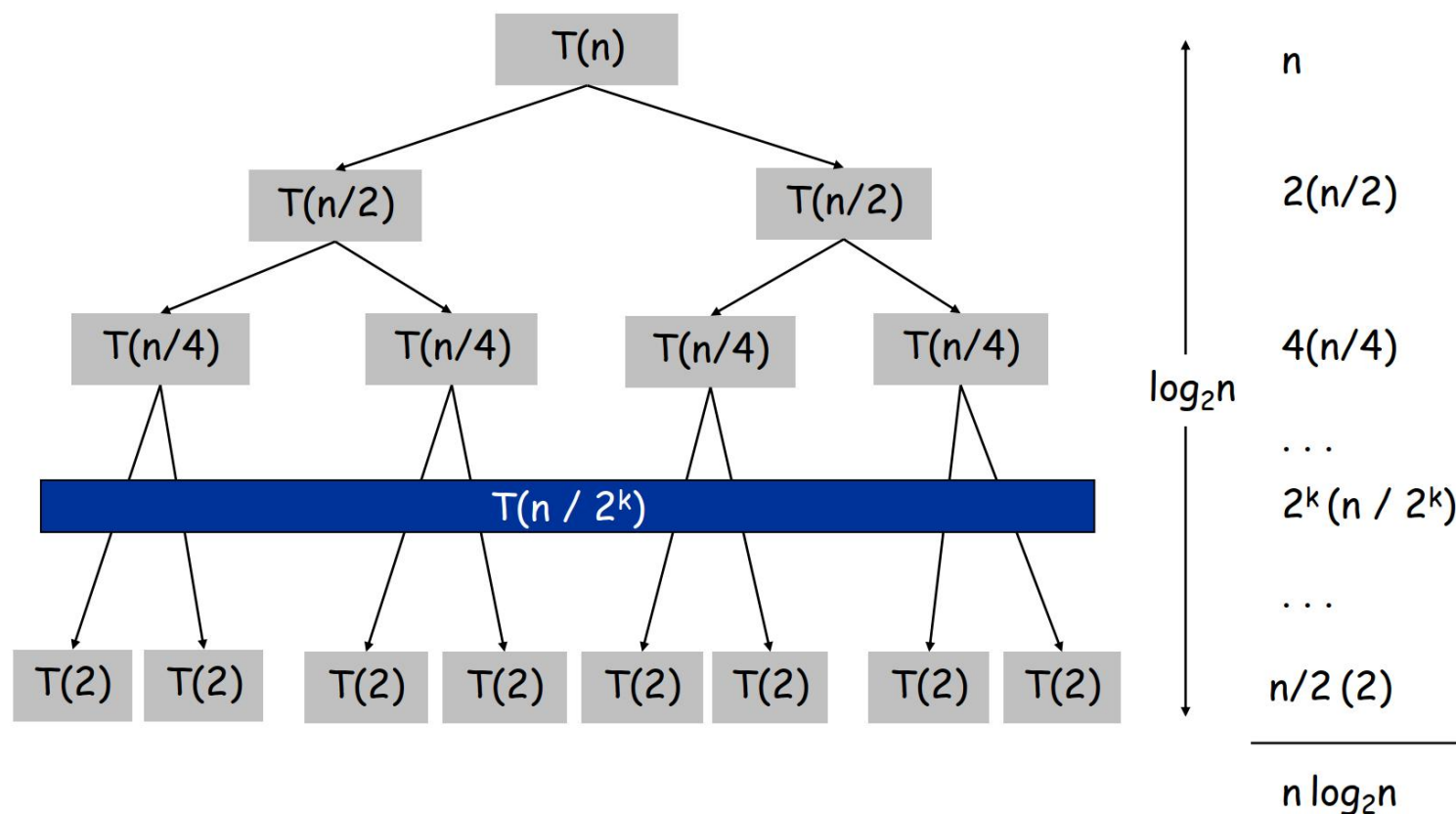




# Proof by Recursive Tree



$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



# Proof by Induction



- **Claim.** If  $T(n)$  satisfies this recurrence, then  $T(n) = n \log_2 n$

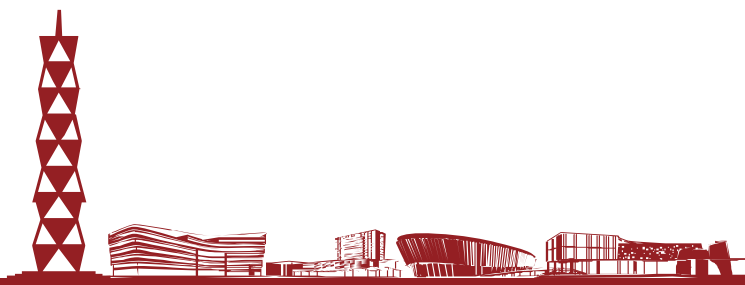


Assumes  $n$  is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

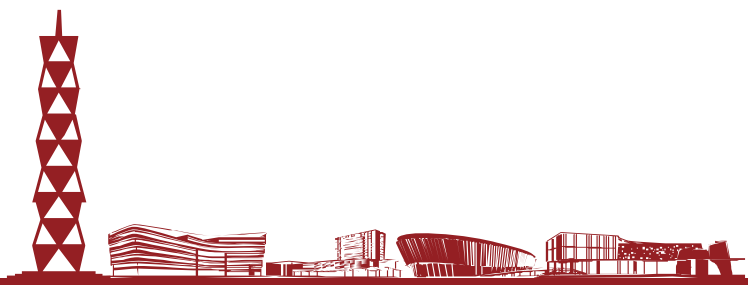
- **Pf.** (by induction on  $n$ )
  - Base case:  $n = 1$
  - Inductive hypothesis:  $T(n) = n \log_2 n$
  - Goal: show that  $T(2n) = 2n \log_2(2n)$

