CS5800: Algorithms

Week 2 – Divide & Conquer

Dr. Ryan Rad

Spring 2024

Northeastern
University

Divide and Conquer

- Group Quiz
- Divide and Conquer
  - Maximum Subarray
- Recurrence Relation
- Substitution Method
- Master Theorem
- A geometric problem
- Quiz Review

Week 2 Quiz Groups

Week 2 Quiz - Group 1 — 3 students

Shirui Chen	Jixian Jia	Kedar Prabhune

Week 2 Quiz - Group 2 — 3 students

Kai Li *(He/Him)*	Palkan Motwani	Liyao Zhang

Week 2 Quiz - Group 3 — 3 students

Ji Chen	Pengcheng Ding	Tingyu Li

Week 2 Quiz - Group 4 — 3 students

Chengyan Tan	Yangfei Wu *(He/Him)*	Kai Zong

Week 2 Quiz - Group 5 — 3 students

Meida Li	Zhecheng Li	Jingzi Wang

Week 2 Quiz - Group 6 — 3 students

Hanzhang Peng	Yihe Xiong	Zecheng Zhou

Week 2 Quiz - Group 7 — 3 students

Hanqing Bao	Yichen Wang *(He/Him)*	Liting Zhou

Week 2 Quiz - Group 8 — 3 students

Jianyan Chen	Zixiang Hu	Zhixin Zeng

Week 2 Quiz - Group 9 — 3 students

Yu Ge	Chanyuan Liu	Lu Yan

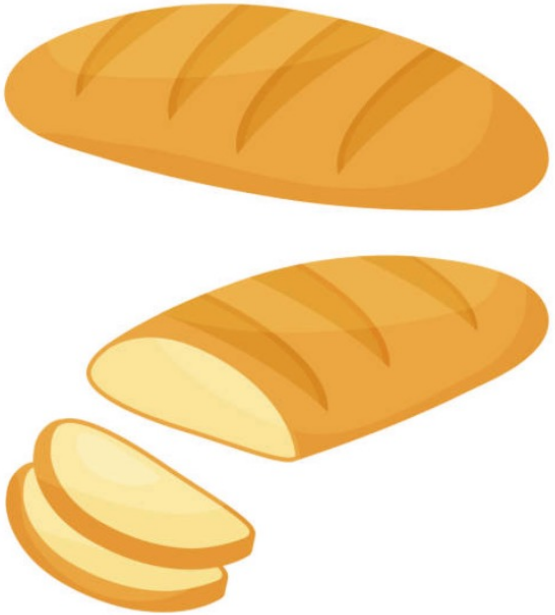
Week 2 Quiz - Group 10 — 3 students

Shuojia Lin	Xinyu Xie	Yue Xu

Week 2 Quiz - Group 11 — 2 students

Wei Song	Jiale Zhang

DAC examples in Real Life



“Divide-and-Conquer” paradigm

Divide

- The problem into one or more subproblems that are smaller instances of the same problem.

Conquer

- The subproblems by solving them recursively.
- ***Base case:*** If the subproblems are small enough, just solve them by brute force.

Combine

- The subproblem solutions to form a solution to the original problem.

with the divide-and-conquer approach, computational complexity is estimated using mathematical equations known as **recurrence relations**.

If you prefer some pseudocode…

Solve(P):

```
if (P is small enough)
  solution = DirectSolve(P)
else
  <P1,P2,…,Pk> = DivideProblem(P)
  for (i=1 to k)
    solutioni = Solve(Pi)
  solution = Combine(solution1,…,solutionk)
return solution
```

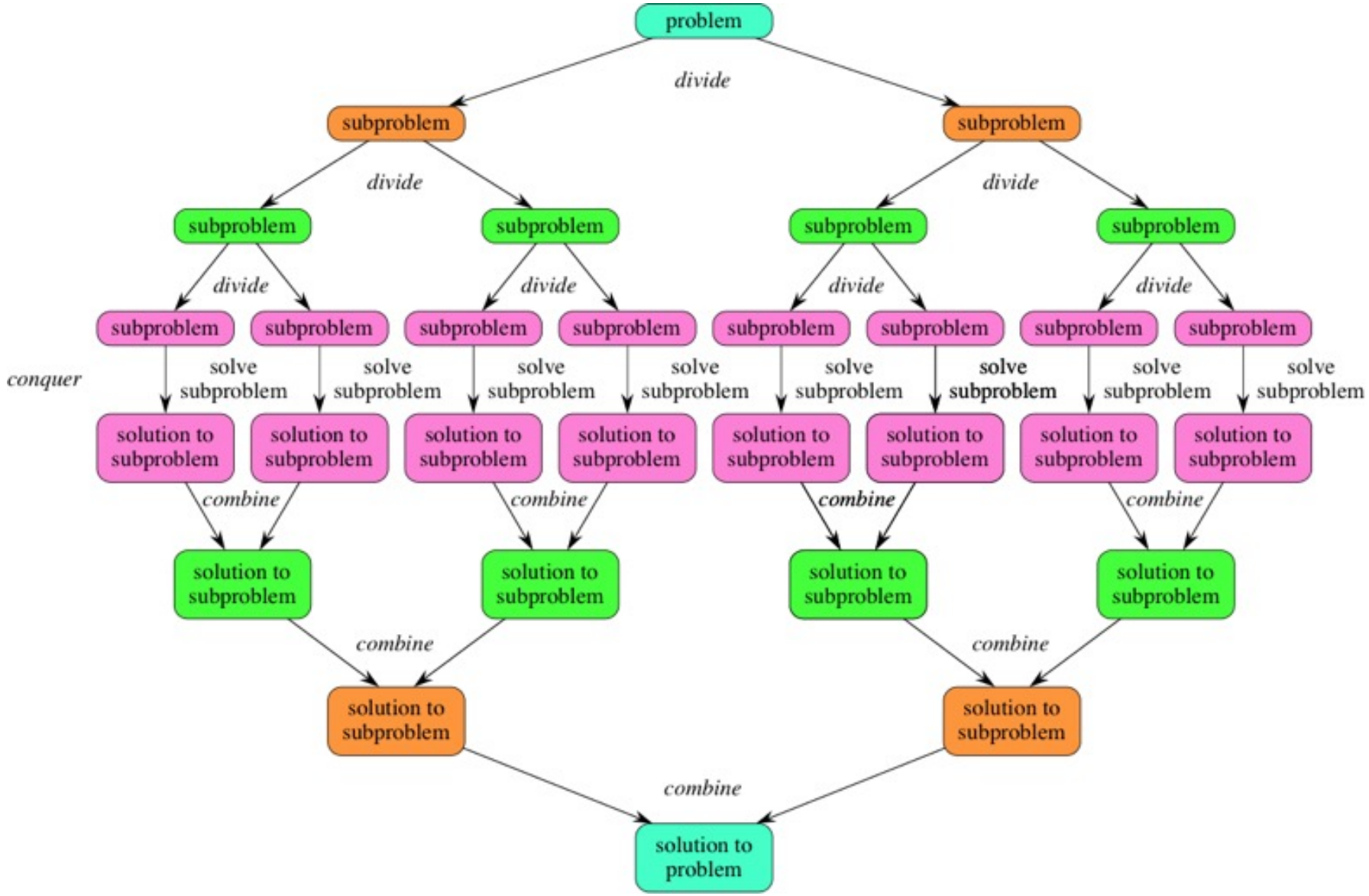
Or, use brute-force if (sub)problem is simple.

