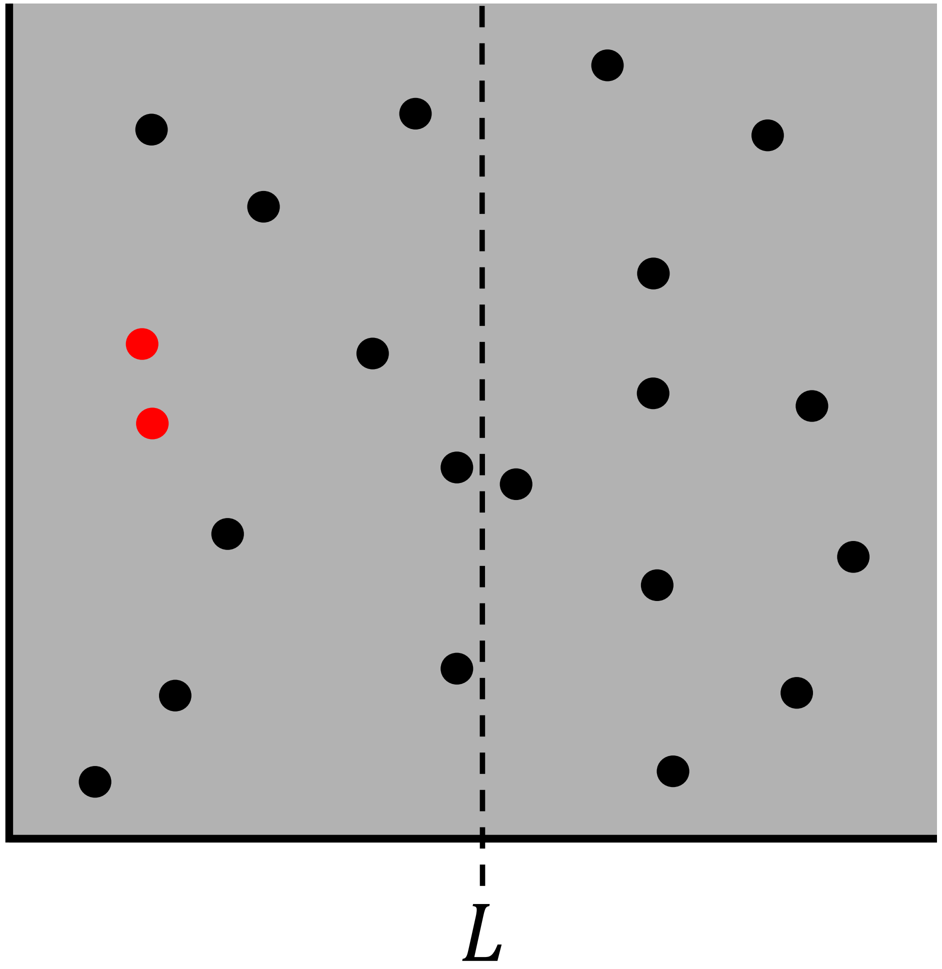
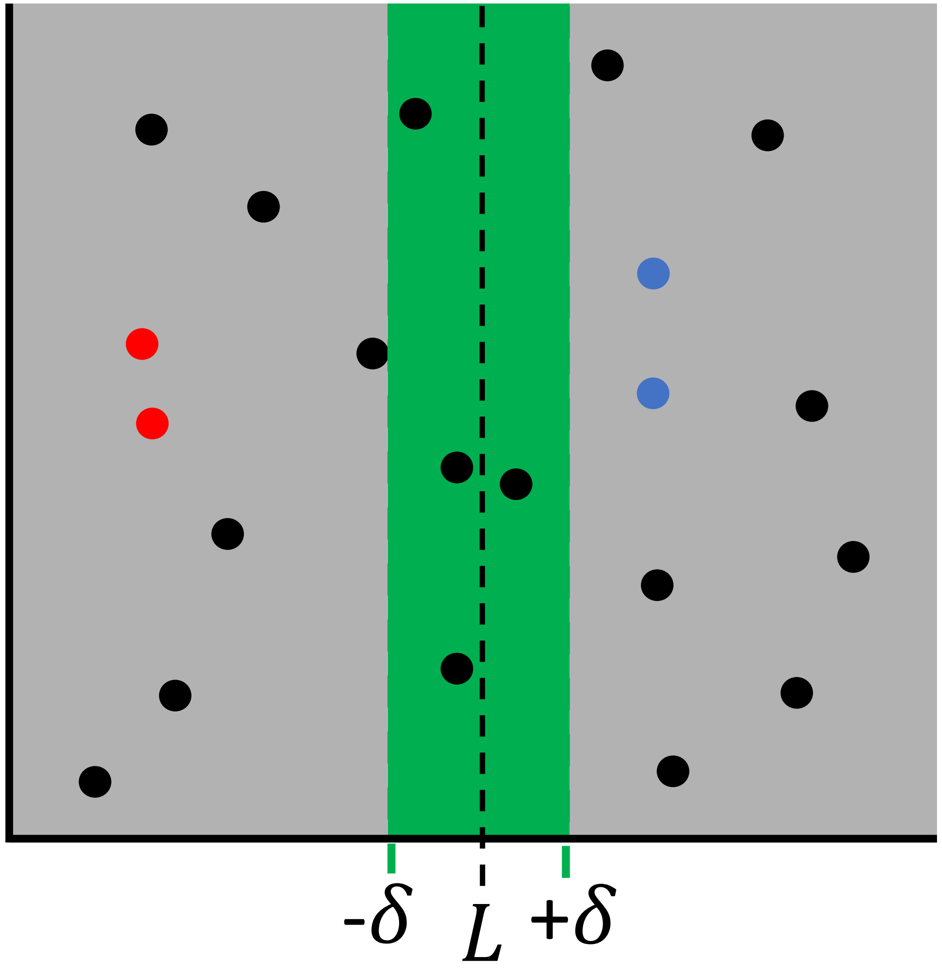
Closest Pair of Points

CSCI 532

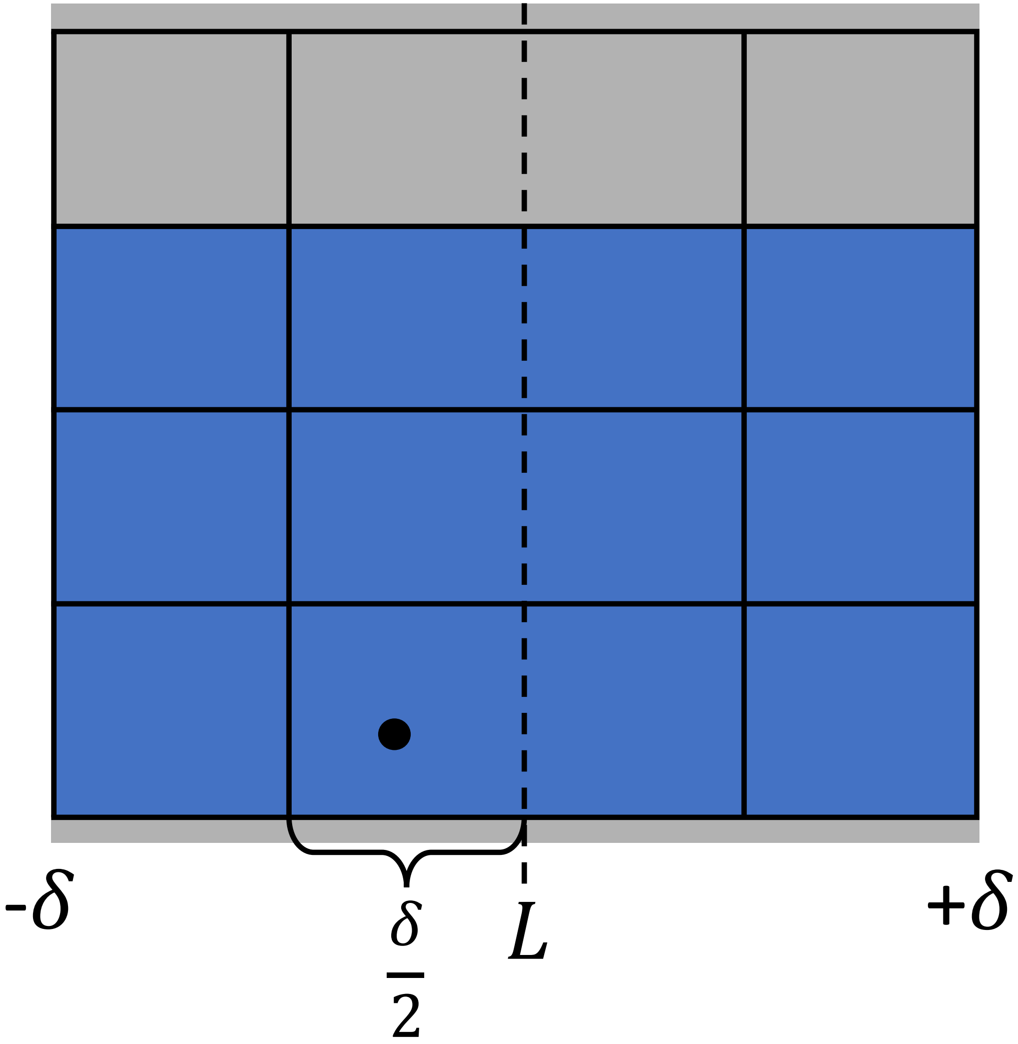
Closest Pair Problem – Divide and Conquer