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n(\log_2(2n) - 1) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$





# Closest Pair of Points





# Closest Pair of Points



- **Closest pair.** Given  $n$  points in the plane, find a pair with smallest Euclidean distance between them
- **Fundamental geometric primitive**
  - Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  - Special case of nearest neighbor, Euclidean MST, Voronoi

Fast closest pair inspired fast algorithms for these problems

- **Brute force.** Check all pairs of points  $p$  and  $q$  with  $\Theta(n^2)$  comparisons
- **Assumption.** No two points have same  $x$  coordinate

To make presentation cleaner



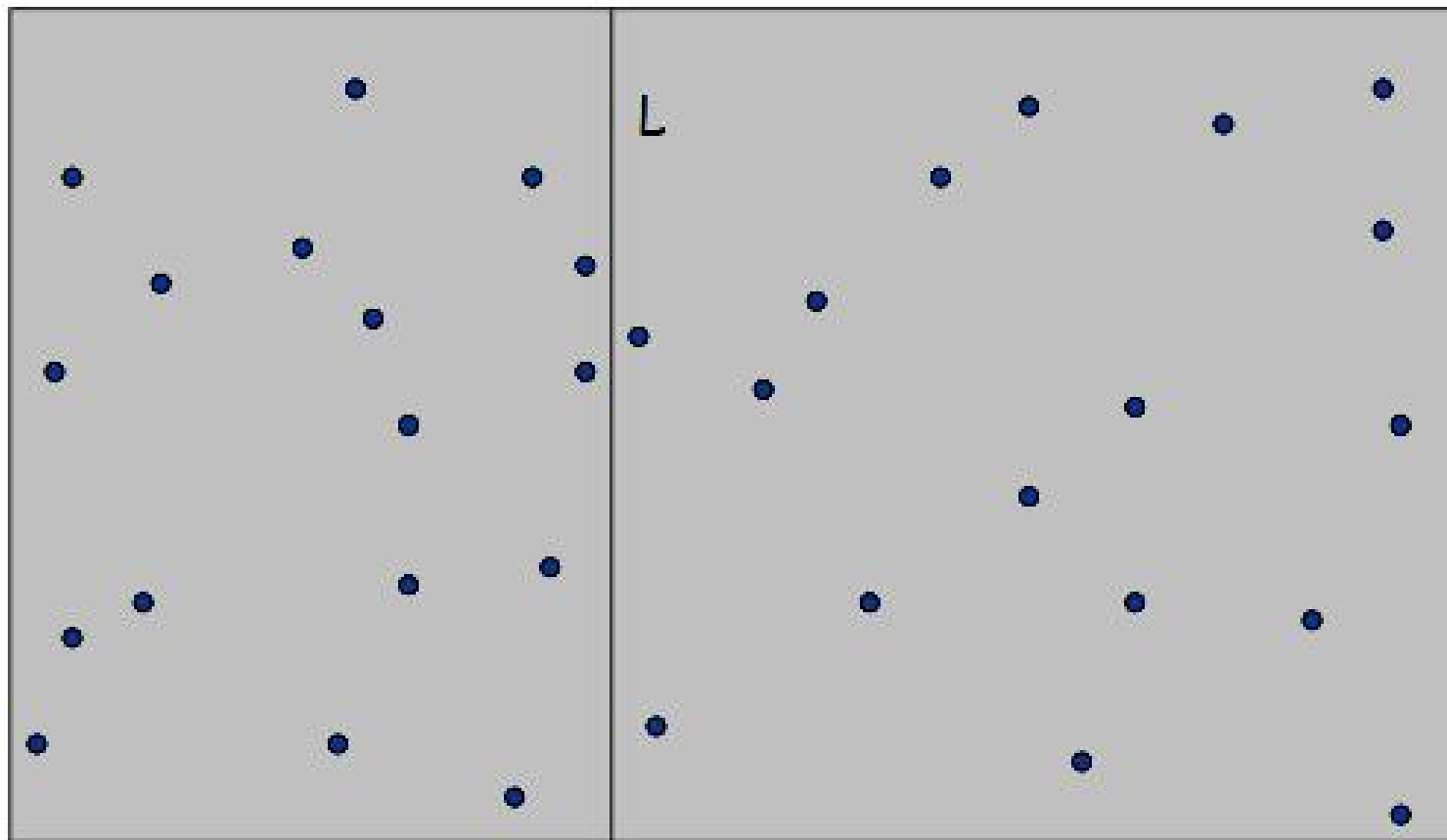


# Closest Pair of Points



- **Algorithm**

- **Divide:** draw vertical line  $L$  so that roughly  $\frac{1}{2}n$  points on each side



