CS240 Algorithm Design and Analysis

Lecture 3

Greedy Algorithms (Cont.) Divide and Conquer

Quan Li
Fall 2023
2023.10.10

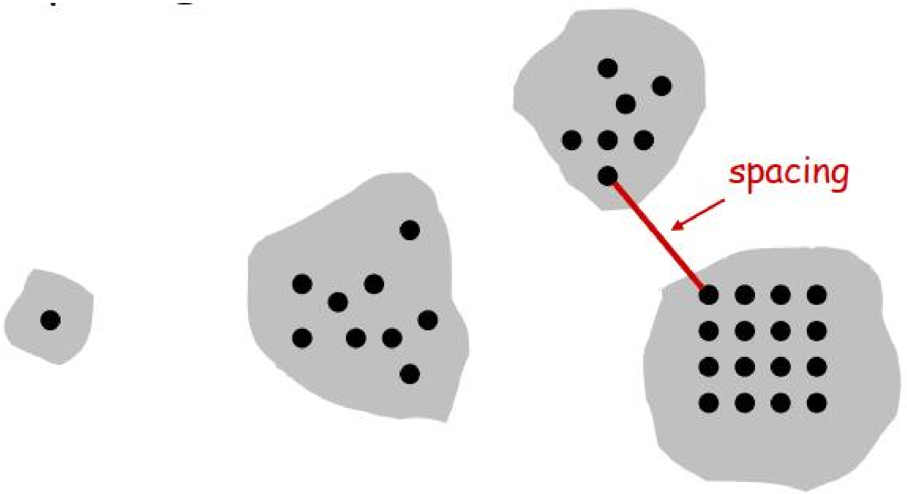
Last Time – What you need to know

- 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
    - Single-link k-clustering: precisely Kruskal's algorithm (except **we stop when there are k connected components**)

立志成才 报国裕民

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)
  - $d(p_i, p_j) >= 0$ (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



k = 4

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

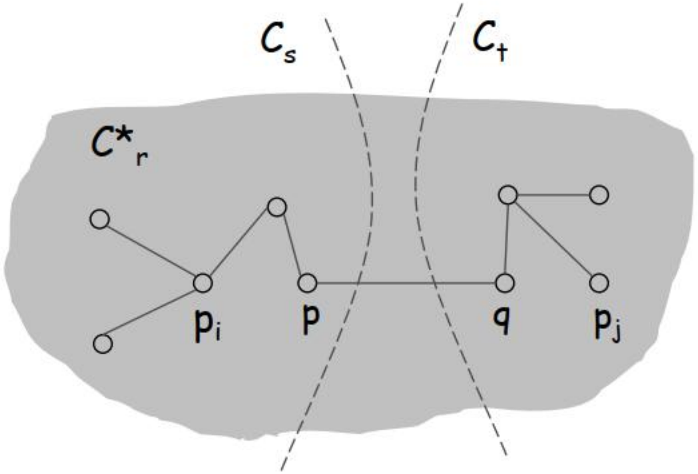
- **Key observation.** This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components)

- **Remark.** Equivalent to finding an MST and deleting the k-1 most expensive edges

Greedy Clustering Algorithm: Analysis

- **Theorem.** Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ 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 $C_1, \ldots, C_k$
  - The spacing of C^* is the length d^* of the (k-1)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 --- p_j path in C^*_r spans two different clusters in C.
  - All edges on p_i --- p_j path have length <= d^* since Kruskal chose them
  - Spacing of C is <= d^* since p and q are in different clusters

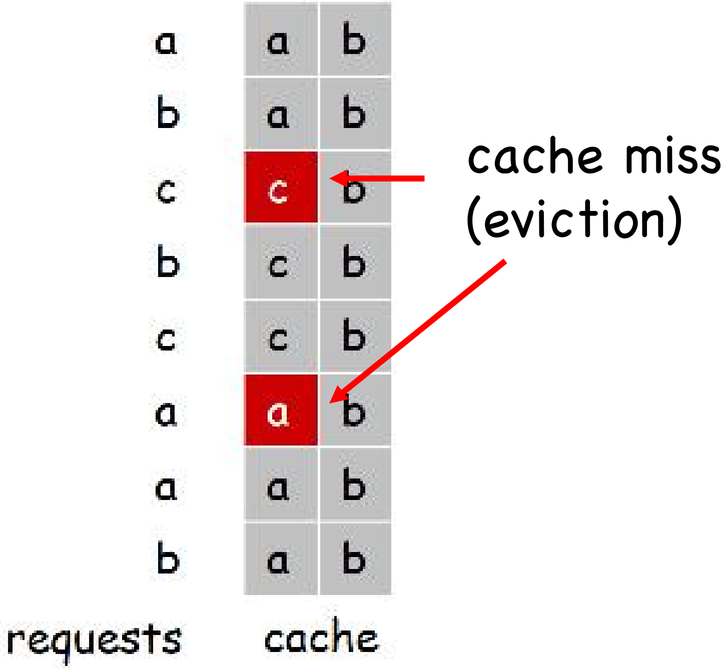


Optimal Caching

Optimal Offline Caching

- **Caching**
  - Cache with capacity to store k items
  - Sequence of m item requests $d_1, d_2, \ldots, d_m$
  - **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

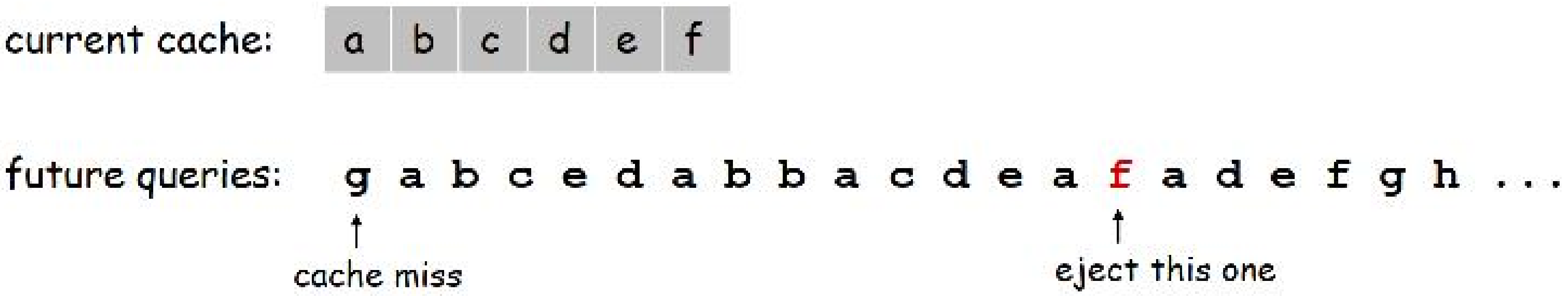


requests	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

cache miss (eviction)

Optimal Offline Caching: Farthest-In-Future (clairvoyant algorithm)