Closest Pair Problem – Divide and Conquer



Closest Pair Problem – Divide and Conquer



Only care about certain boxes

\+ At most one point per box

Fixed number of points to check

Straddle point hunting:

$O(n^2) \longrightarrow O(n \log n)$

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

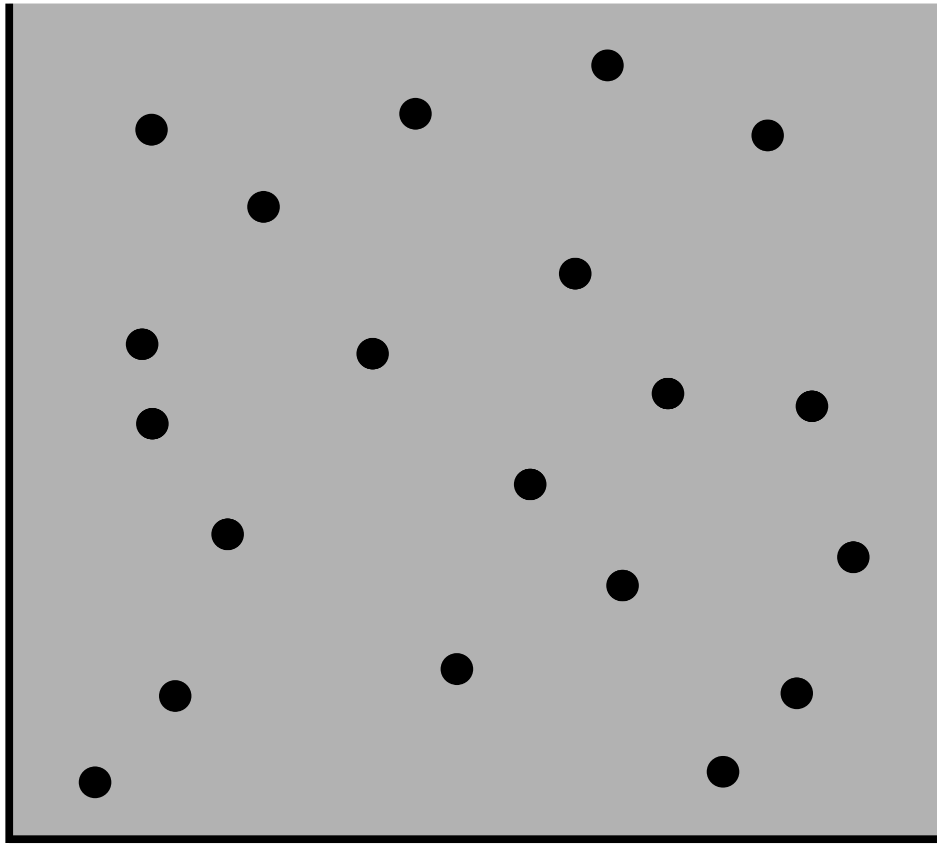
Closest Pair Problem – Algorithm

`findClosestPair(P):`

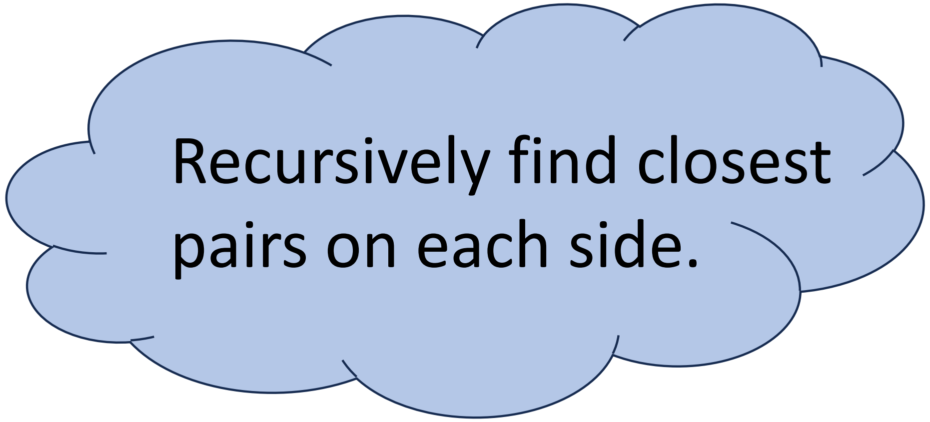
1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. **Determine d_{left} and d_{right}.**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Recursively find closest pairs on each side.

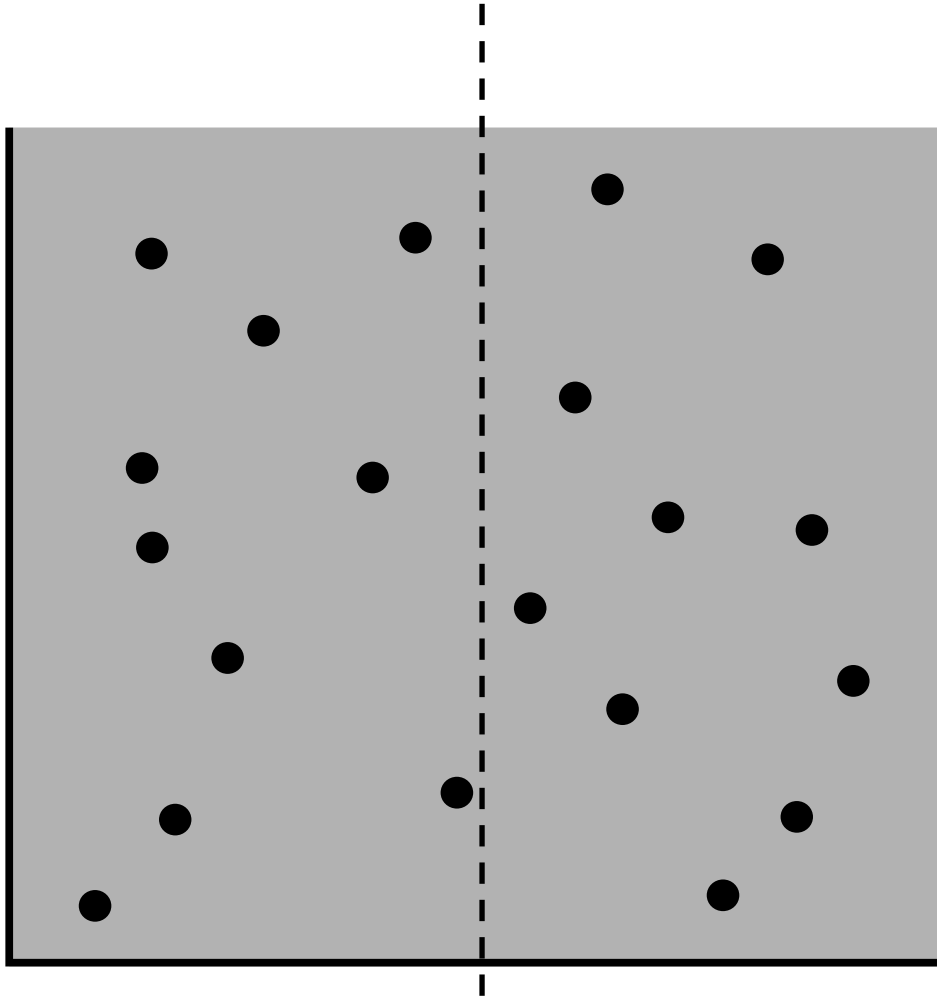
Closest Pair Problem – Divide and Conquer