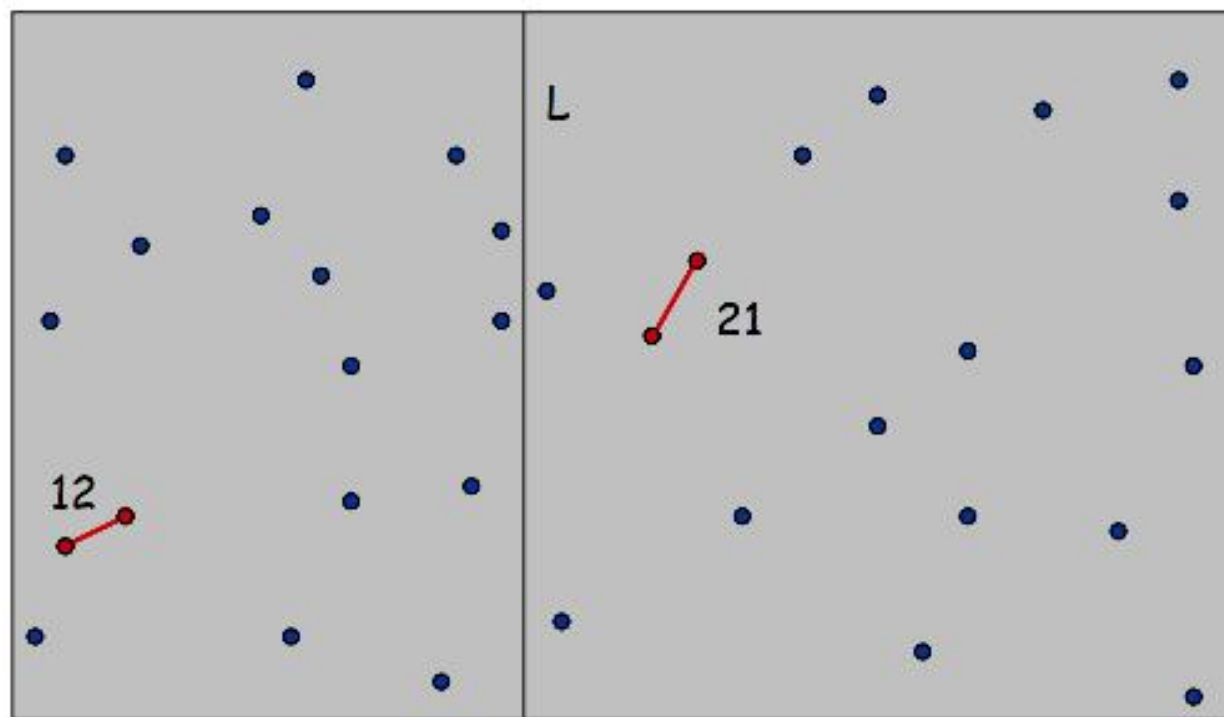


# Closest Pair of Points



- **Algorithm**

- Divide: draw vertical line  $L$  so that roughly  $1/2 n$  points on each side
- **Conquer:** find closest pair in each side recursively





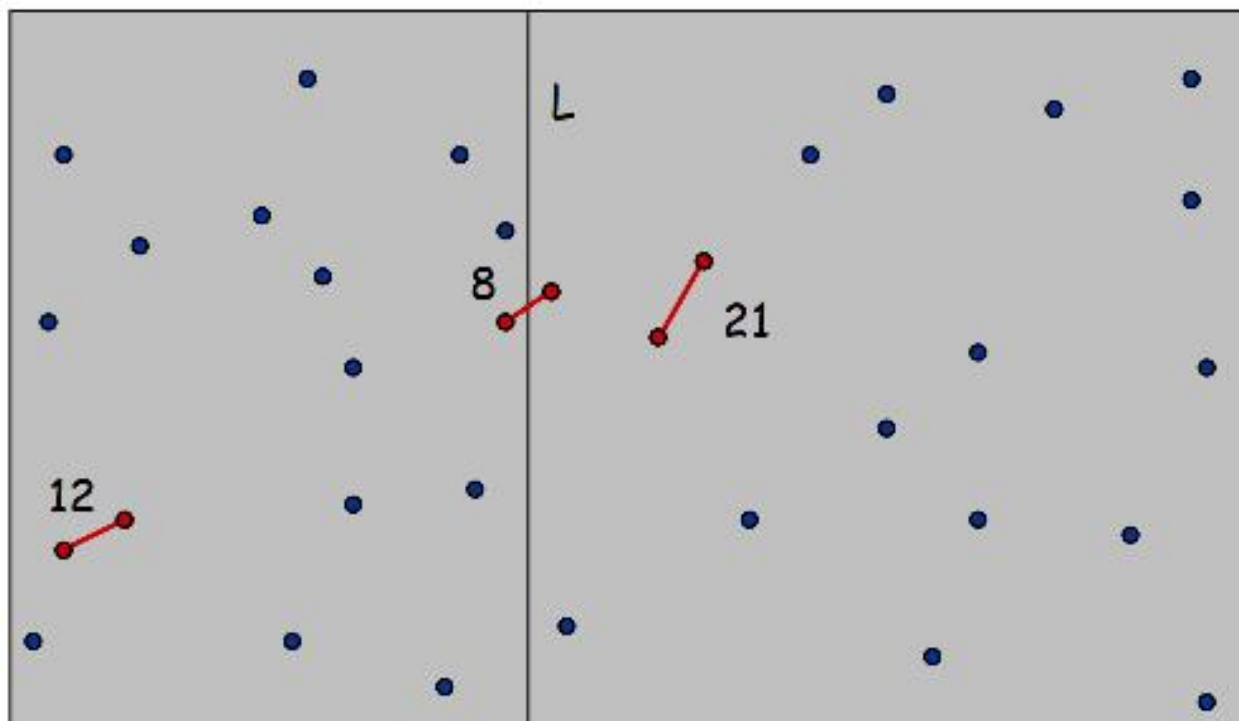


# Closest Pair of Points



- Algorithm

- Divide: draw vertical line  $L$  so that roughly  $\frac{1}{2}n$  points on each side
- Conquer: find closest pair in each side recursively
- **Combine:** find closest pair with one point in each side  $\leftarrow \Theta(n^2)$
- Return best of 3 solutions