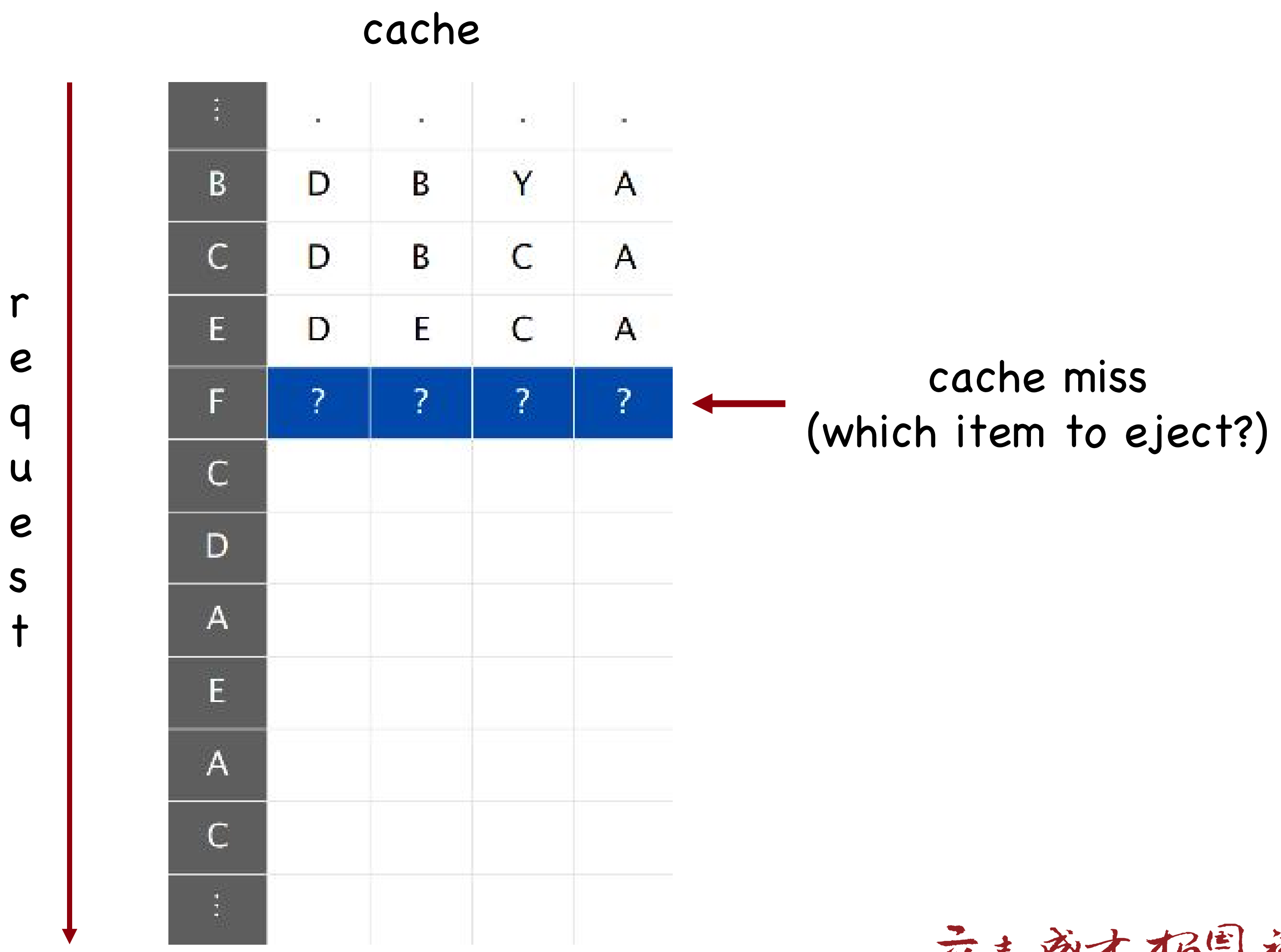
- **Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future



- **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

request

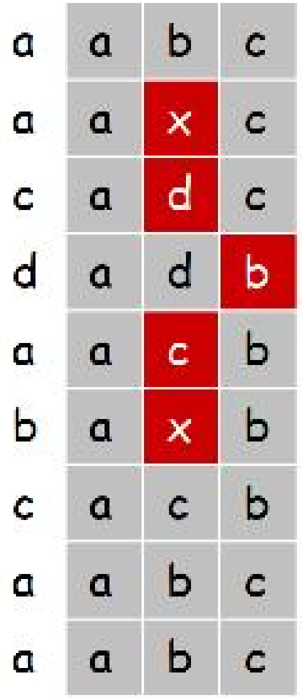
⋮	.	.	.	.
B	D	B	Y	A
C	D	B	C	A
E	D	E	C	A
F	?	?	?	?
C				
D				
A				
E				
A				
C				
⋮				

cache miss
(which item to eject?)

Reduced Eviction Schedules

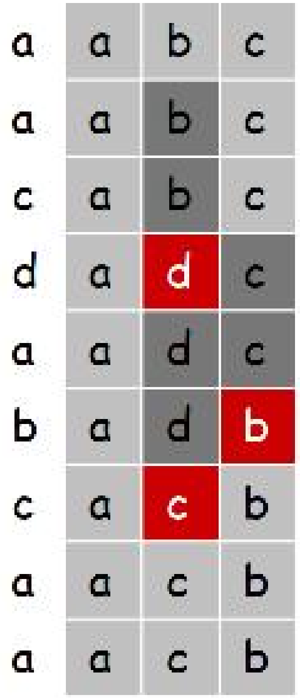
- **Def.** A **reduced** schedule is a schedule that only inserts an item d into the cache in a step in which d is requested and d is not already in the cache
- **Intuition.** Can transform an unreduced schedule into a reduced one with no more evictions