Recursive Process:



Closest Pair Problem – Divide and Conquer

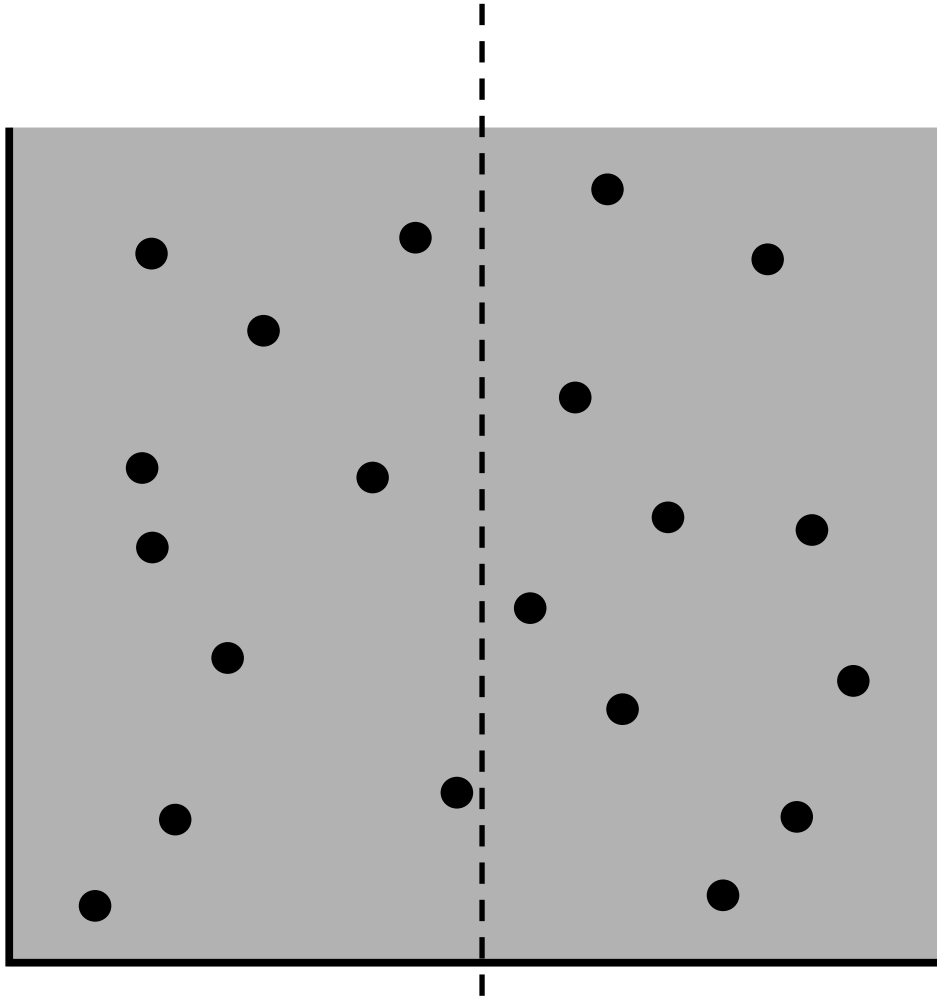


Recursive Process:

1. Divide points in half.

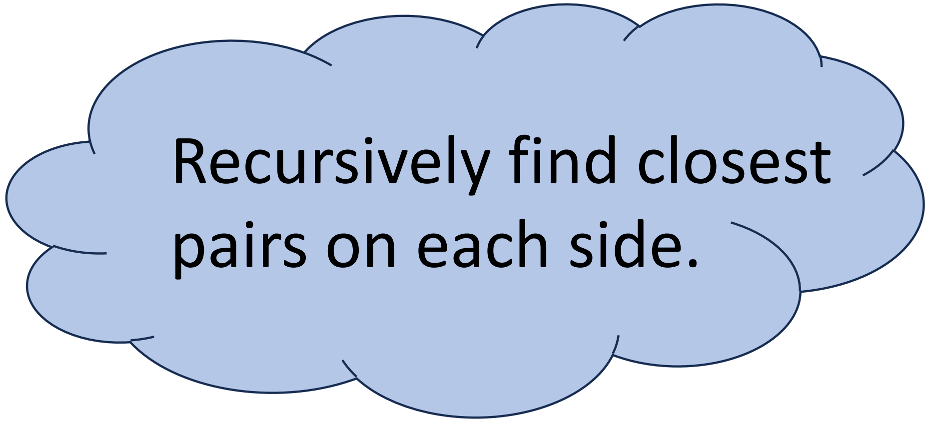
Recursively find closest pairs on each side.

Closest Pair Problem – Divide and Conquer

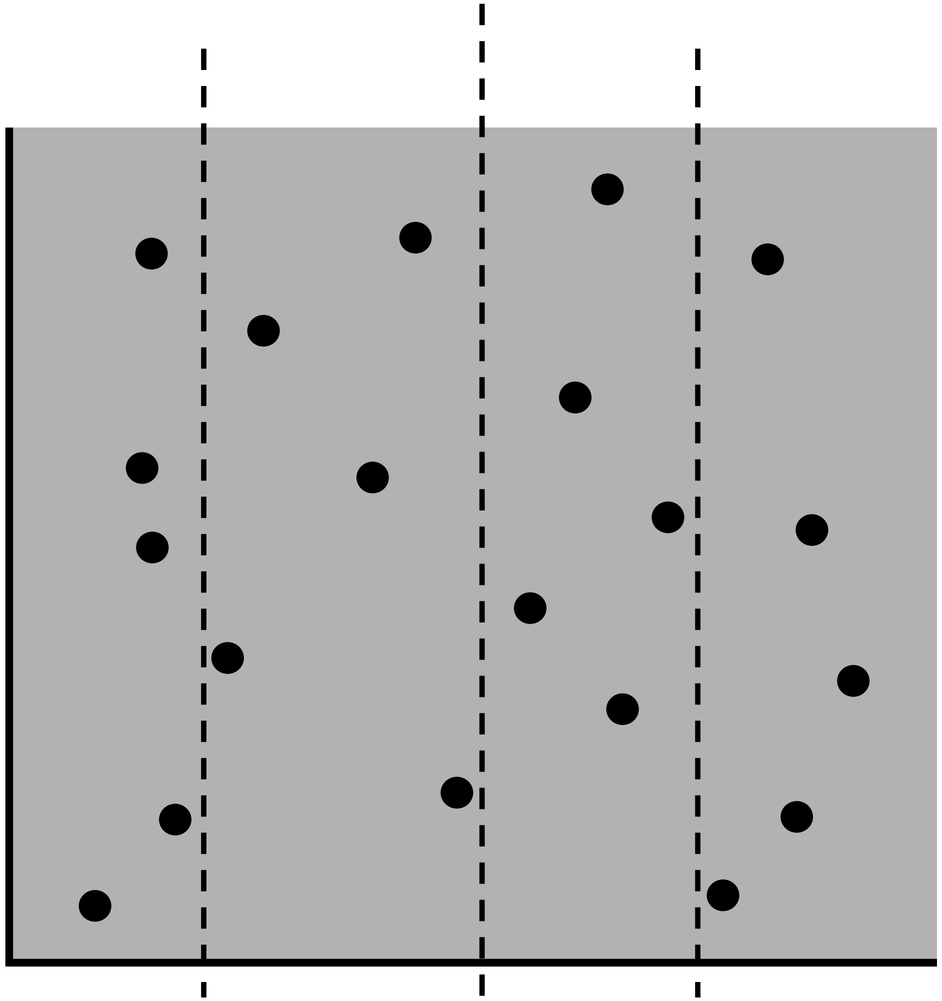


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until determining d_{left} and d_{right} is trivial.