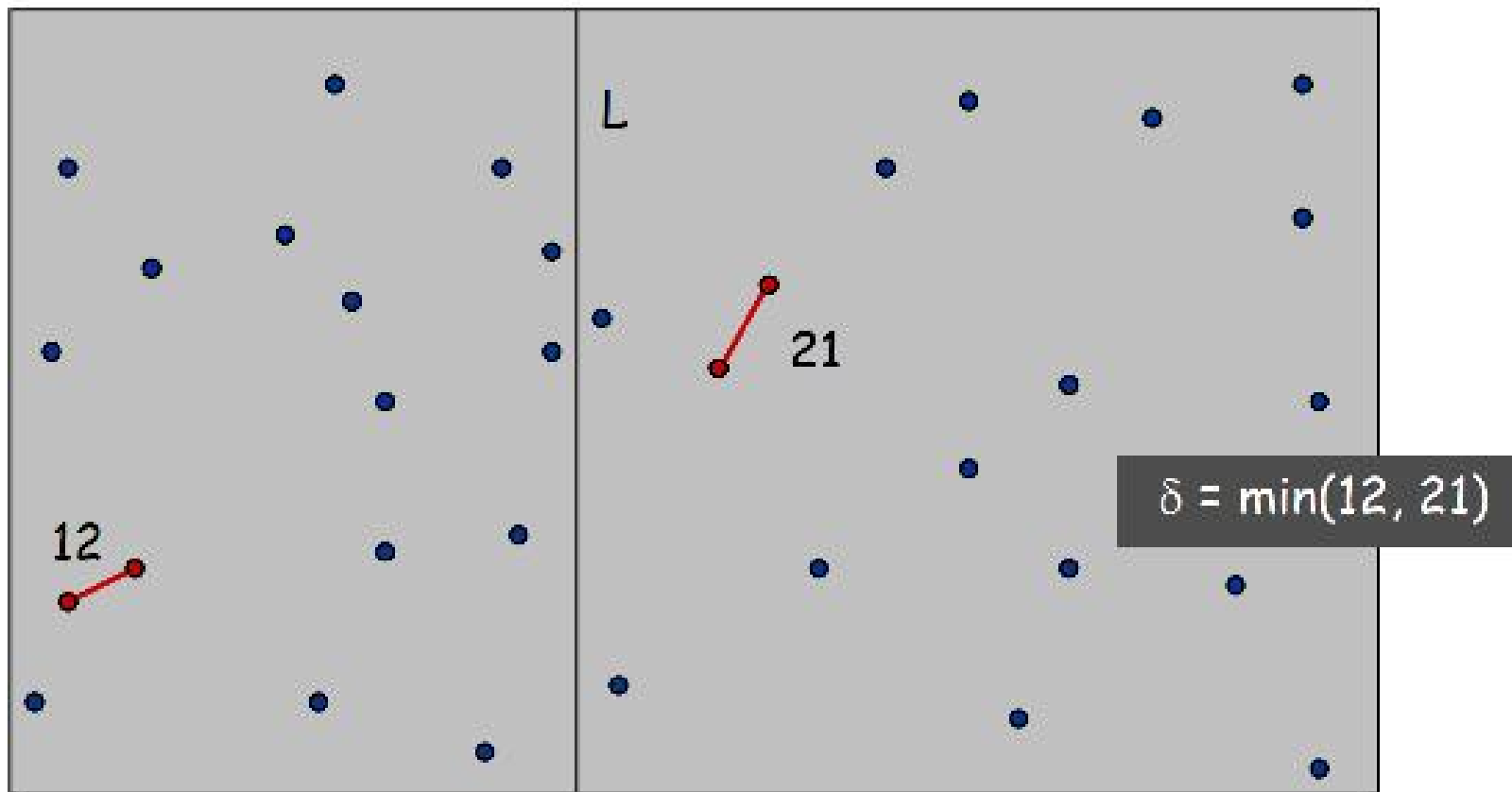




# Closest Pair of Points



- Find closest pair with one point in each side, **assuming that distance  $< \delta$**