- 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

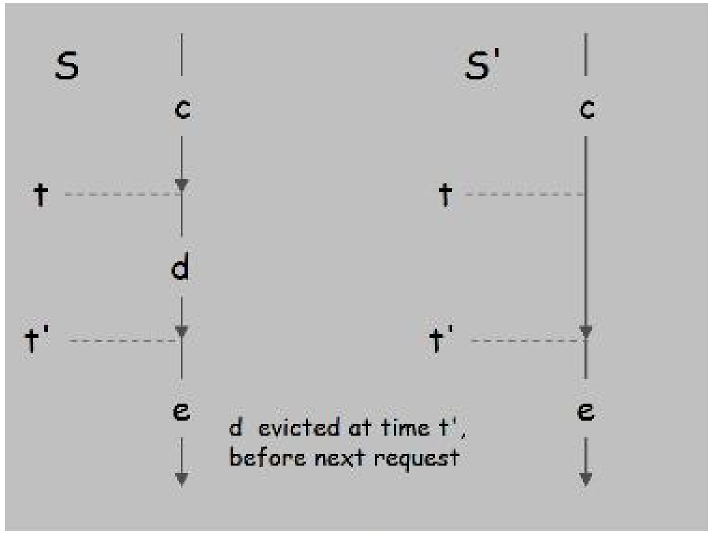
a reduced schedule

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**

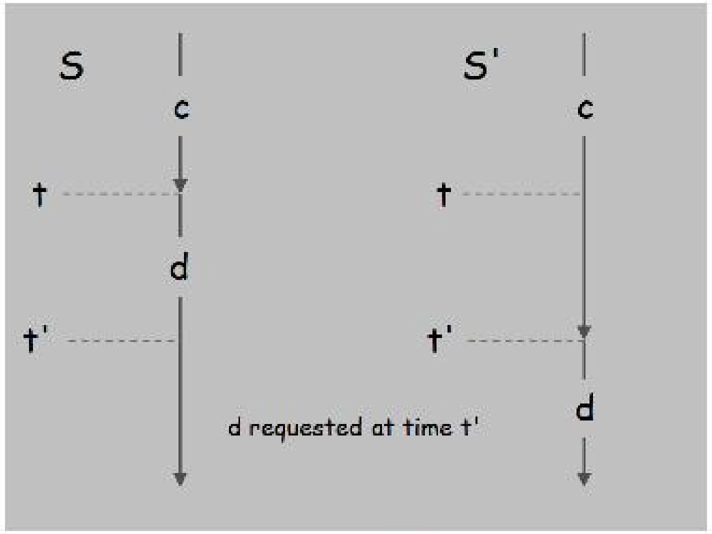


d evicted at time t', before next request

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

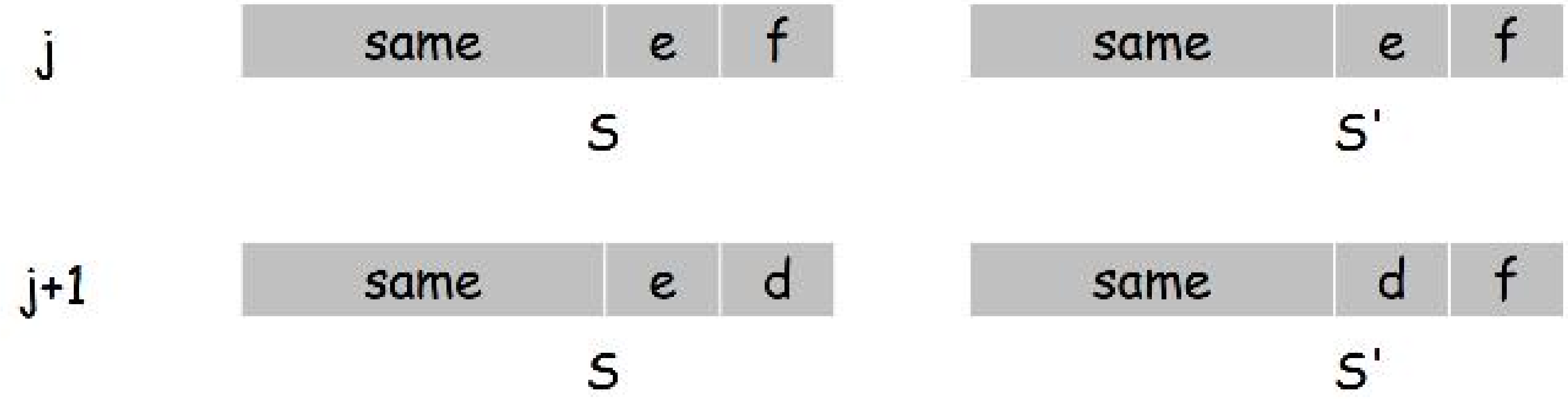
Farthest-In-Future: Analysis

- **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 ≠ 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 ≠ 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′

j'

same | e

S

same | f

S'

- 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′ ≠ 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 ½ n in **linear time**

  Solve two parts recursively

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

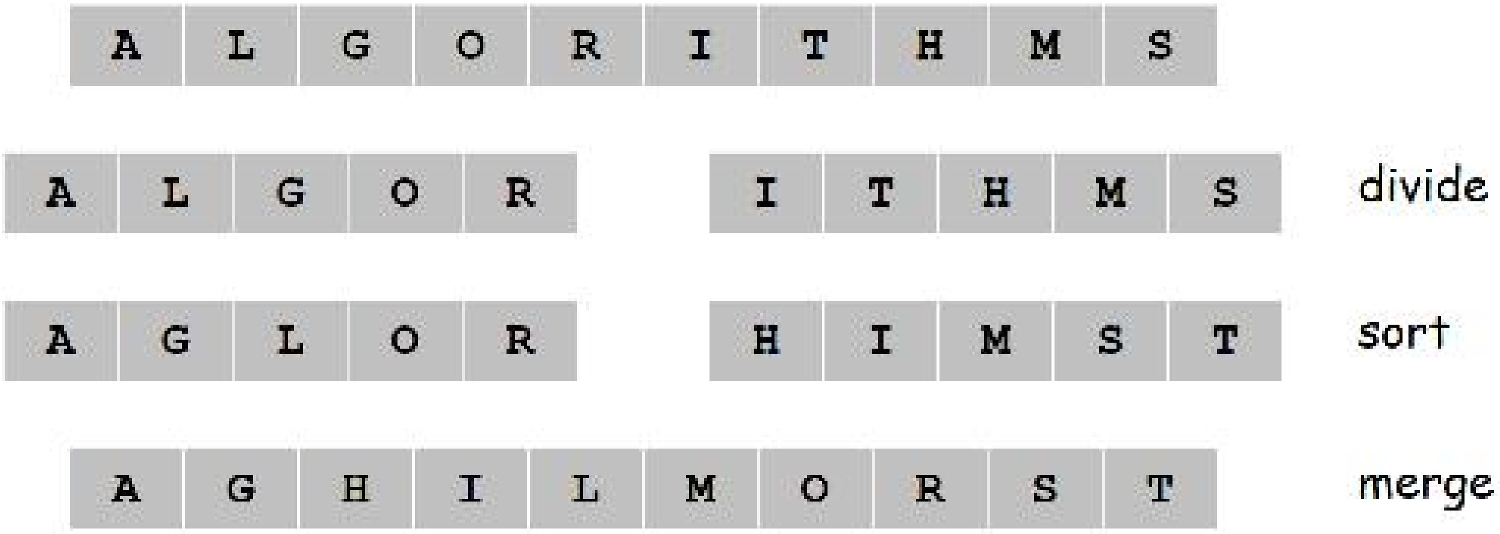
- **Consequence**

  Divide-and-conquer: Θ(nlogn)