Divide the problem into smaller subproblems.

Recursively solve subproblems.

Combine solutions of subproblems to get solution for original problem.

If you prefer something graphical



Source: https://www.khanacademy.org/

Quiz Review

Consider the following problems:

Inventory Management: In supply chain management, finding the minimum subarray can help identify the shortest sequence of days with low product demand, allowing businesses to optimize their inventory and reduce costs.

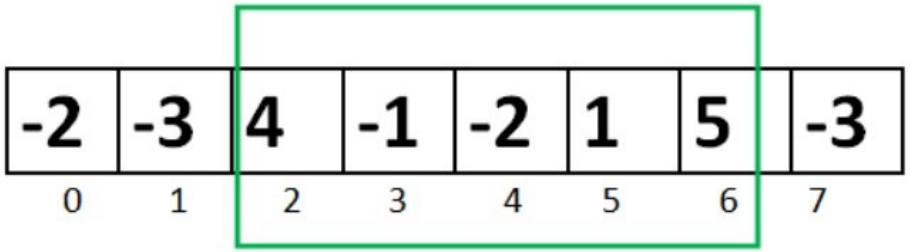
Week #2 Quiz – Q2

Let A be an array of numbers. In the maximum subarray problem, your goal is to determine indices x and y so that the sum of the entries in the subarray A[x..y] is as large as possible.

For example, in the array below, the optimal indices are x=2 and y=6, and the maximum subarray is [4,-1,2,1,5]. This subarray has sum S=4-1-2+1+5=7, which is the largest possible sum.

Suppose A = [-1, 1, 3, -5, 7, 9, -11, 13, 15, -17, 19, 21, -23].

Determine S, the largest possible sum of a subarray of A.



Week #2 Quiz – Q2

There's an obvious **$O(n^3)$** algorithm.

Run a double for loop

```
For x in range(n):
        For y in range(x,n):
                // Find sum of subarray[x,y]
                For k in range(x,y):
                If this sum is better than any we've found, update this value
```

Return best answer

Week #2 Quiz – Q2

- Quick improvement:
  - Keep the sum so far

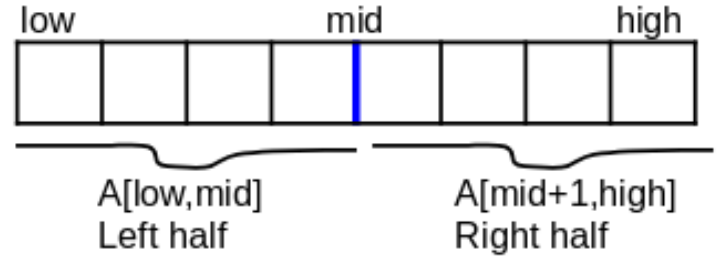
```
For x in range(n):
        sum = 0
        For y in range(x,n):
                sum += A[y]
                If this sum is better than any we've found, update this value
```

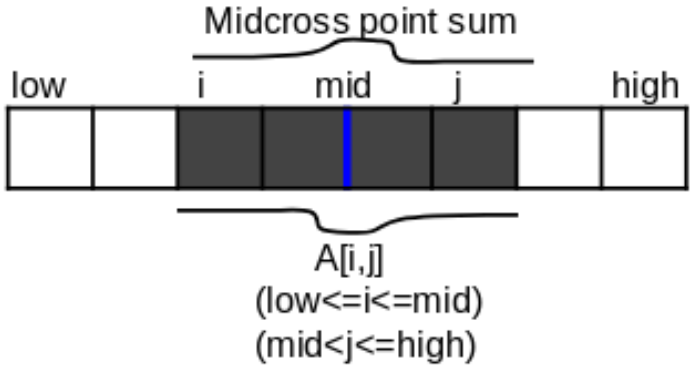
Return best answer

$O(n^2)$

Week #2 Quiz – Q2

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```





Week #2 Quiz – Q2

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

$O(n \log n)$

Week #2 Quiz – Q2

Take a look at the following paper for a more efficient DAC solution:

Divide & Conquer strikes back: maximum-subarray in linear time

https://personal.utdallas.edu/~daescu/maxsa.pdf

Kadane's Algorithm

Two variables:

localMax: store the maximum subarray sum ending at index i

globalMax: store the maximum subarray sum we have encountered so far

at each index i:

```
currentNum = array[i]
localMax = max(localMax + currentNum, currentNum)
globalMax = max(globalMax, localMax)
```

	0	1	2	3	4	5	6	7	8
array	-2	1	-3	4	-1	2	1	-5	4
localMax	-2	1	-2	4	3	5	6	1	5
globalMax	-2	1	1	4	4	5	6	6	6