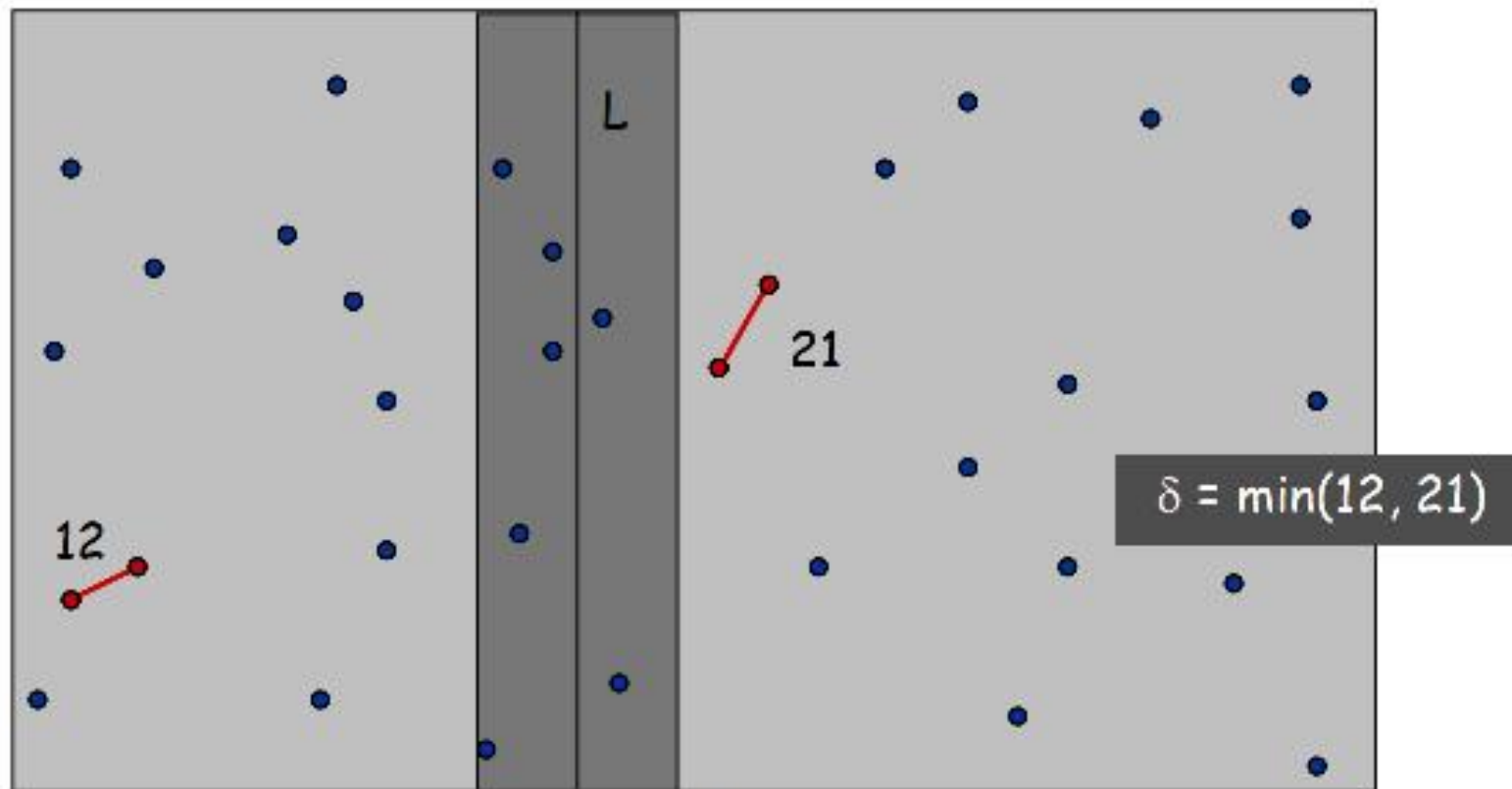


# Closest Pair of Points



- Find closest pair with one point in each side, **assuming that distance  $< \delta$**

**Observation: only need to consider points within  $\delta$  of line  $L$**





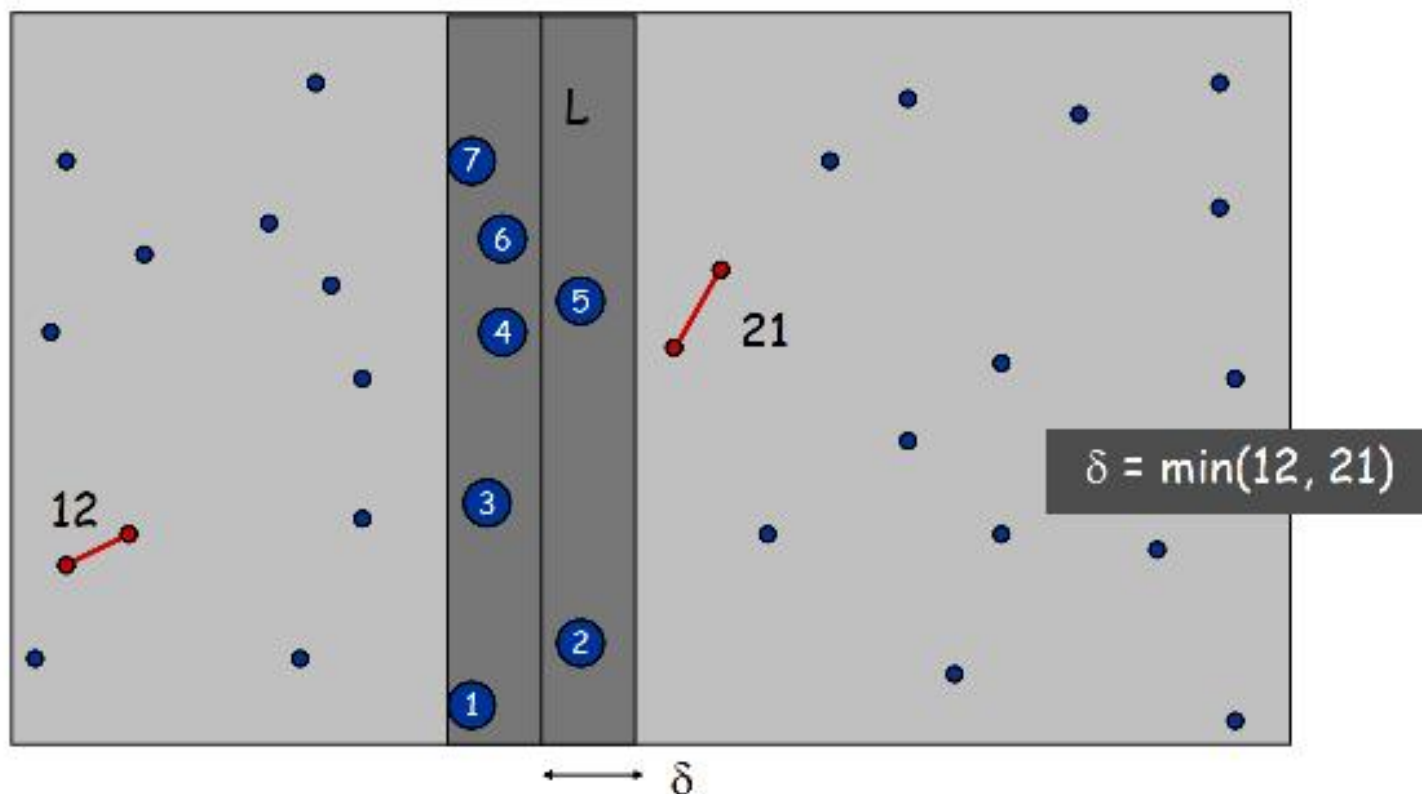
# Closest Pair of Points



- Find closest pair with one point in each side, **assuming that distance  $< \delta$**