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

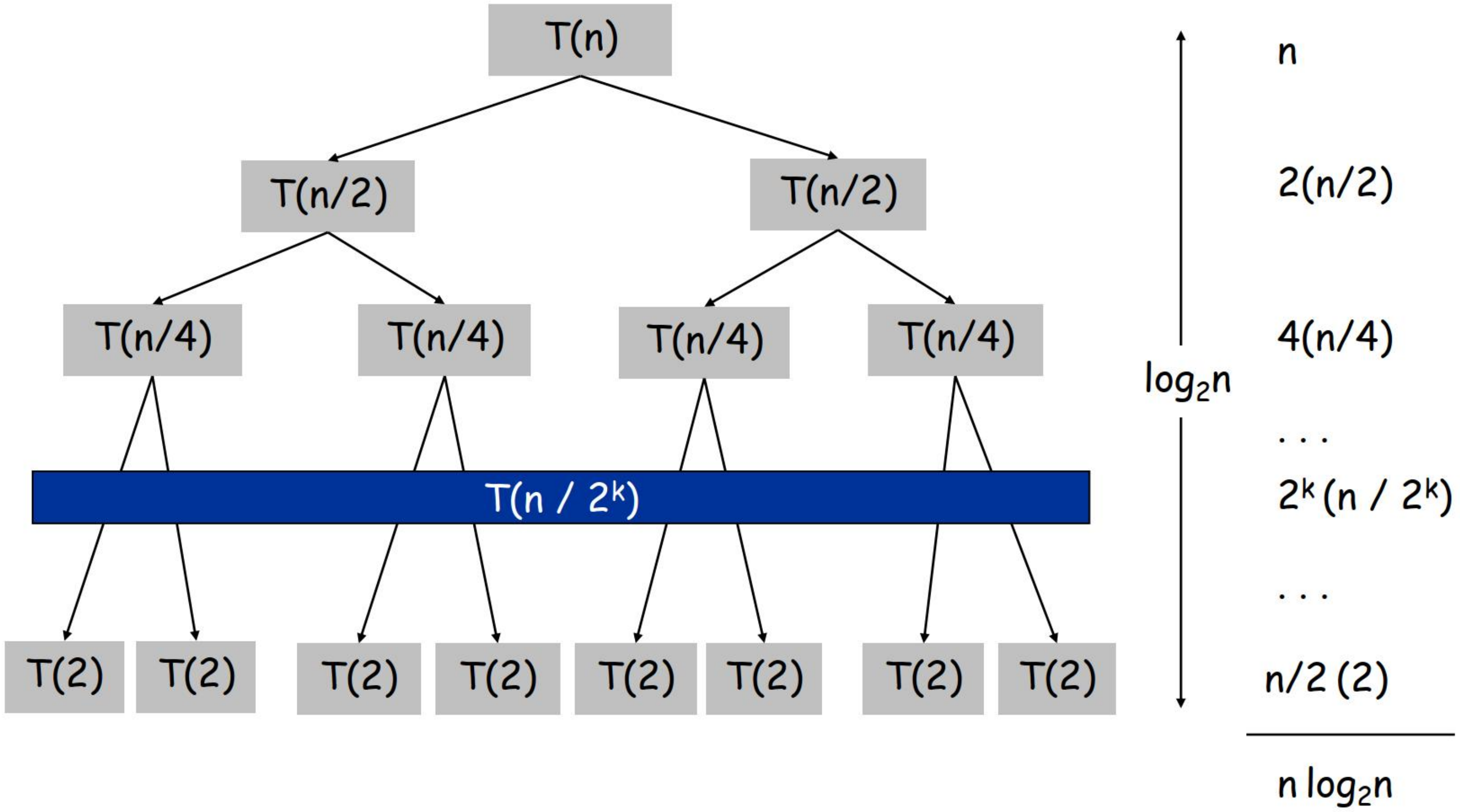
A L G O R　I T H M S　divide

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) = $n\log_2 n$

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

Assumes n is a power of 2

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

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

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

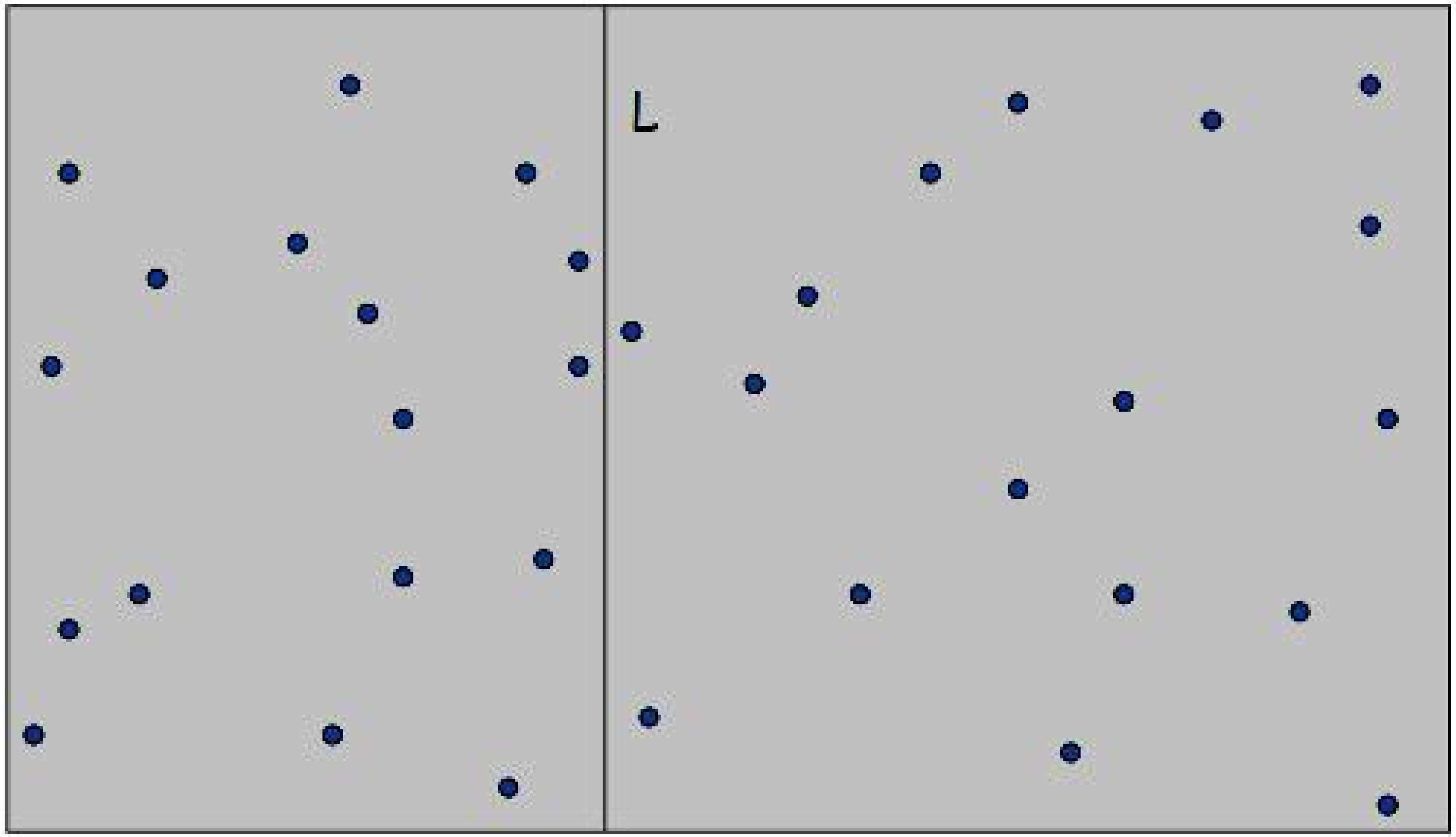
Fast closest pair inspired fast algorithms for these problems