[4, -1, 2, 1] has the largest sum = 6

Kadane's Algorithm

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        maxSub = nums[0]
        curSum = 0

        for n in nums:
            if curSum < 0:
                curSum = 0
            curSum += n
            maxSub = max(maxSub, curSum)
        return maxSub
```

$O(n)$

Methods to solve Recurrences

Substitution	First guess, then prove with induction
Recursion Tree 	Mostly used for guessing
Master Theorem	

Substitution Method

In the substitution method for solving recurrences we:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

Substitution (Example)

Let’s solve the following Recurrence using the ***Substitution*** method:

Recurrence: $T(1) = 1$ and $T(n) = 2T(\lfloor n/2 \rfloor) + n$ for $n > 1$.

Do this on whiteboard!

Substitution (Example)

$a < b, \text{then } \log(a) < \log(b)$

Let's solve the following Recurrence using the ***Substitution*** method:

Recurrence: $T(1) = 1$ and $T(n) = 2T(\lfloor n/2 \rfloor) + n$ for $n > 1$.

We guess that the solution is $T(n) = O(n \log n)$. So we must prove that $T(n) \leq cn \log n$ for some constant c. (We will get to n_0 later, but for now let's try to prove the statement for all $n \geq 1$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \quad (\text{for } c \geq 1) \end{aligned}$$

Warning