**Observation: only need to consider points within  $\delta$  of line L**

**Sort points in  $2\delta$ -strip by their y coordinate**





# Closest Pair of Points

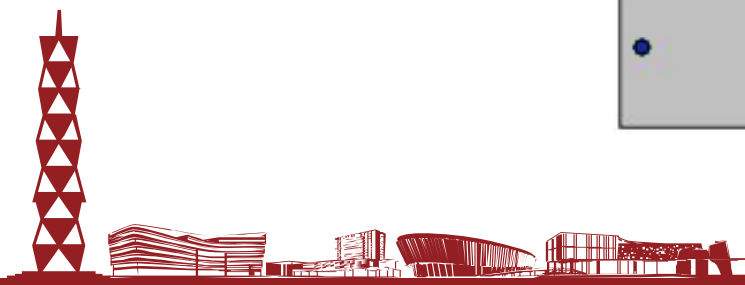
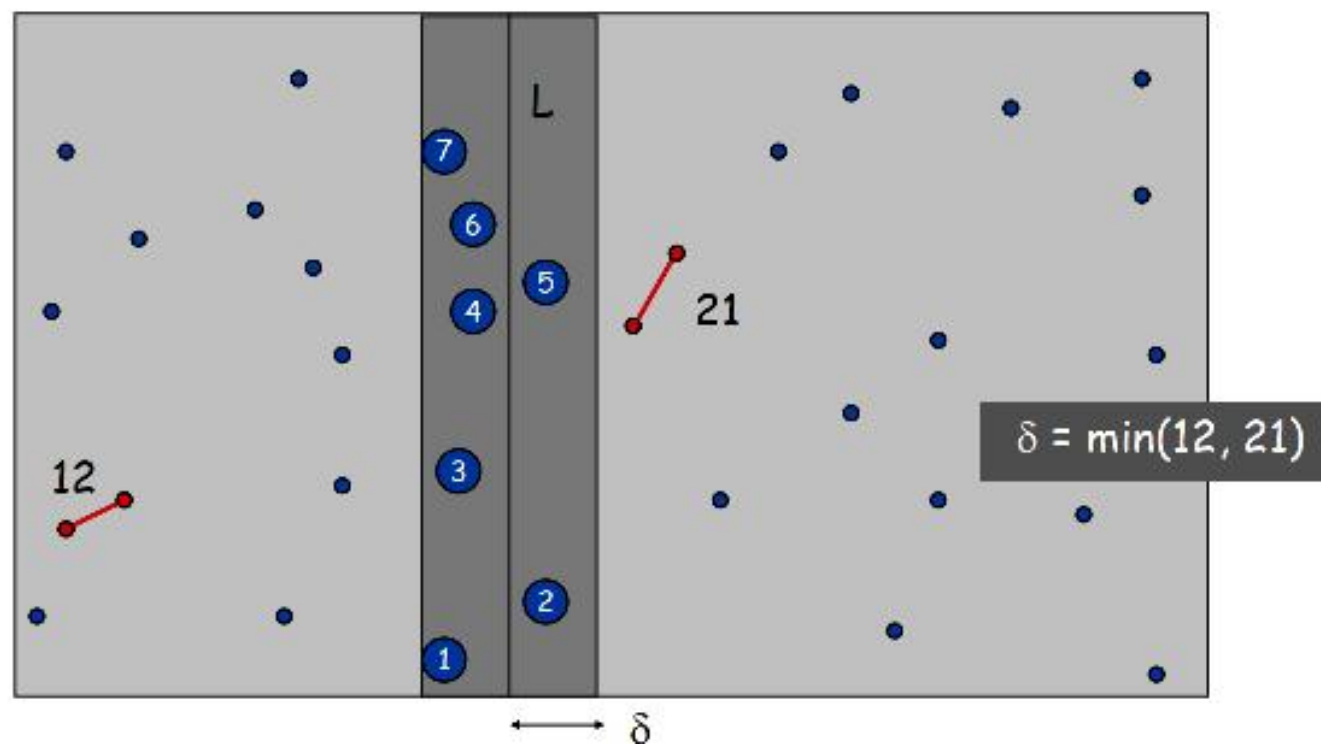


- Find closest pair with one point in each side, **assuming that distance  $< \delta$**