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

To make presentation cleaner

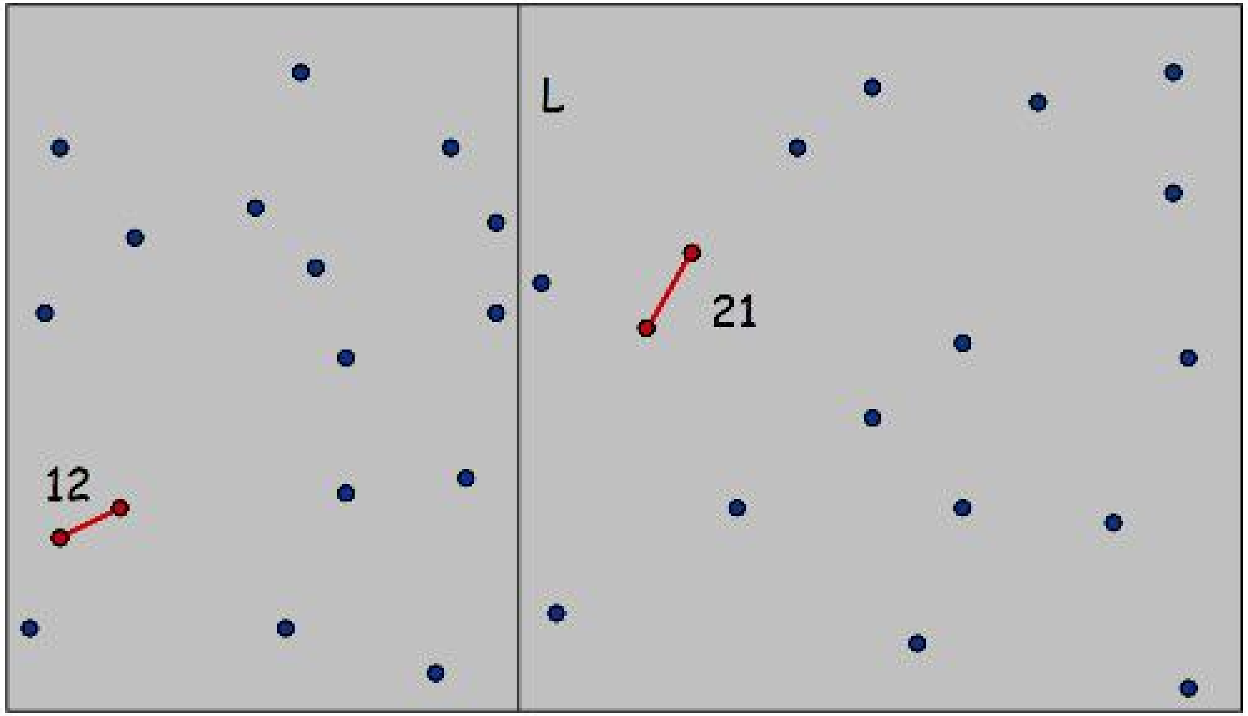
Closest Pair of Points

- **Algorithm**
  - **Divide:** draw vertical line L so that roughly ½ n points on each side