Closest Pair Problem – Divide and Conquer

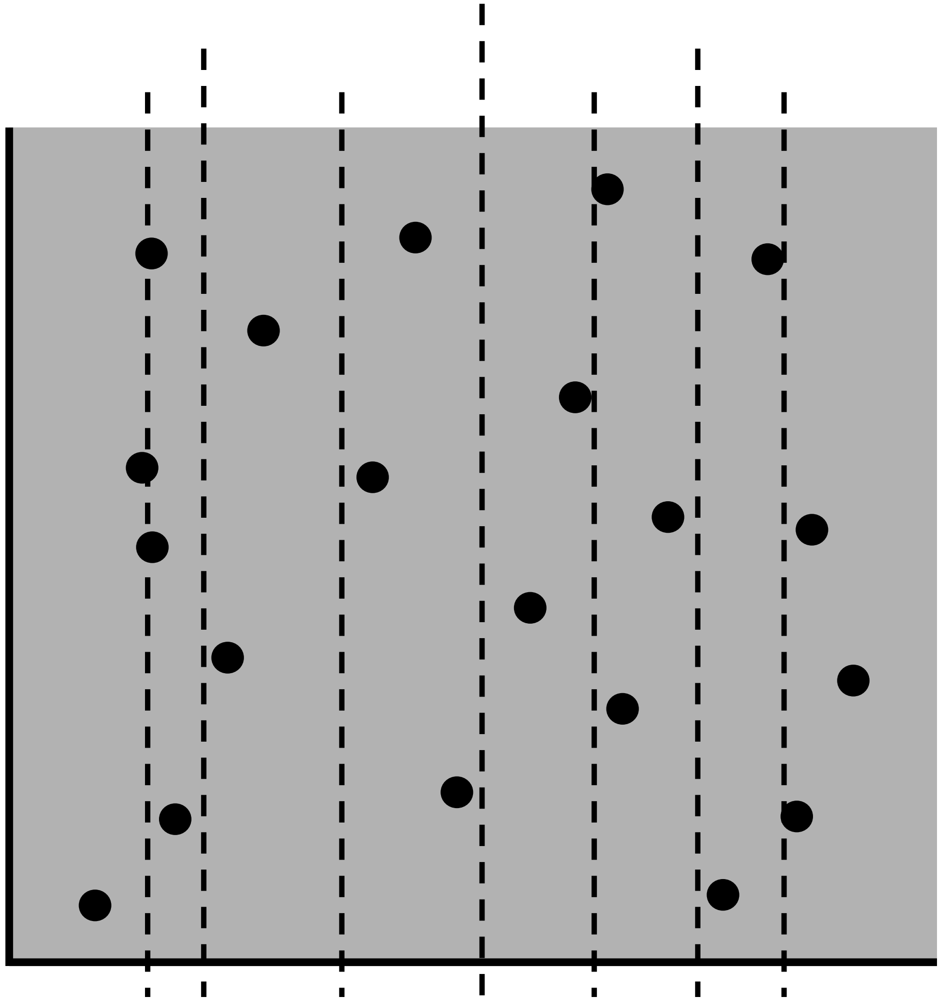


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until determining d_{left} and d_{right} is trivial.

Recursively find closest pairs on each side.

Closest Pair Problem – Divide and Conquer

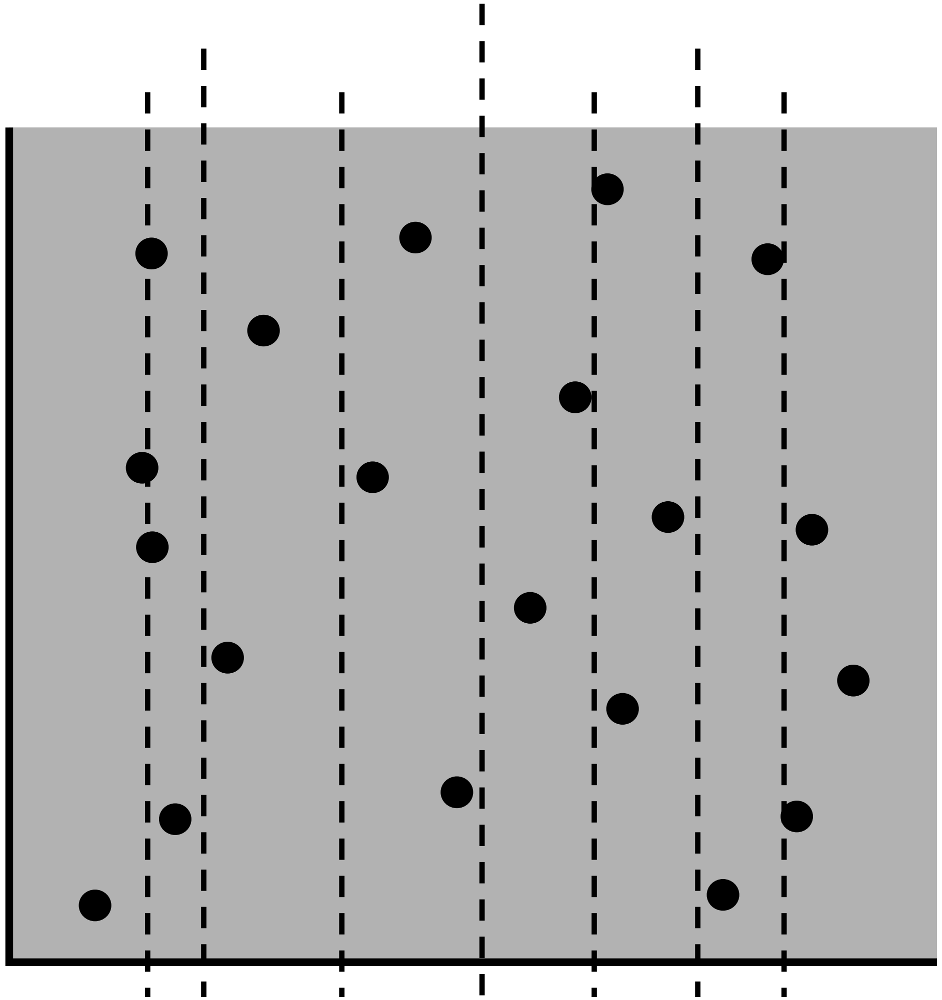


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until determining d_{left} and d_{right} is trivial.

Recursively find closest pairs on each side.

Closest Pair Problem – Divide and Conquer

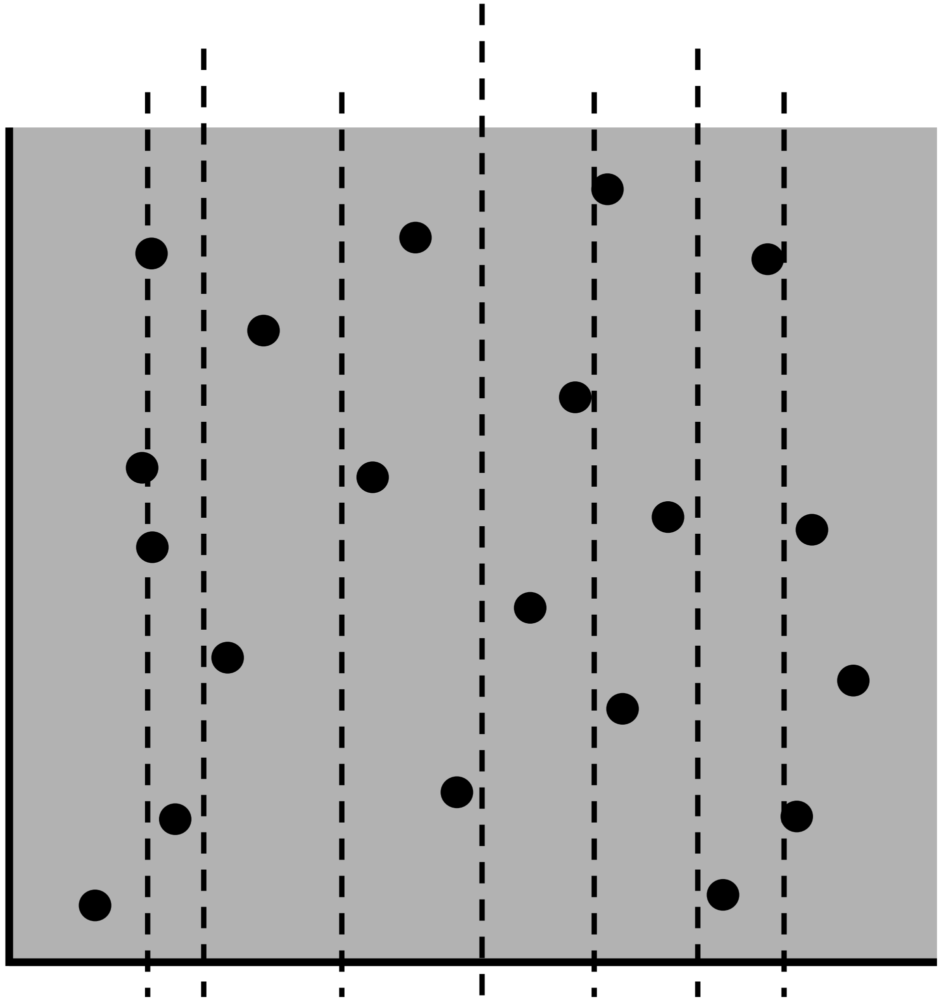


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until determining d_{left} and d_{right} is trivial.