**Observation: only need to consider points within  $\delta$  of line L**

**Sort points in  $2\delta$ -strip by their y coordinate**

**Only check distances of those within 11 positions in sorted list!**

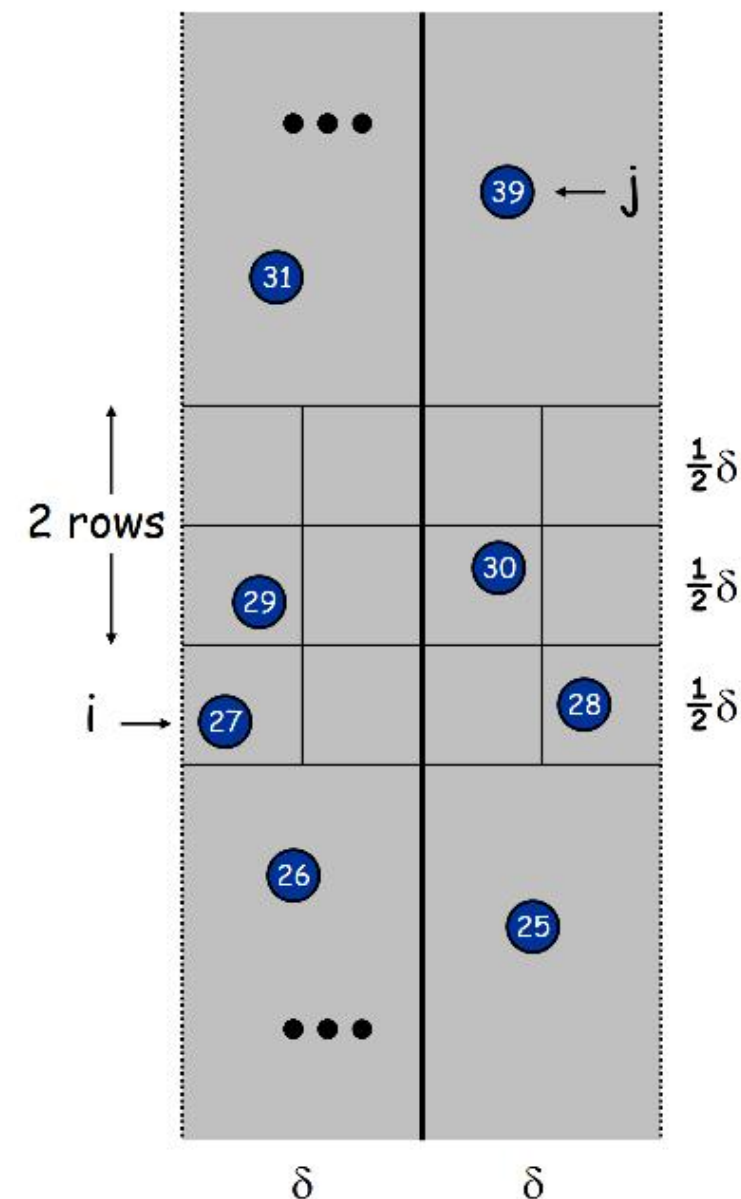




# Closest Pair of Points



- **Def.** Let  $s_i$  be the point in the  $2\delta$ -strip, with the  $i^{\text{th}}$  smallest  $y$ -coordinate
- **Claim.** If  $|i - j| \geq 12$ , then the distance between  $s_i$  and  $s_j$  is at least  $\delta$
- **Pf.**
  - No two points lie in some  $\frac{1}{2}\delta$ -by- $\frac{1}{2}\delta$  box
  - Two points at least 2 rows apart have distance  $\geq 2(\frac{1}{2}\delta)$

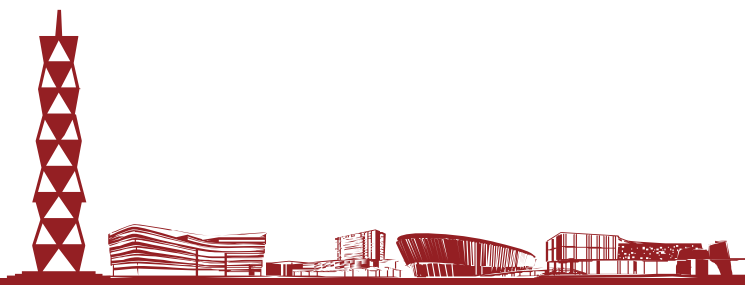




# Closest Pair of Points



- Find closest pair with one point in each side, **assuming that distance  $< \delta$** 
  - Linear time algorithm!
- Without this assumption?
  - Run the same algorithm
  - If assumption true ,we will find the right closest pair with one point in each side
  - If false, the algorithm will find a pair with distance  $\geq \delta$  and then the combine step will correctly return  $\delta$  as distance of closest pair





# Closest Pair Algorithm



```
Closest-Pair( $p_1, \dots, p_n$ ) {  
    Compute separation line  $L$  such that half the points  
    are on one side and half on the other side.  $O(n \log n)$   
  
     $\delta_1 = \text{Closest-Pair}(\text{left half})$   
     $\delta_2 = \text{Closest-Pair}(\text{right half})$   $2T(n / 2)$   
     $\delta = \min(\delta_1, \delta_2)$   
  
    Delete all points further than  $\delta$  from separation line  $L$   $O(n)$   
  
    Sort remaining points by y-coordinate.  $O(n \log n)$   
  
    Scan points in y-order and compare distance between  
    each point and next 11 neighbors. If any of these  
    distances is less than  $\delta$ , update  $\delta$ .  $O(n)$   
  
    return  $\delta$ .  
}
```







# Closest Pair of Points: Analysis



- Running time

$$T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$$

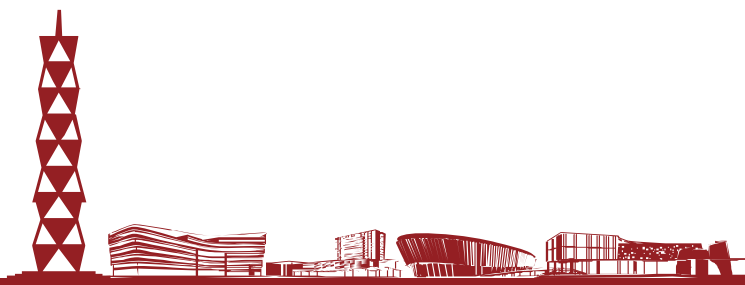
- Q. Can we achieve  $O(n \log n)$ ?
- A. Yes. Don't sort points from scratch each time.
  - Sort all the points twice before recursive call, once by x coordinate and once by y coordinate
  - Reuse the sorted sequences when needed (linear time)

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$





# Integer Multiplication (Revisit)



- [illegible]