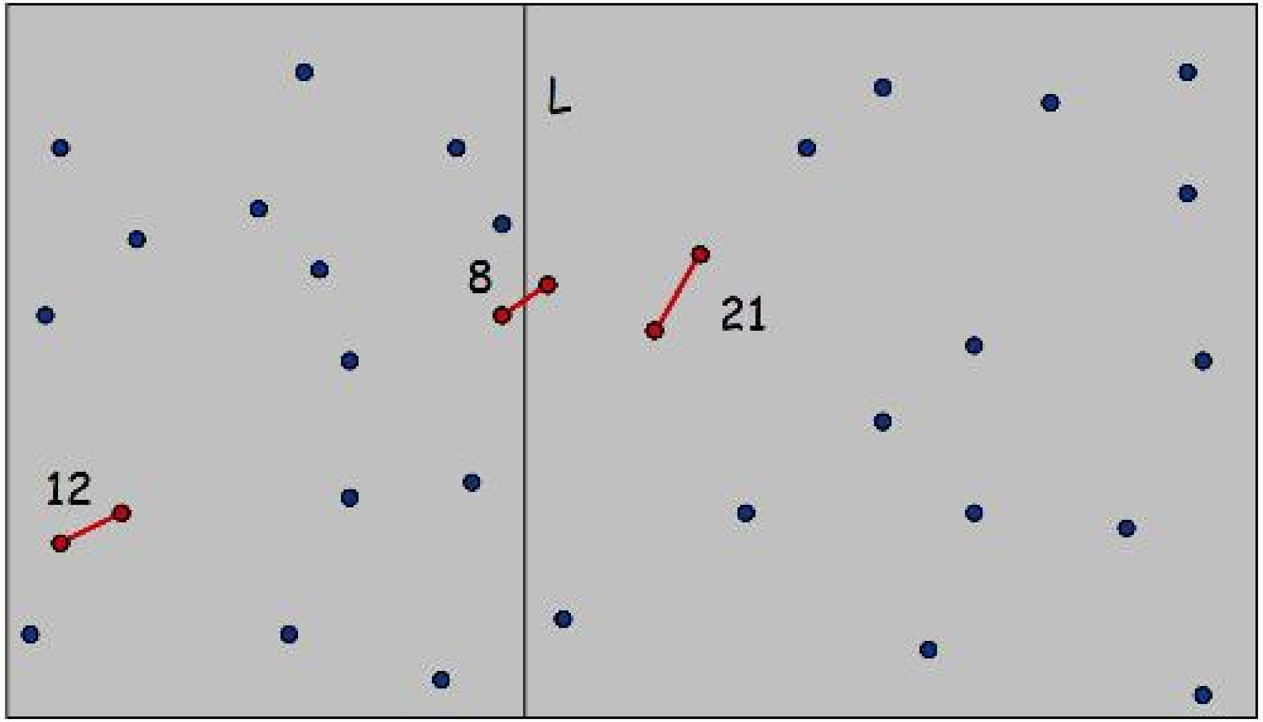
Closest Pair of Points

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



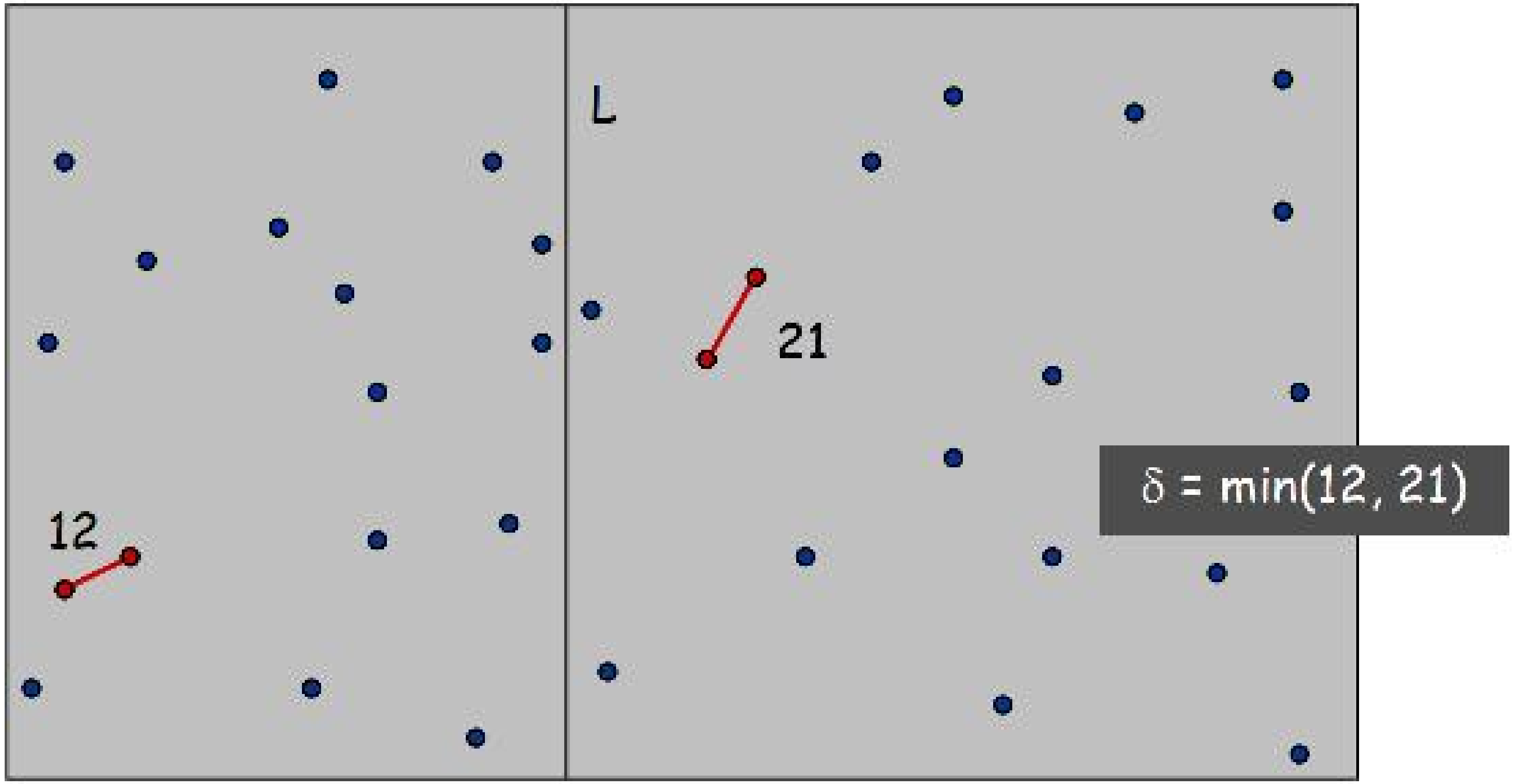
Closest Pair of Points

- Algorithm
  - Divide: draw vertical line L so that roughly ½n points on each side
  - Conquer: find closest pair in each side recursively
  - **Combine:** find closest pair with one point in each side ← Θ(n^2)
  - Return best of 3 solutions



Closest Pair of Points

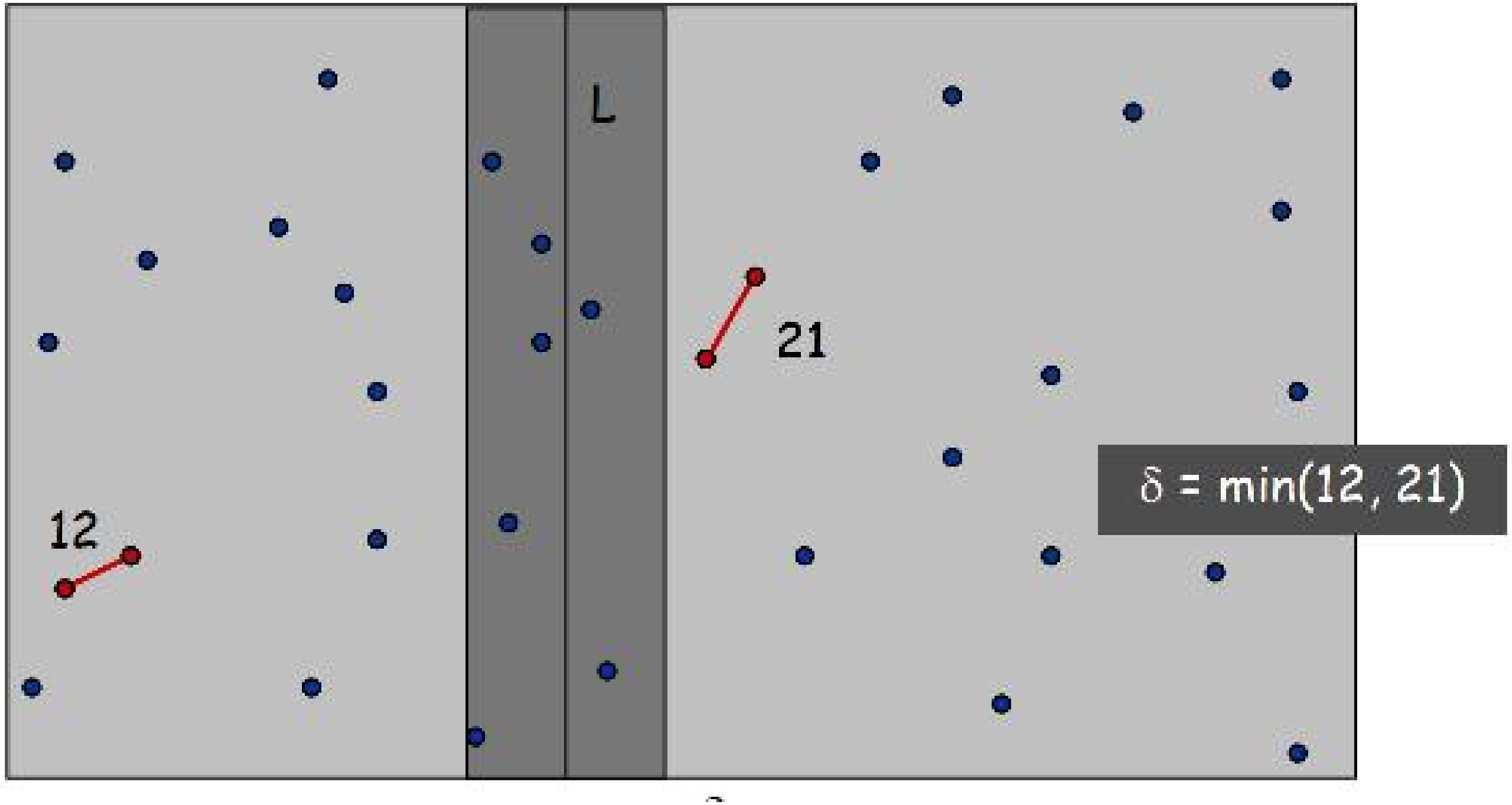
- Find closest pair with one point in each side, **assuming that distance < δ**