When is finding d_{left} and d_{right} trivial?

Closest Pair Problem – Divide and Conquer



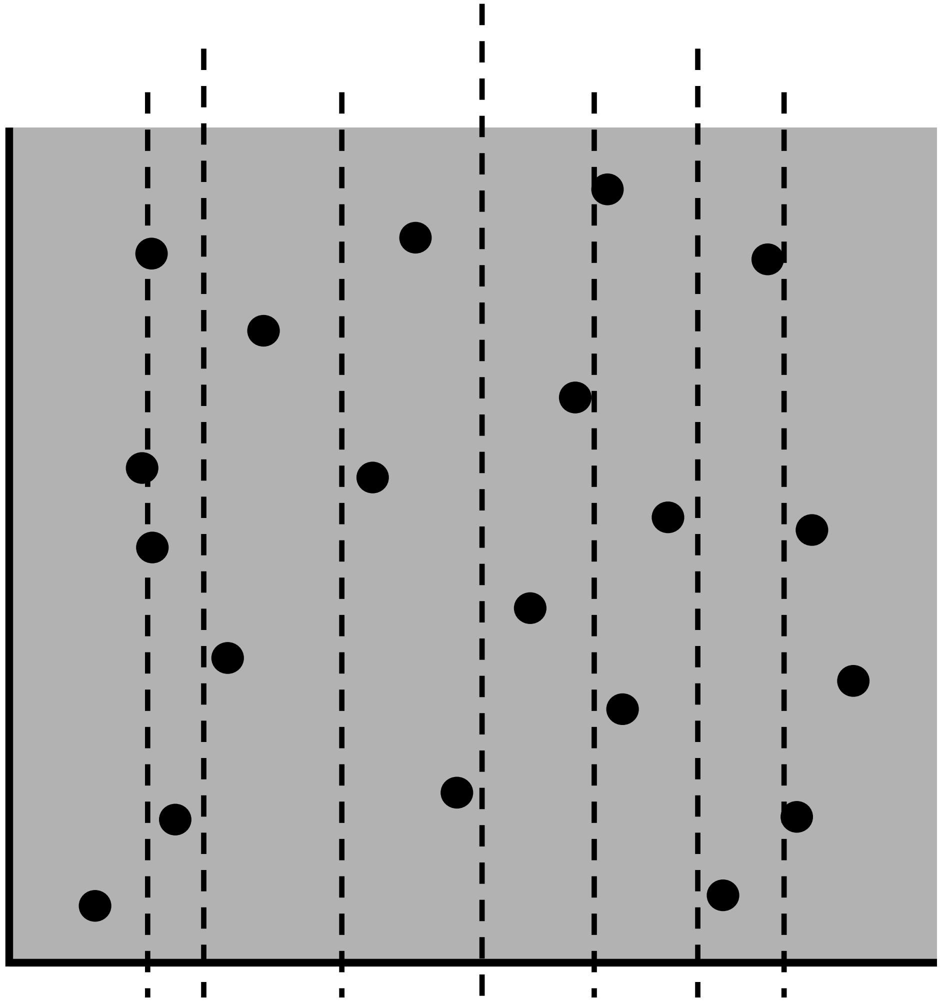
Recursive Process:

1. Divide points in half.
2. Repeat step 1 until determining d_{left} and d_{right} is trivial.

When is finding d_{left} and d_{right} trivial?

When there are one or two points on the left and right sides.

Closest Pair Problem – Divide and Conquer



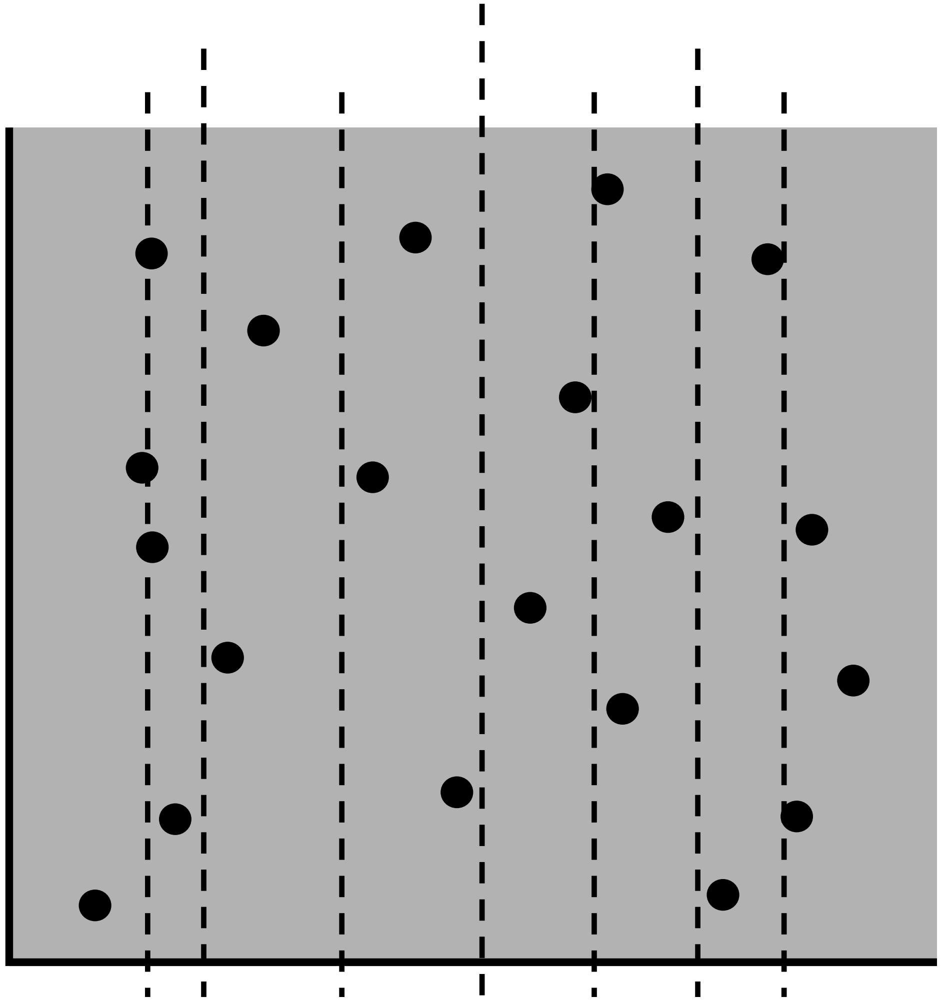
Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.

When is finding d_{left} and d_{right} trivial?

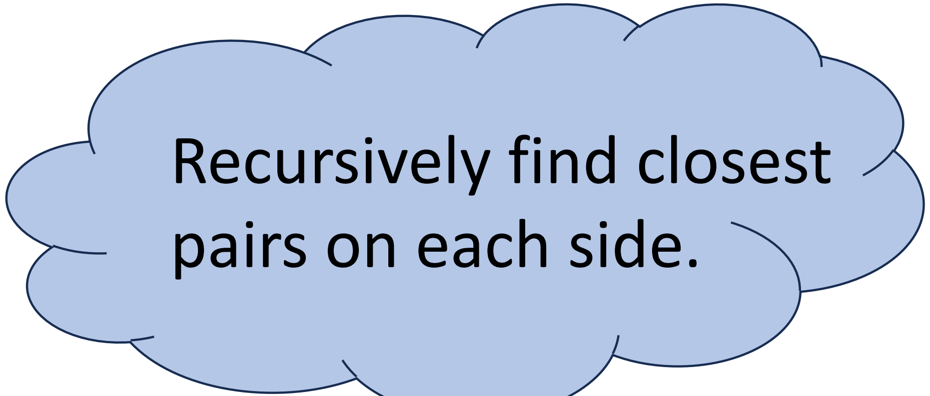
When there are one or two points on the left and right sides.