1. Using the substitution method, it is easy to prove a weaker bound than the one you're supposed to prove.

Recursion Tree

Recursion Tree:

- A recursion tree is a tree where **each node** represents the ***cost*** of a certain recursive ***subproblem***.

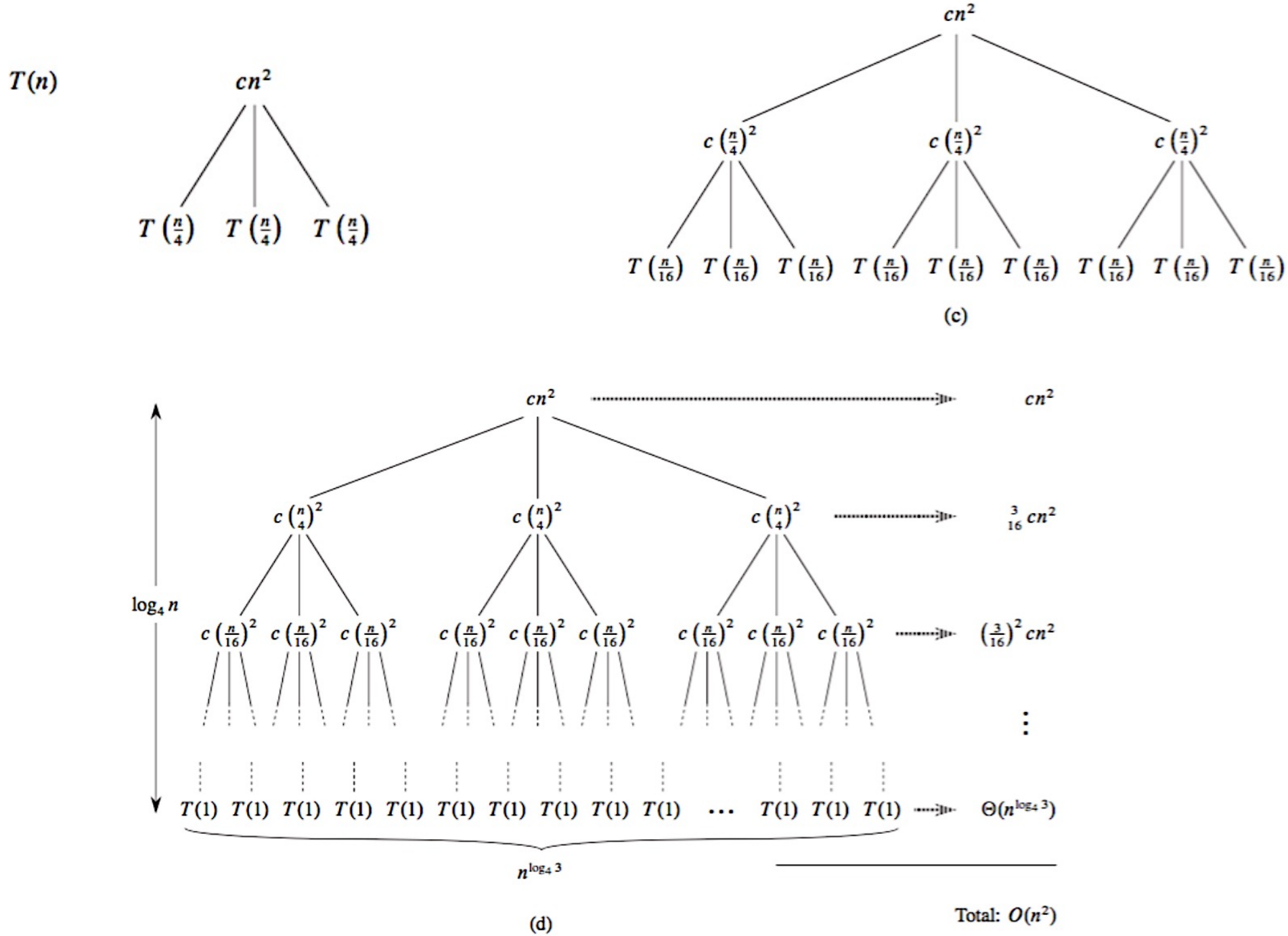
Note:
We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them.

Recursion Tree (Example)

Let's **generate guesses** for the following Recurrence using the ***Recursion Tree*** method:

$$\textbf{Recurrence: } T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

Do this on whiteboard!


T(n)
cn²
T(n/4) T(n/4) T(n/4)
cn²
c(n/4)² c(n/4)² c(n/4)²
T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16) T(n/16)
(c)
cn²
cn²
c(n/4)² c(n/4)² c(n/4)²
(3/16) cn²
log₄ n
c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)² c(n/16)²
(3/16)² cn²
T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) T(1) ... T(1) T(1) T(1)
Θ(n^log₄ 3)
n^log₄ 3
(d)
Total: O(n²)

Recursion Tree (Example)

Let's **generate guesses** for the following Recurrence using the ***Recursion Tree*** method:

Recurrence: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

Do this on whiteboard!

Sum of finite geometric sequence:

$$S_n = \sum_{i=1}^{n} a_i r^{i-1} = a_1 \left(\frac{1 - r^n}{1 - r} \right)$$

Sum of infinite geometric series:

$$S = \sum_{i=0}^{\infty} a_i r^i = \frac{a_1}{1 - r}$$