Closest Pair of Points

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

  Observation: only need to consider points within δ of line L

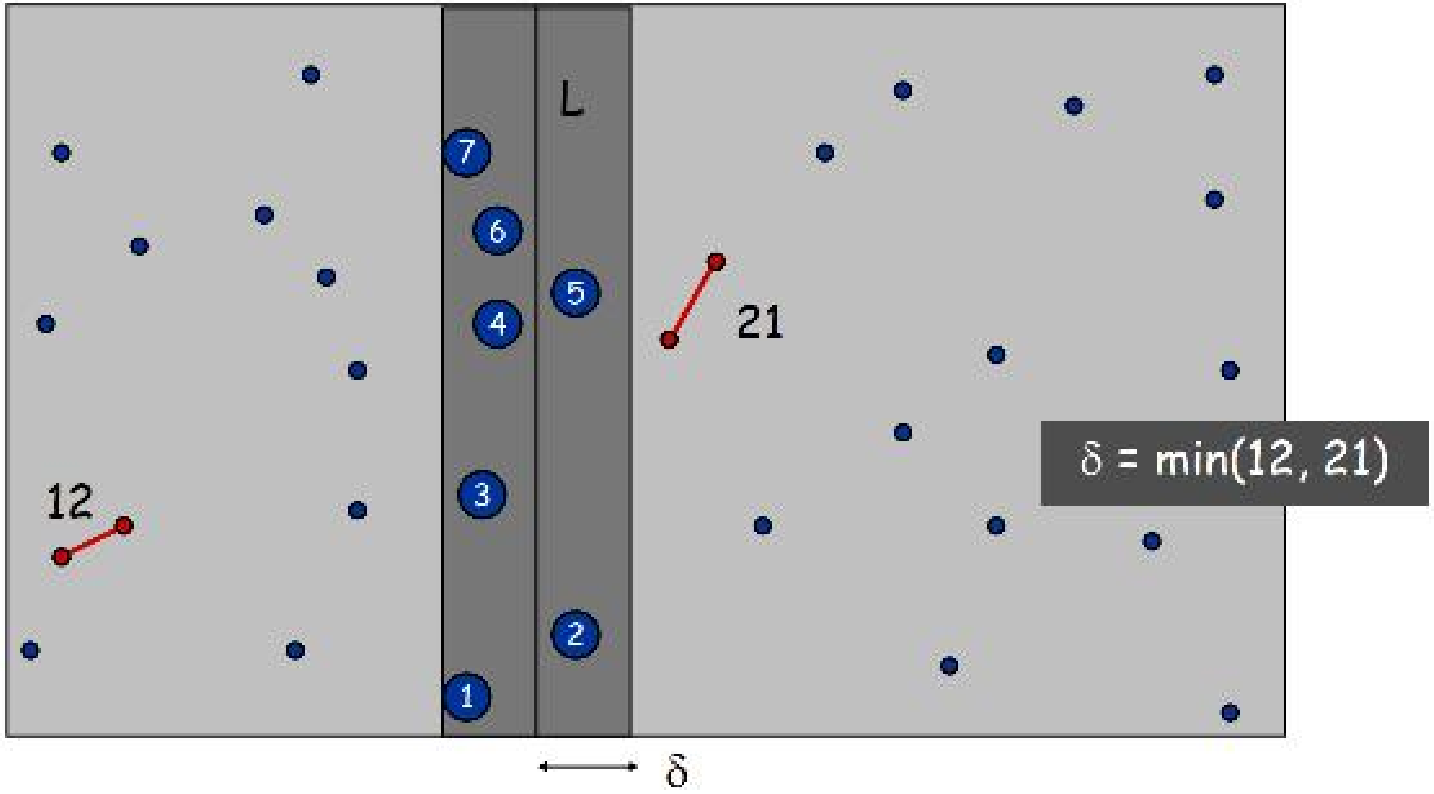


Closest Pair of Points

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

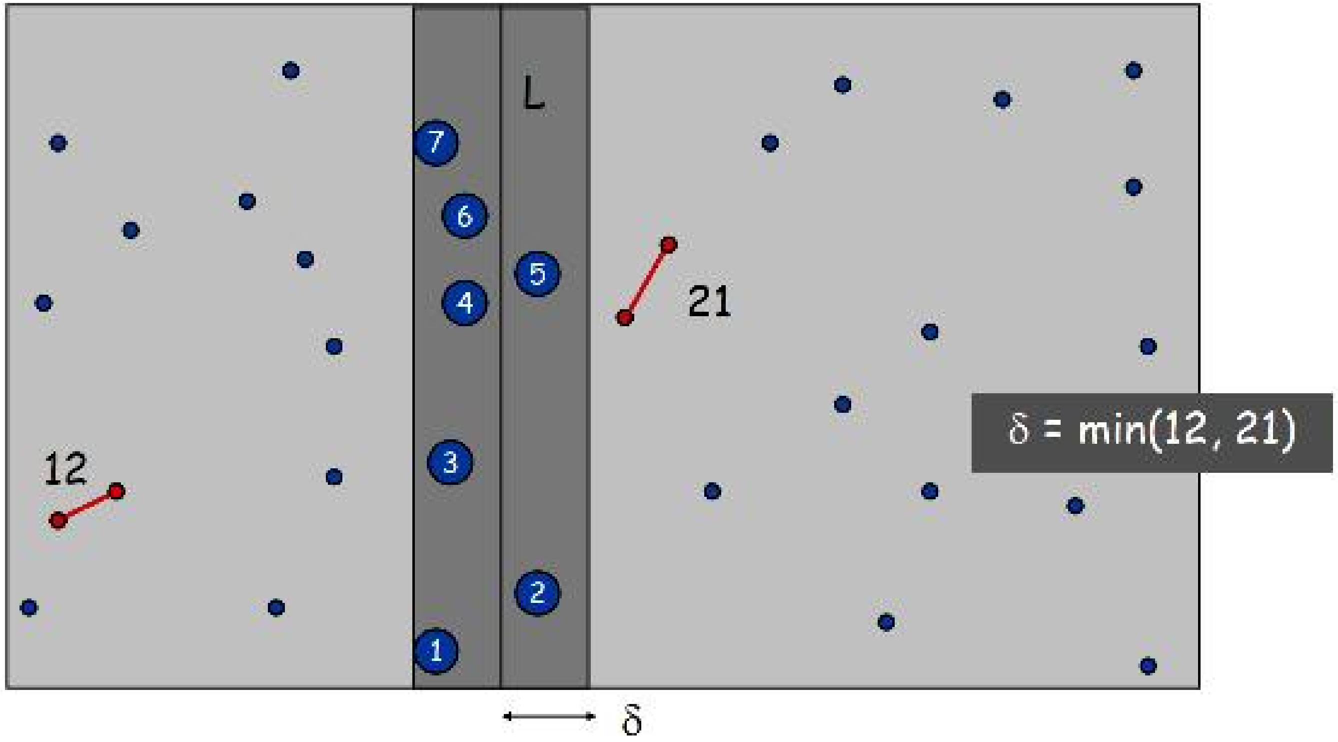
  Observation: only need to consider points within δ of line L

  Sort points in 2δ-strip by their y coordinate



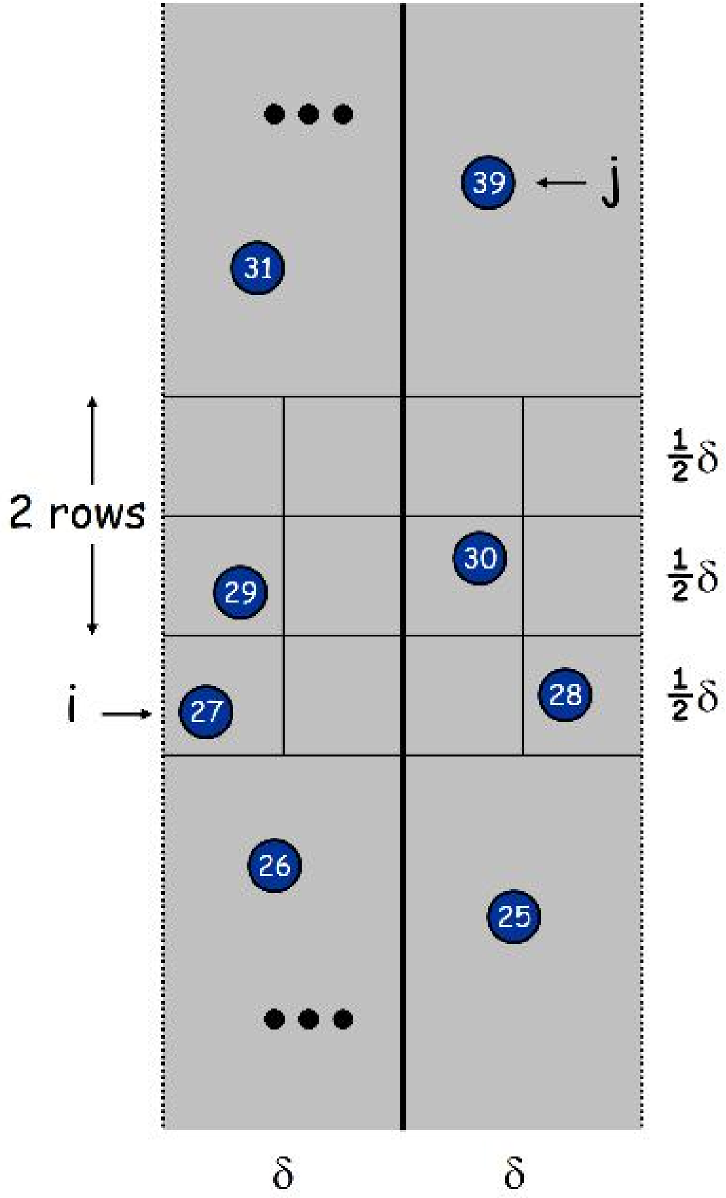
Closest Pair of Points

- Find closest pair with one point in each side, **assuming that distance < δ**
  - **Observation: only need to consider points within δ of line L**
  - **Sort points in 2δ-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 2δ-strip, with the i^{th} smallest y-coordinate
- **Claim.** If $|i - j| >= 12$, then the distance between s_i and s_j is at least δ
- **Pf.**
  - No two points lie in some $½\delta$-by-$½\delta$ box
  - Two points at least 2 rows apart have distance >= $2(½\delta)$