Closest Pair Problem – Divide and Conquer

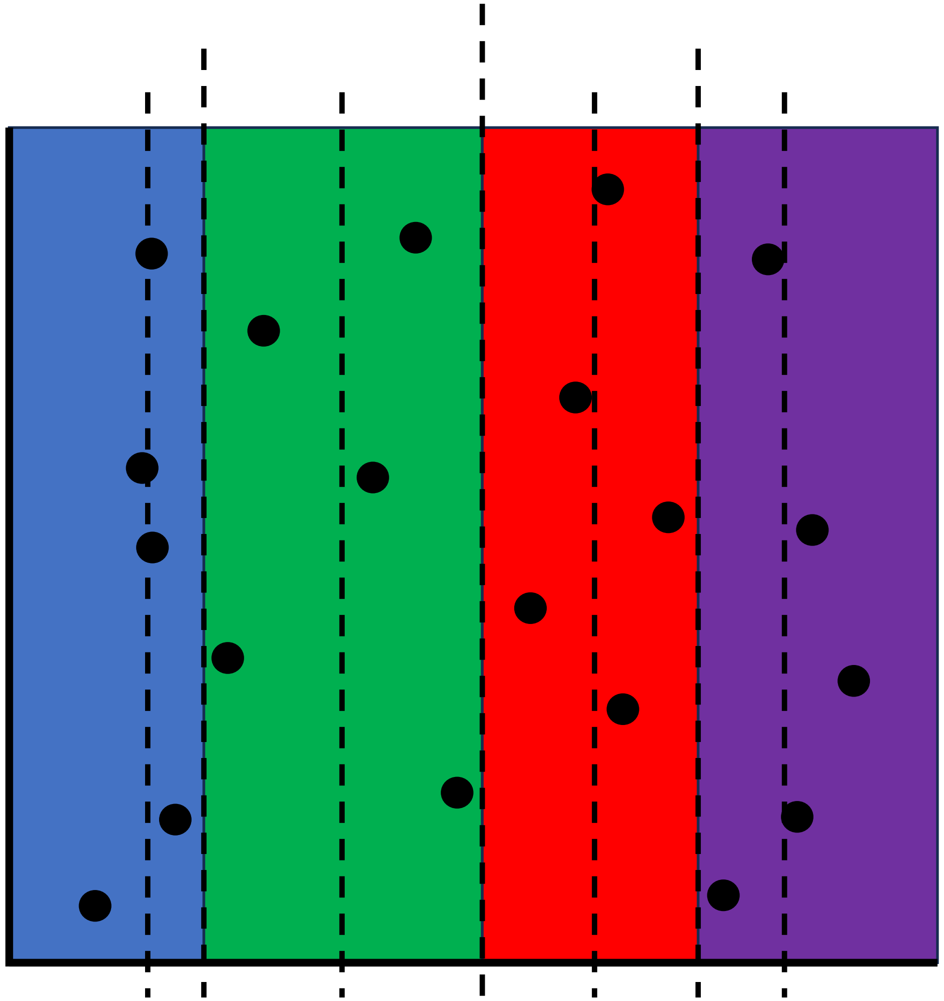


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.
3. Combine left and right sides to find closest of subproblems.



Closest Pair Problem – Divide and Conquer

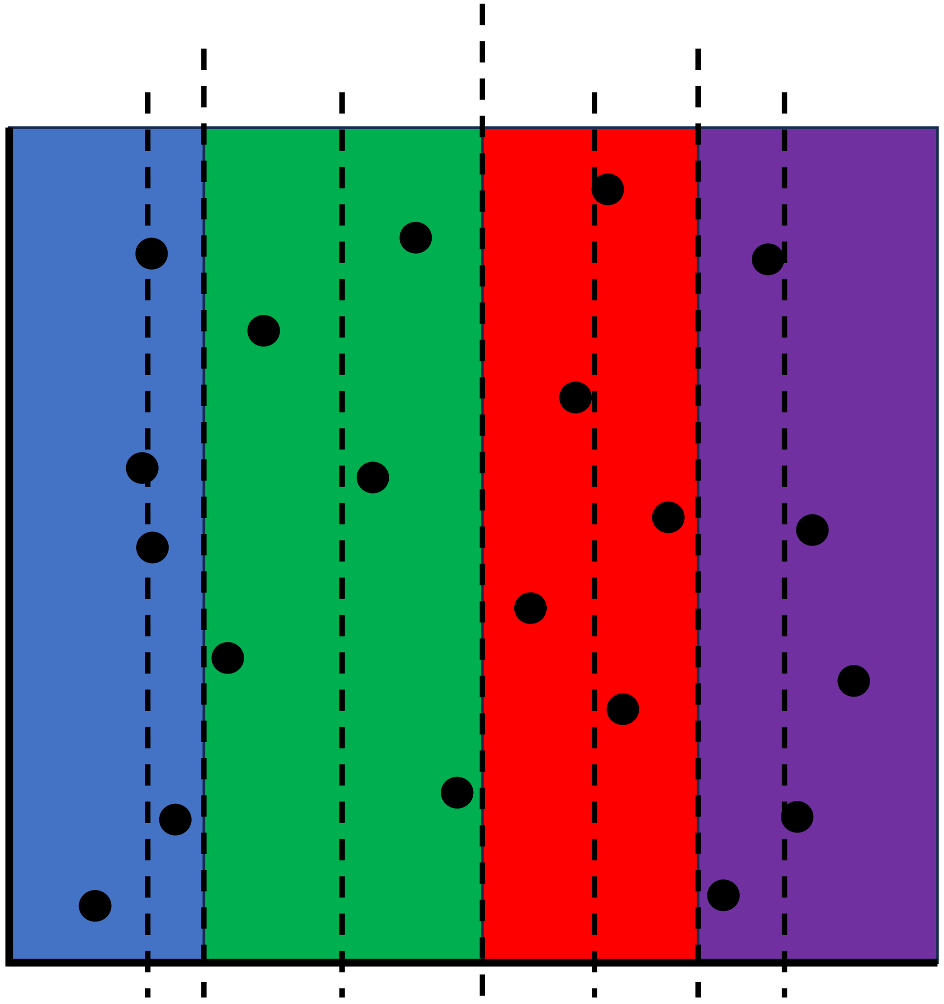


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.
3. Combine left and right sides to find closest of subproblems.

Recursively find closest pairs on each side.

Closest Pair Problem – Divide and Conquer

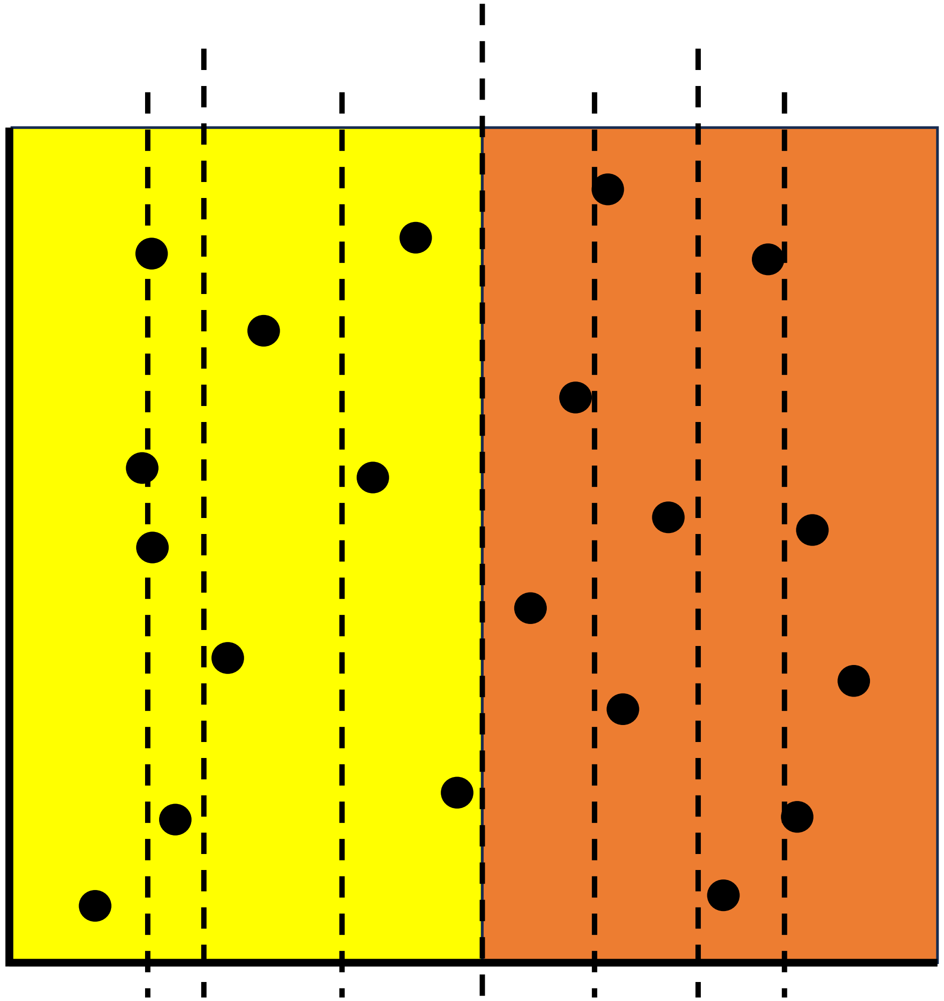


Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.
3. Combine left and right sides to find closest of subproblems.
4. Repeat until initial division is combined.