[note: if $|r| \geq 1$, the infinite series does not have a sum]

Master Theorem

- Used for many divide-and-conquer ***master recurrences*** of the form $T(n) = aT(n/b) + f(n)$, where a ≥ 1, $b > 1$, and $f(n)$ is an asymptotically nonnegative function defined over all sufficiently large positive numbers.

- Master recurrences describe recursive algorithms that divide a problem of size n into a subproblems, each of size n/b. Each recursive subproblem takes time $T(n/b)$ (unless it's a base case). Call $f(n)$ the ***driving function***.

Master Theorem

Used for many divide-and-conquer ***master recurrences*** of the form

$$\boldsymbol{T(n) = aT(n/b) + f(n)}$$

where a ≥ 1, $b > 1$, and $f(n)$ is an asymptotically nonnegative function defined over all sufficiently large positive numbers.

Master recurrences describe recursive algorithms that divide a problem of size n into a subproblems, each of size n/b. Each recursive subproblem takes time $T(n/b)$ (unless it's a base case). Call $f(n)$ the ***driving function***.

The master theorem compares the function $\boldsymbol{n^{log_b{}^a}}$ to the function $\boldsymbol{f(n)}$

Master Theorem

Then you can solve the recurrence by comparing $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.
(Intuitively: cost is dominated by leaves.)

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$ is a constant.