Add



# Divide-and-Conquer Multiplication: Warmup



- To multiply two  $n$ -digit integers:
  - Multiply four  $\frac{1}{2}n$  - digit integers
  - Add two  $\frac{1}{2}n$ -digit integers, and shift to obtain result

$$x = \underbrace{1000}_{x_1} \underbrace{1101}_{x_0}$$

$$\begin{aligned}x &= 2^{n/2} \cdot x_1 + x_0 \\y &= 2^{n/2} \cdot y_1 + y_0 \\xy &= (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0\end{aligned}$$

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$



assumes  $n$  is a power of 2





# Karatsuba Multiplication



- To multiply two  $n$ -digit integers:
  - Add two  $\frac{1}{2}n$  digit integers
  - Multiply **three**  $\frac{1}{2}n$ -digit integers
  - Add, subtract, and shift  $\frac{1}{2}n$ -digit integers to obtain results

$$\begin{aligned}x &= 2^{n/2} \cdot x_1 + x_0 \\y &= 2^{n/2} \cdot y_1 + y_0 \\xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\&= 2^n \cdot \underbrace{x_1 y_1}_A + 2^{n/2} \cdot \underbrace{((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0)}_{B \quad A \quad C \quad C} + x_0 y_0\end{aligned}$$

- **Theorem.** Can multiply two  $n$ -digit integers in  $O(n^{1.585})$  bit operations

$$\begin{aligned}T(n) &\leq \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \\&\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})\end{aligned}$$



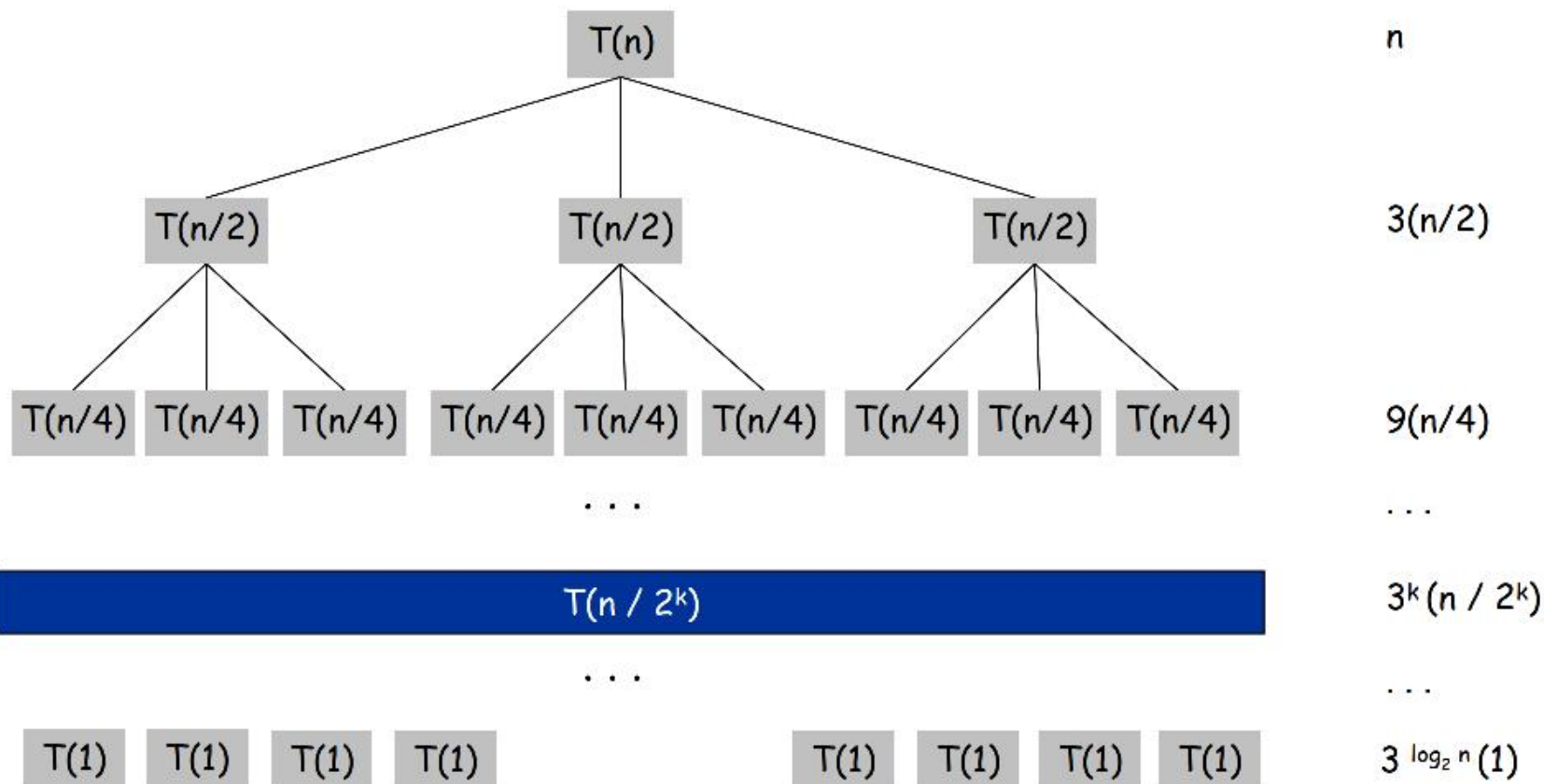


# Karatsuba: Recursion Tree



$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n \left(\frac{3}{2}\right)^k = \frac{\left(\frac{3}{2}\right)^{1+\log_2 n} - 1}{\frac{3}{2} - 1} n = 3n^{\log_2 3} - 2n$$





# Next Time: Divide and Conquer (Cont.)