Closest Pair of Points

- Find closest pair with one point in each side, **assuming that distance < δ**
    - 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 >= δ and then the combine step will correctly return δ as distance of closest pair

Closest Pair Algorithm

```
Closest-Pair($p_1$, …, $p_n$) {
    Compute separation line L such that half the points          O(n log n)
    are on one side and half on the other side.

    $\delta_1$ = Closest-Pair(left half)                                    2T(n / 2)
    $\delta_2$ = Closest-Pair(right half)
    $\delta$  = 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           O(n)
    each point and next 11 neighbors. If any of these
    distances is less than $\delta$, update $\delta$.

    return $\delta$.
}
```

Closest Pair of Points: Analysis

- Running time

$$\mathrm{T}(n) \le 2T(n/2) + O(n \log n) \Rightarrow \mathrm{T}(n) = O(n \log^2 n)$$

- Q. Can we achieve O(nlogn)?
- 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)

$$T(n) \le 2T(n/2) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$

Integer Multiplication (Revisit)

Integer Arithmetic

- **Add.** Given two n-digit integers a and b, compute a + b.
  - O(n) bit operations

- **Multiply.** Given two n-digit integers a and b, compute a * b
  - Brute force solution: $\Theta(n^2)$ bit operations

	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Add

								1	1	0	1	0	1	0	1	0
								*	0	1	1	1	1	1	0	1
								1	1	0	1	0	1	0	1	0
							0	0	0	0	0	0	0	0	0	
						1	1	0	1	0	1	0	1	0		
					1	1	0	1	0	1	0	1	0			
				1	1	0	1	0	1	0	1	0				
			1	1	0	1	0	1	0	1	0					
		1	1	0	1	0	1	0	1	0						
	0	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0

Multiply

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Multiply four ½n - digit integers
  - Add two ½n-digit integers, and shift to obtain result

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

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

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

↑

assumes n is a power of 2

Karatsuba Multiplication

- To multiply two n-digit integers:
  - Add two ½n digit integers
  - Multiply **three** ½n-digit integers
  - Add, subtract, and shift ½n-digit integers to obtain results

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

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

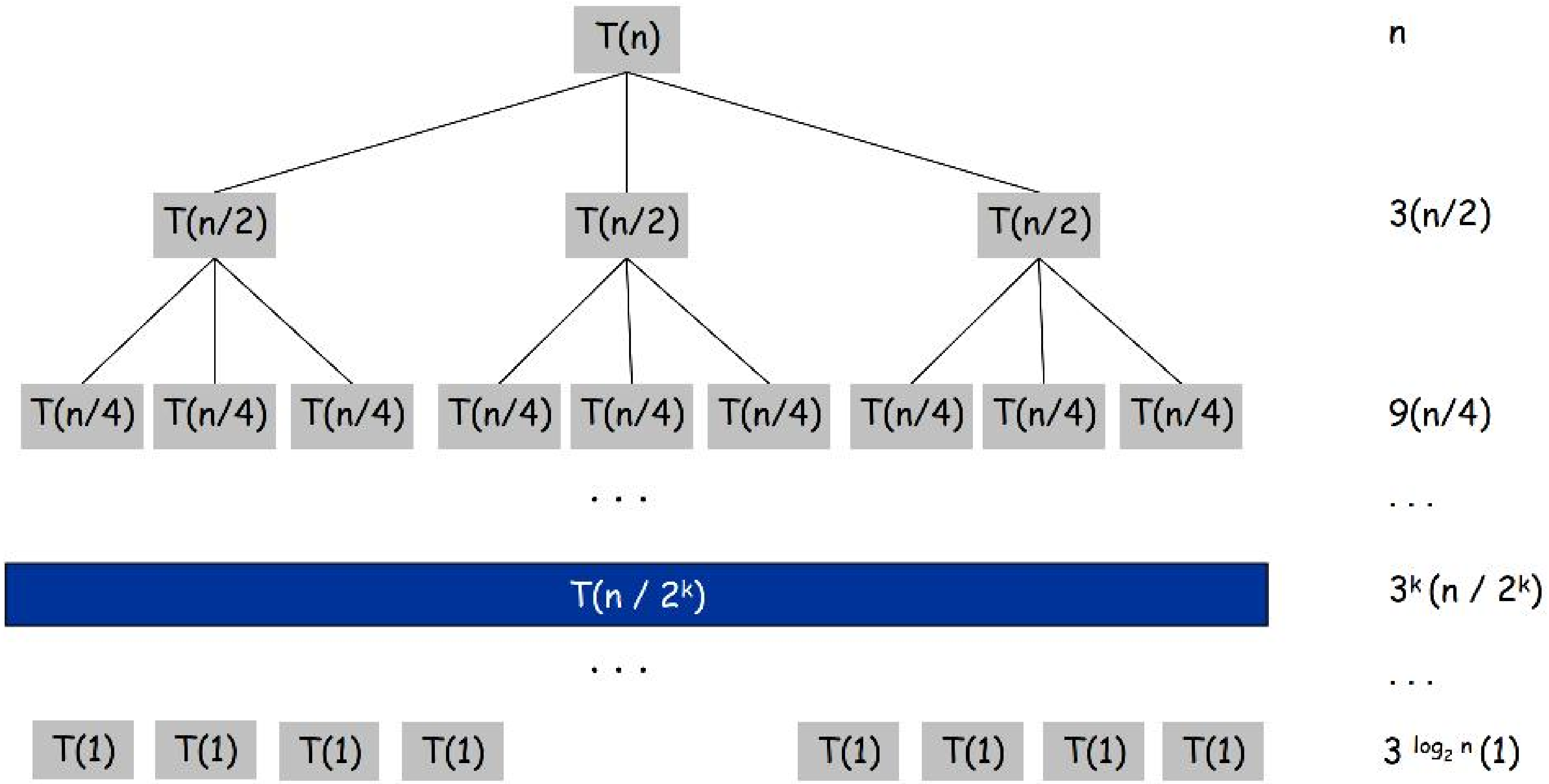
$$
\mathrm{T}(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}
$$

$$
\Rightarrow \mathrm{T}(n) = O(n^{\log_2 3}) = O(n^{1.585})
$$

Karatsuba: Recursion Tree

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

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



T(n) — n

T(n/2) T(n/2) T(n/2) — 3(n/2)

T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) — 9(n/4)

. . . — . . .

$T(n / 2^k)$ — $3^k (n / 2^k)$

. . . — . . .

T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) — $3^{\log_2 n}(1)$

Next Time:
Divide and Conquer (Cont.)