($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.
[In the previous editions of the book, case 2 was stated for only $k = 0$.]

Master Theorem

Case 3: $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

What's with the Case 3 regularity condition?

- Generally not a problem.
- It always holds whenever $f(n) = n^k$ and $f(n) = \Omega(n^{\log_b a+\epsilon})$ for constant $\epsilon > 0$. So you don't need to check it when $f(n)$ is a polynomial.

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The master theorem compares the function $\boldsymbol{n^{log_b{}^a}}$ to the function $\boldsymbol{f(n)}$

Master Theorem

Example 1 $T(n) = 9T(n/3)+n$

$$\boldsymbol{T(n) = aT(n/b) + f(n)}$$

The master theorem compares the function $\boldsymbol{n^{log_b{}^a}}$ to the function $\boldsymbol{f(n)}$

Master Theorem

Example 1 $T(n) = 9T(n/3)+n$

Solution Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Master Theorem

Example 2 $T(n) = T(2n/3) + 1$

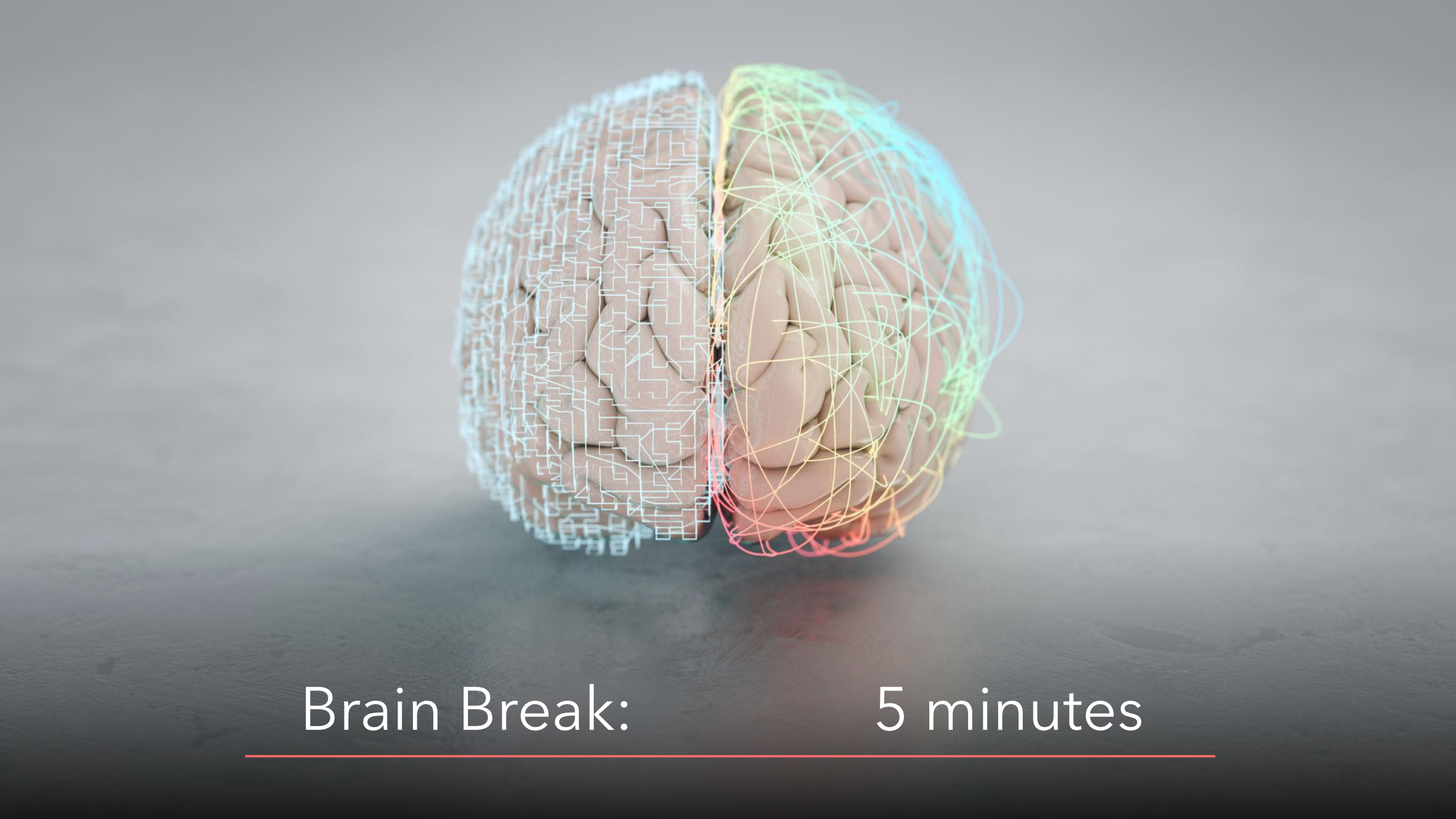
$$\boldsymbol{T(n) = aT(n/b) + f(n)}$$

The master theorem compares the function $\boldsymbol{n^{log_b{}^a}}$ to the function $\boldsymbol{f(n)}$

Master Theorem

Example 2 $T(n) = T(2n/3) + 1$

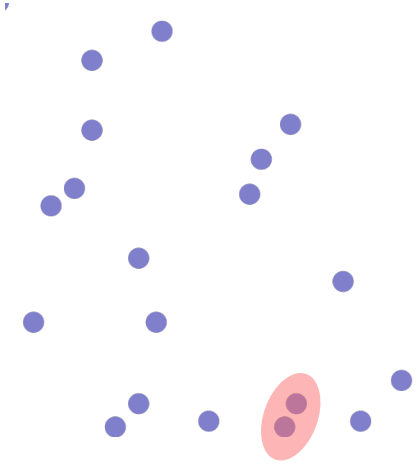
Solution Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is $T(n) = \Theta(\log n)$.



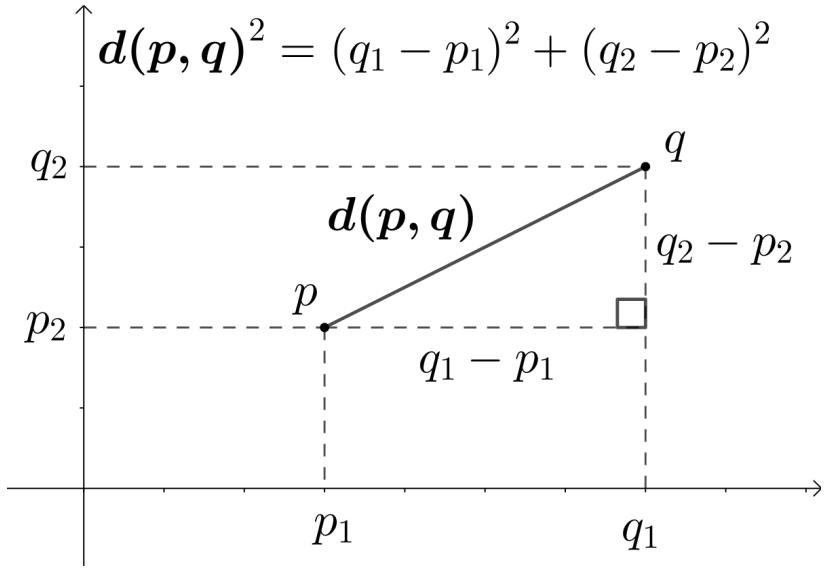
Brain Break: 5 minutes

Closest Pair of Points

Computational Problem:
Given a set of points, find the closest pair of points

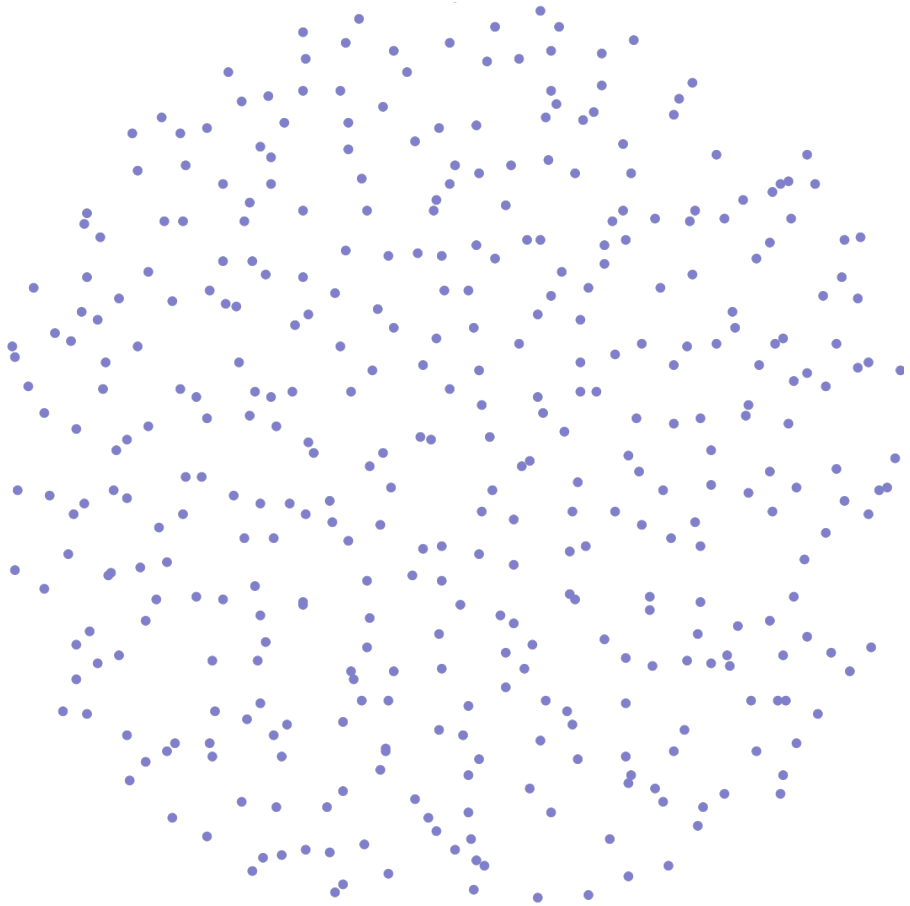


Euclidean Distance Formula:

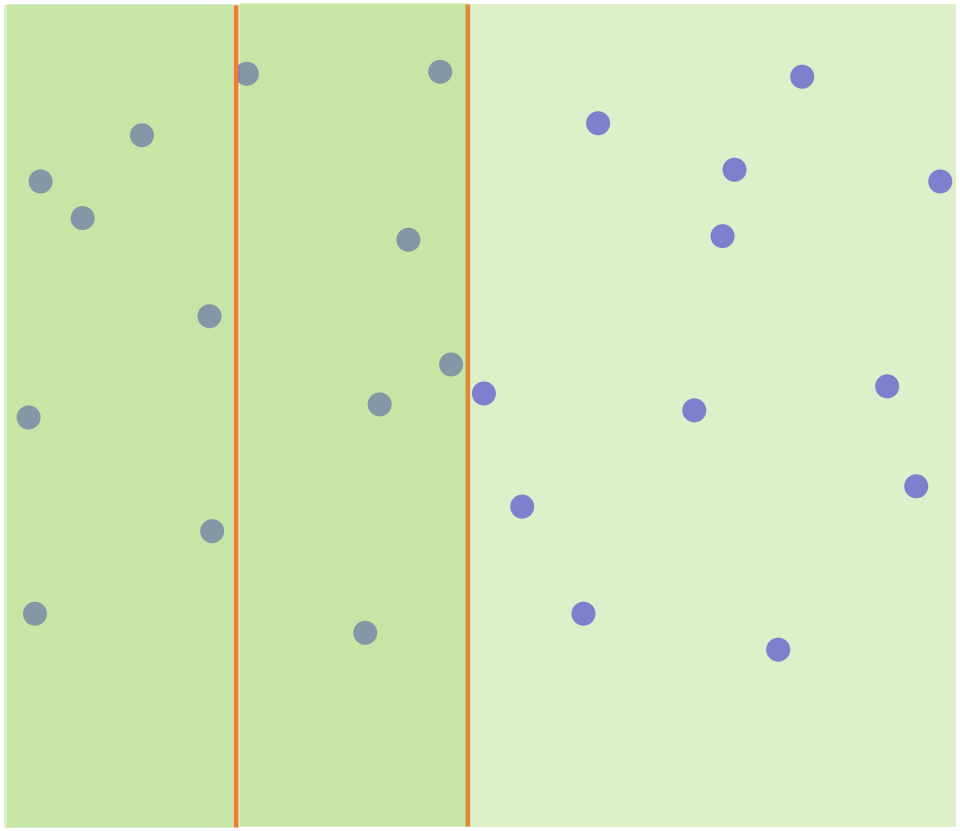


Closest Pair (Brute Force)

- Pairwise distance between every single pair - $\boldsymbol{O(n^2)}$
- Guess what?
  - We need a better algorithm!

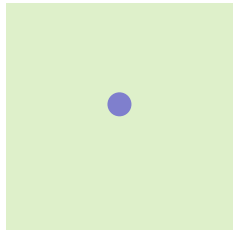


Closest Pair (DAC Solution)

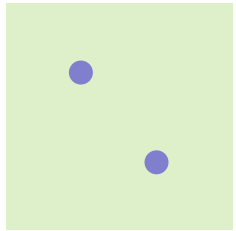


Closest Pair (DAC Solution)

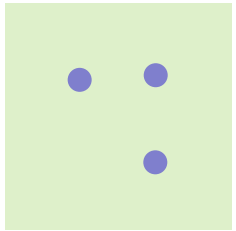
$n = 1$



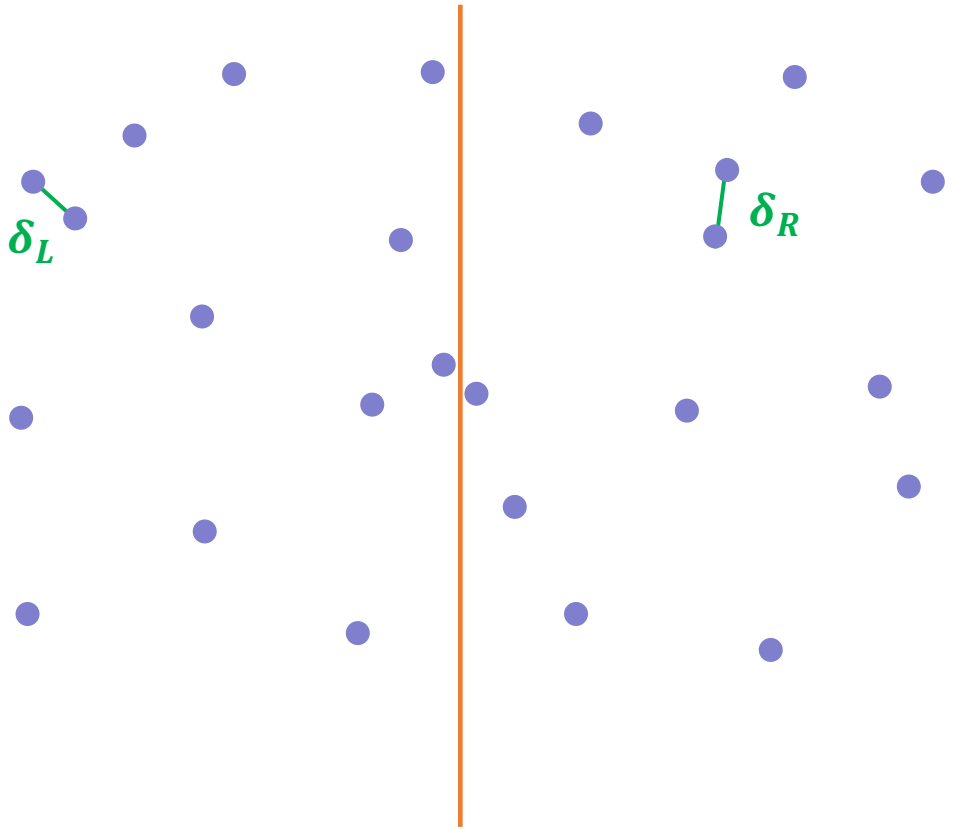
$n = 2$



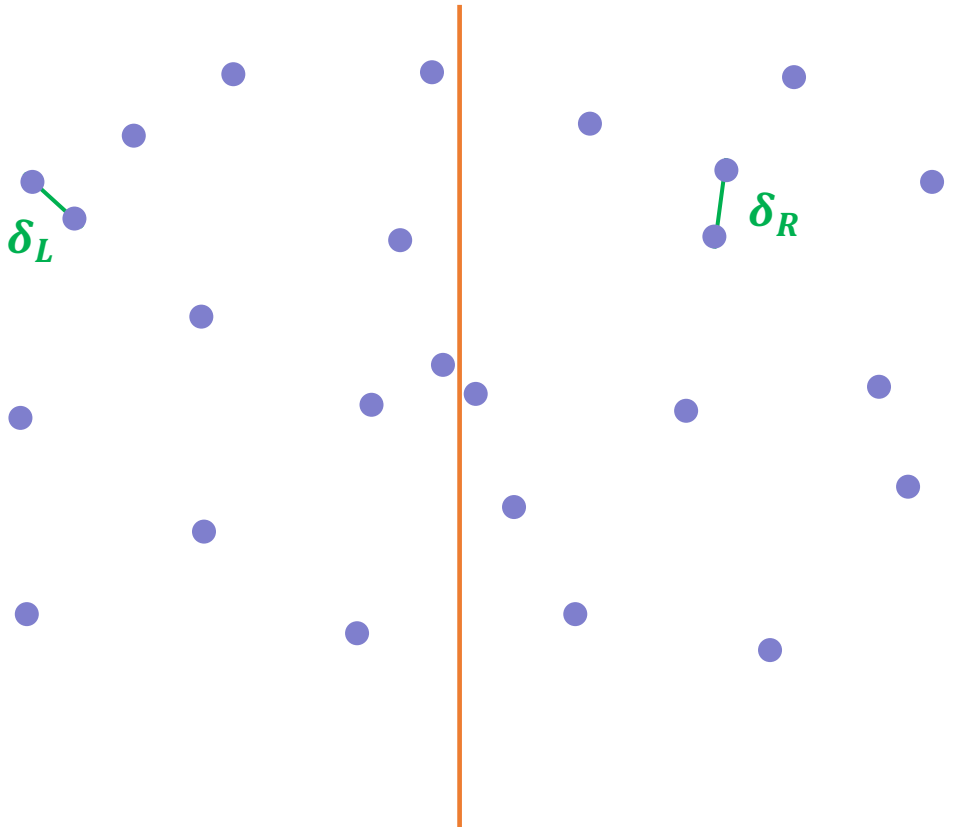
$n = 3$



Closest Pair (DAC Solution)



Closest Pair (DAC Solution)



$$\delta = min\,(\delta_L, \delta_R)$$

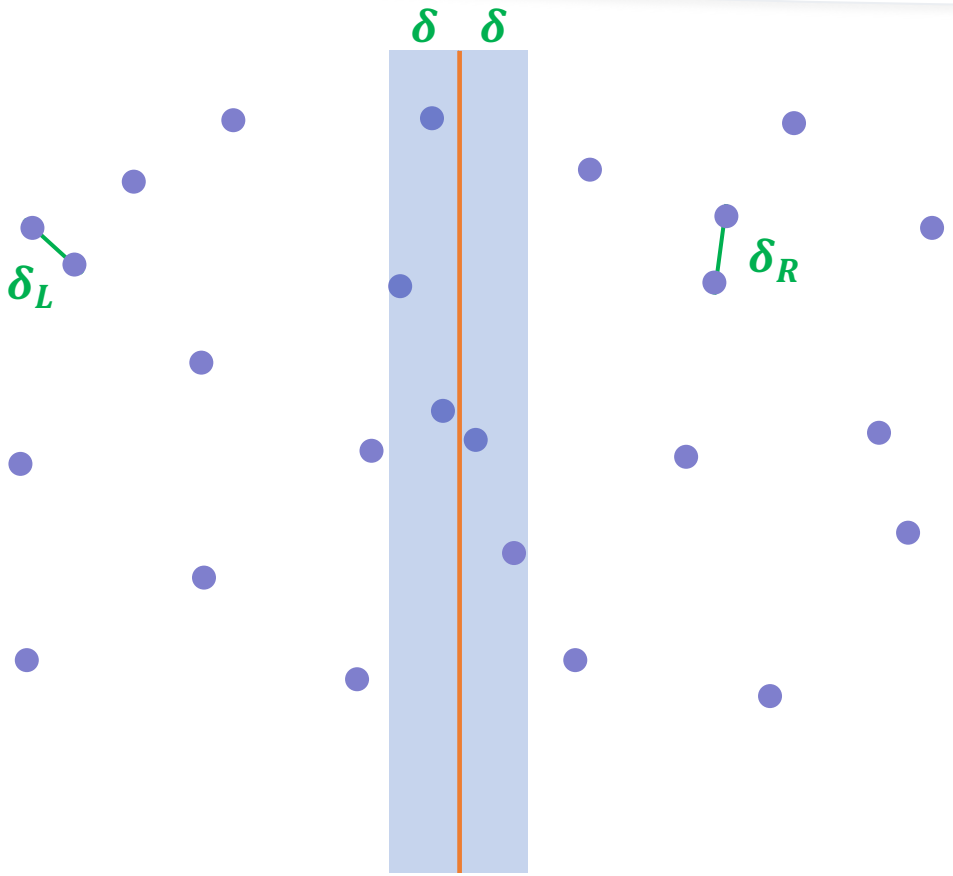
Question:
Would this approach guarantee we find the closest pair?

No!

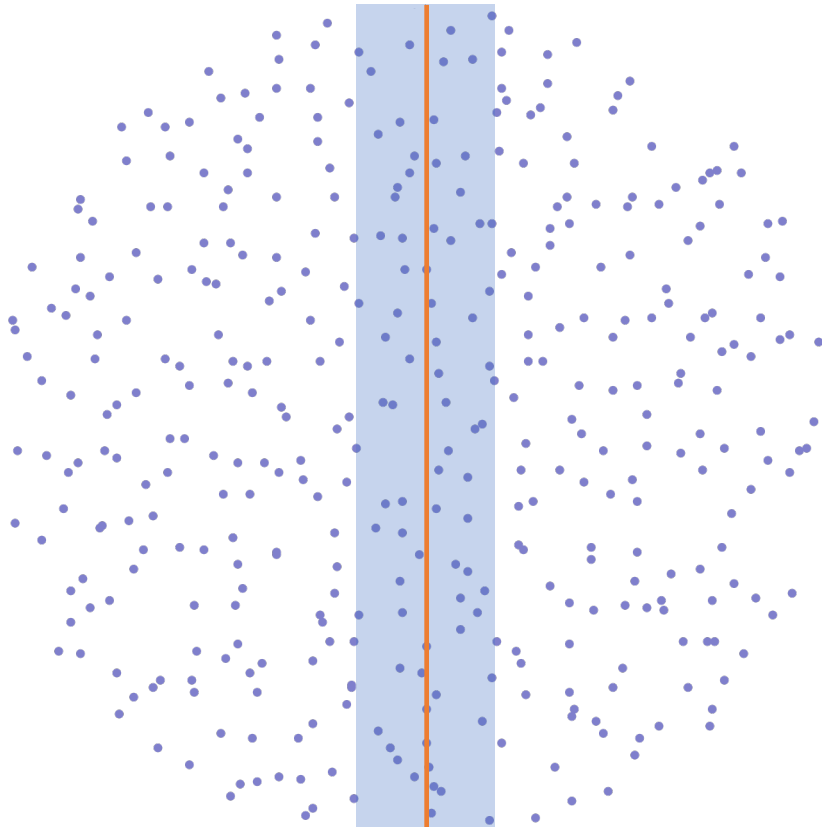
What if two points with different membership (L,R) form the closest pair?

We have a problem here!

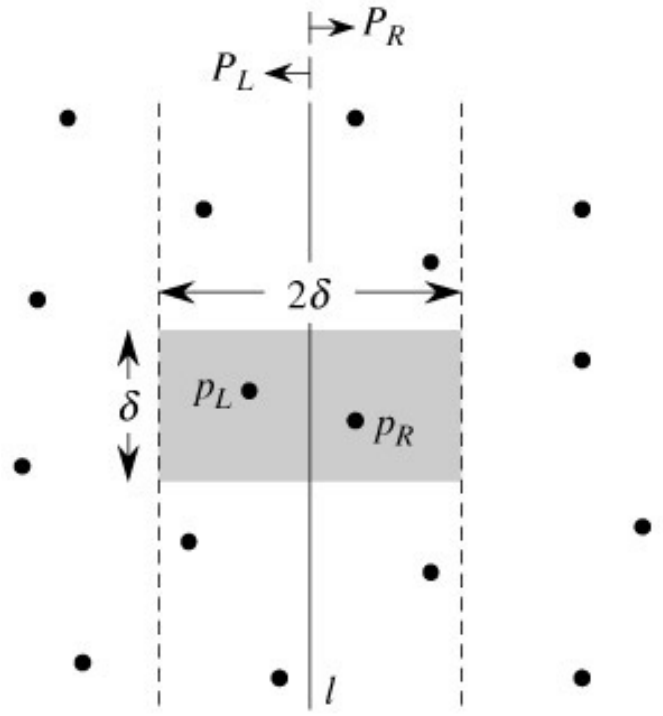
Closest Pair (Optimized DAC Solution)



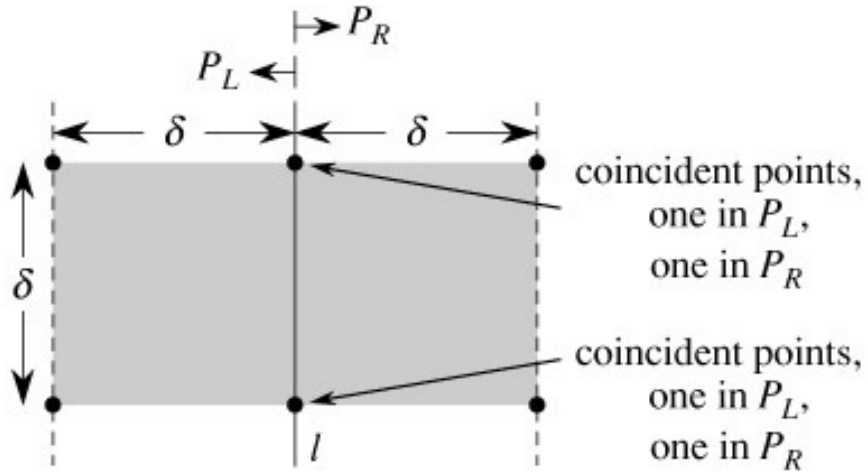
$$\delta = min\,(\delta_L, \delta_R)$$



Closest Pair (Optimized DAC Solution)



(a)



(b)

Closest Pair (Optimized DAC Solution)

Remember we need to sort twice in both dimensions! Once in terms of x and once in terms of y.

```
closestPair(X,Y){
        n = X.length;

        // base cases:
        if (n==2): return dist (X[1], x[2]):
        (n==3) : return min(dist(X[1], x[2]), dist (x[2], X[3]), dist (X[1], X[3]));

        // divide
        mid = X[n/2]:
        d1 = closestPair (X[1...mid], Y) ;
        dr = closestPair(X[mid+1...n], Y);
        d = min(dl, dr);

        // combine
        S = points in Y whose x-coordinates are in the range of [mid.x-d, mid.x+d];
        for i=1 to S.length
                for j=1 to j=5
        d = min(d, dist(S[i], S[i+j]));
        return d;
}
```

Closest Pair (Optimized DAC Solution)

```
closestPair(X,Y){
        n = X.length;

        // base cases:
        if (n==2): return dist (X[1], x[2]):
        (n==3) : return min(dist(X[1], x[2]), dist (x[2], X[3]), dist (X[1], X[3]));

        // divide
        mid = X[n/2]:
        d1 = closestPair (X[1...mid], Y) ;
        dr = closestPair(X[mid+1...n], Y);
        d = min(dl, dr);

        // combine
        S = points in Y whose x-coordinates are in the range of [mid.x-d, mid.x+d];
        for i=1 to S.length
                for j=1 to j=5
                d = min(d, dist(S[i], S[i+j]));
        return d;
}
```

$O(n\,logn)$

$O(1)$

$$T(n) = \begin{cases} n > 3, & 2T\left(\frac{n}{2}\right) + O(n\,logn) \\ n \leq 3, & O(1) \end{cases}$$

$O(n)$

$O(n)$

Week #2 Quiz

Question #1

Let f(n) and g(n) be two functions, defined for each positive integer n.

Recall that f(n) = O(g(n)) if there exist positive constants **c** and $\mathbf{n_0}$ for which for all integers . If no such constants **c** and $\mathbf{n_0}$ exist, then we say that f(n) ≠ O(g(n)).

Let $f(n) = n^3 + 2n^2 + 3n + 4$.

Then I claim that $f(n) = O(n^3)$ and $f(n) \neq O(n^2)$.

Determine whether the above claim is TRUE or FALSE.

Week #2 Quiz – Q1 (Answer)

$n^3 + 2n^2 + 3n + 4 \leq 2n^3$ for all $n \geq 4$. So you can pick $c=2$ and $n_0=4$ to show that $f(n) = O(n^3)$.

By definition, we need to show that constant numbers c and n_0 do not exist.
For example, suppose $c=10000$, and $n^3 + 2n^2 + 3n + 4 \leq 10000n^2$ for all $n \geq n_0$.
Divide both sides by n^2 so that we get $n + 2 + 3/n + 4/n^2 \leq 10000$.
Notice that this inequality is <u>false</u> for $n=10000$. In fact, it's false for <u>every</u> $n \geq 9998$. So there is no way that a constant n_0 can exist.

Week #2 Quiz

Short answers:

- 1. True
- 2. 56
- 3. 4/3
- 4. Theta(n^3), Theta(n^3 log n), Theta(n^4)

Questions?

Northeastern University