Recursively find closest pairs on each side.

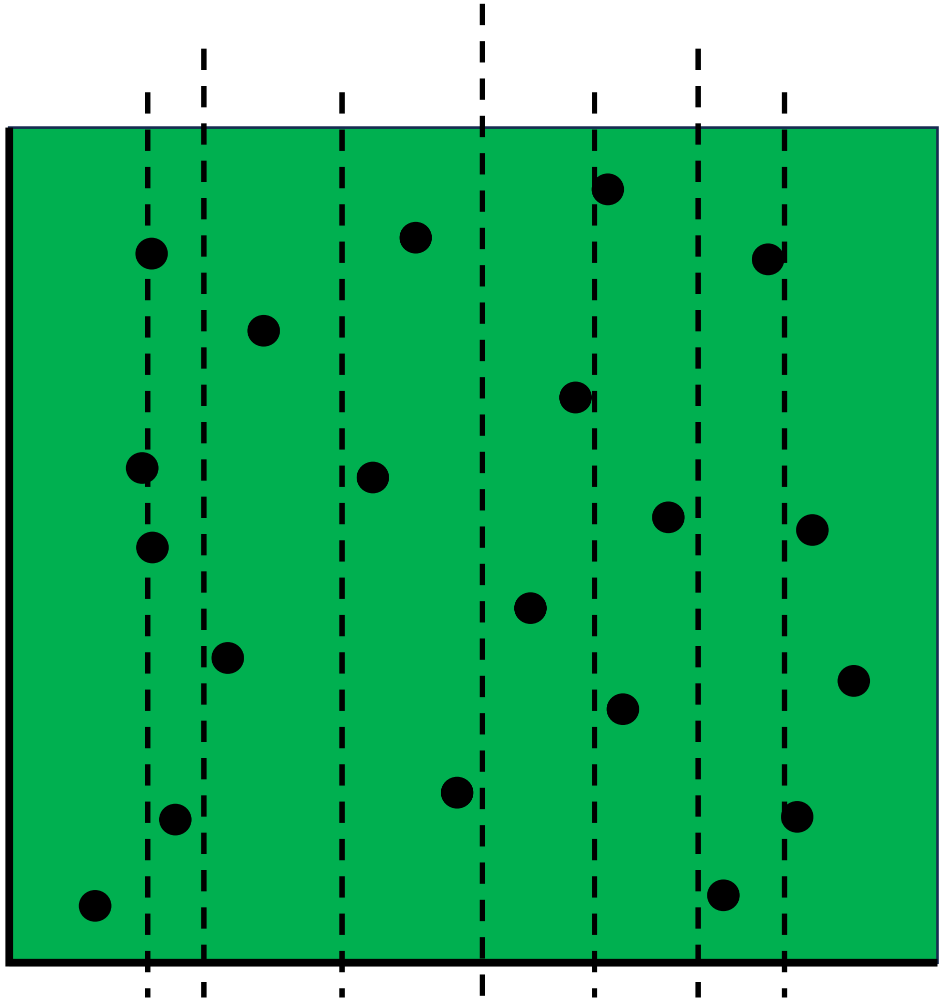
Closest Pair Problem – Divide and Conquer



Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.
3. Combine left and right sides to find closest of subproblems.
4. Repeat until initial division is combined.

Closest Pair Problem – Divide and Conquer



Recursive Process:

1. Divide points in half.
2. Repeat step 1 until there are only one or two points on each side.
3. Combine left and right sides to find closest of subproblems.
4. Repeat until initial division is combined.

Closest Pair Problem – Algorithm

`findClosestPair`(P):

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. **Determine d_{left} and d_{right}.**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

d_{left} = findClosestPair(P_{left})
d_{right} = findClosestPair(P_{right})

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Questions?

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Valid?

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Valid?

It's returning the distance between two points in P.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Optimal?

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Optimal?

If there was a closer pair, they would have been compared on the left side, right side, or as a straddle point.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}.
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Running Time?

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$.
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L.
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord.
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ.
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

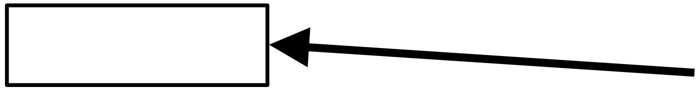
`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $O(n \log n)$
2. Determine d_{left} and d_{right}. **TBD**



How much work is done at the first level?

Closest Pair Problem – Algorithm

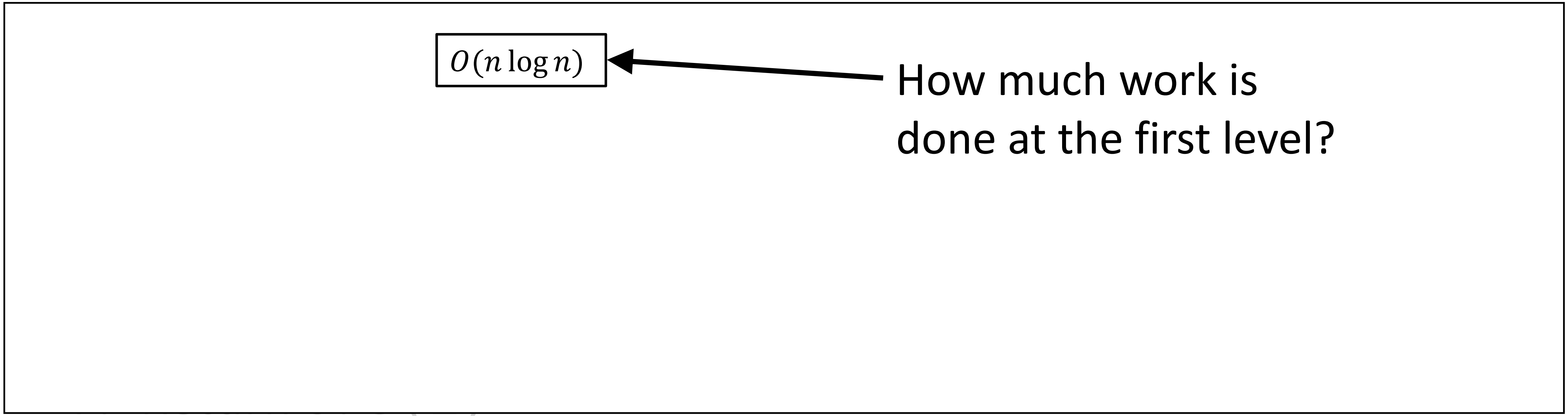
`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

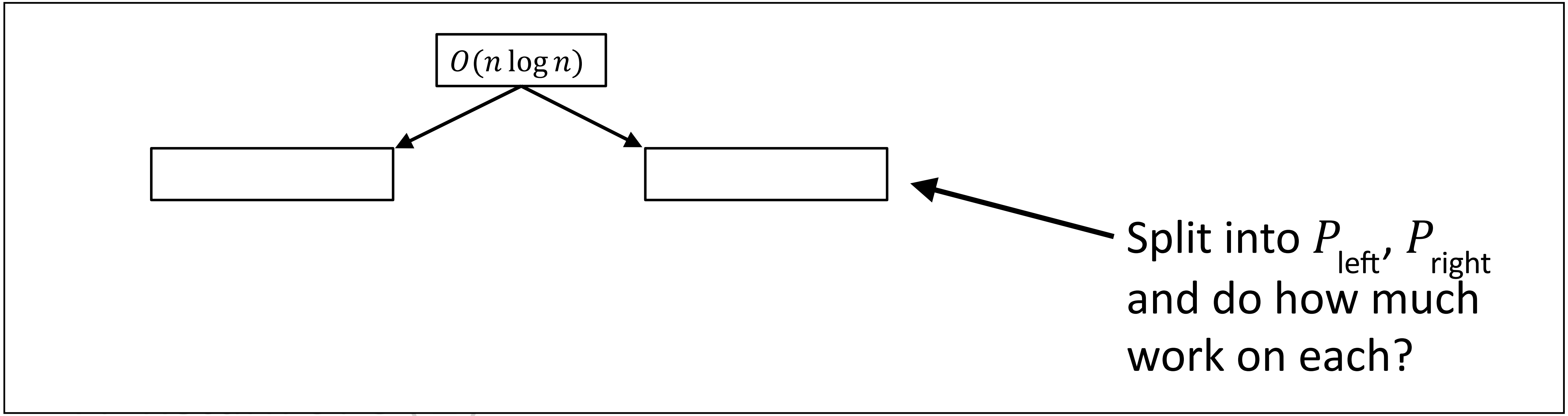
1. Sort points by x-coord, find L, make P_{left}, P_{right}. $O(n \log n)$
2. Determine d_{left} and d_{right}. **TBD**



Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

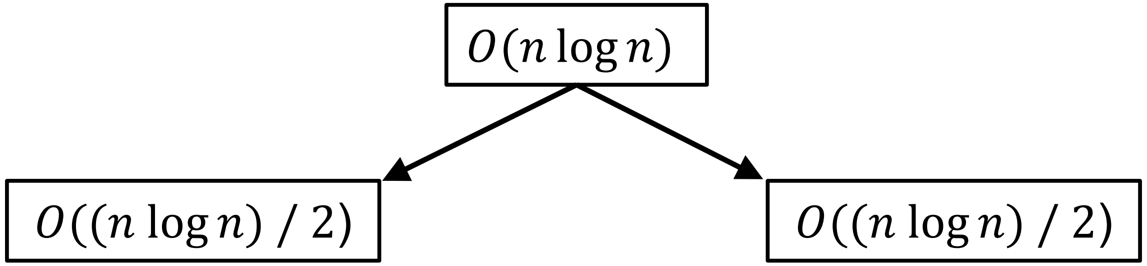
1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**



Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**

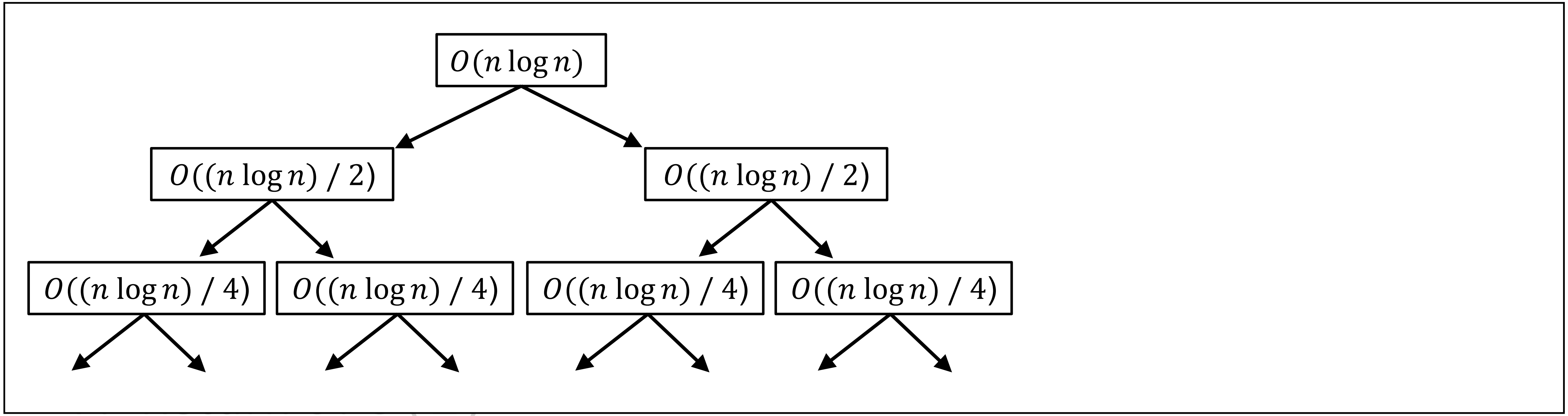


Split into P_{left}, P_{right} and do how much work on each?

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

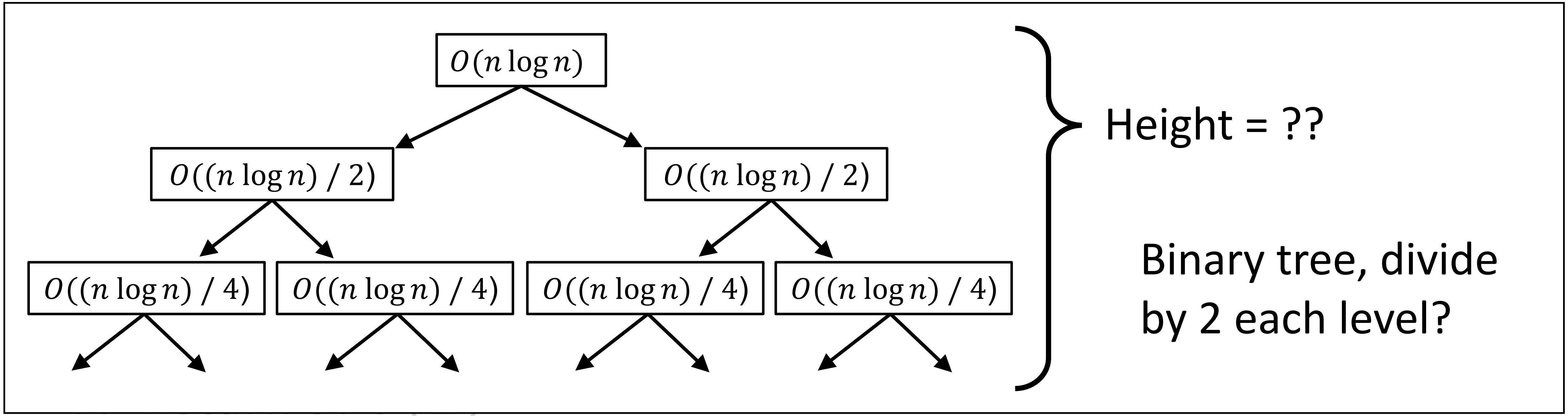
1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**



Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

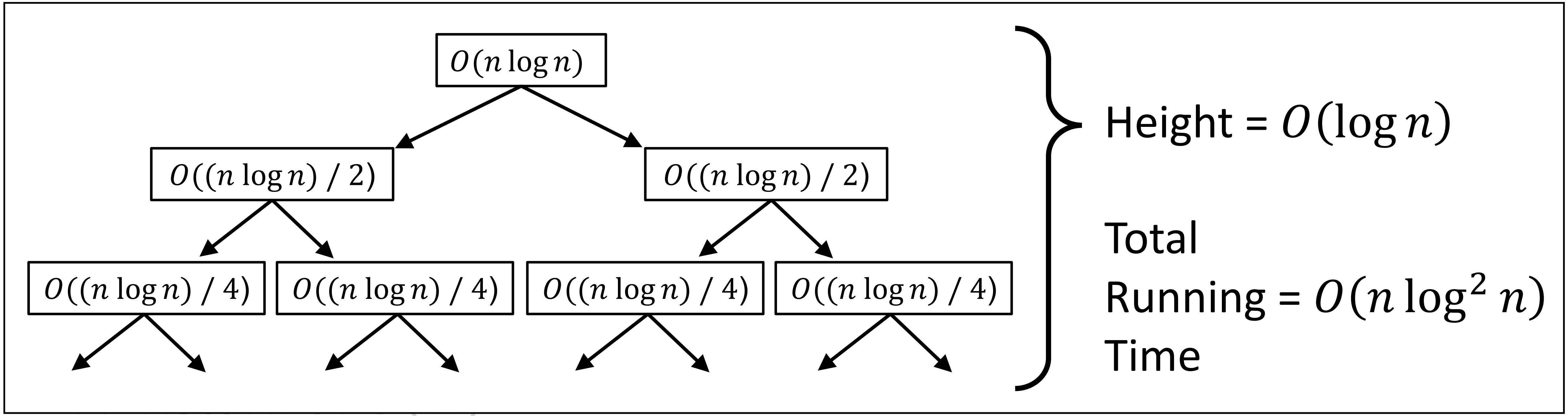
1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}. **TBD**



Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $O(n \log n)$
2. Determine d_{left} and d_{right}. **TBD**

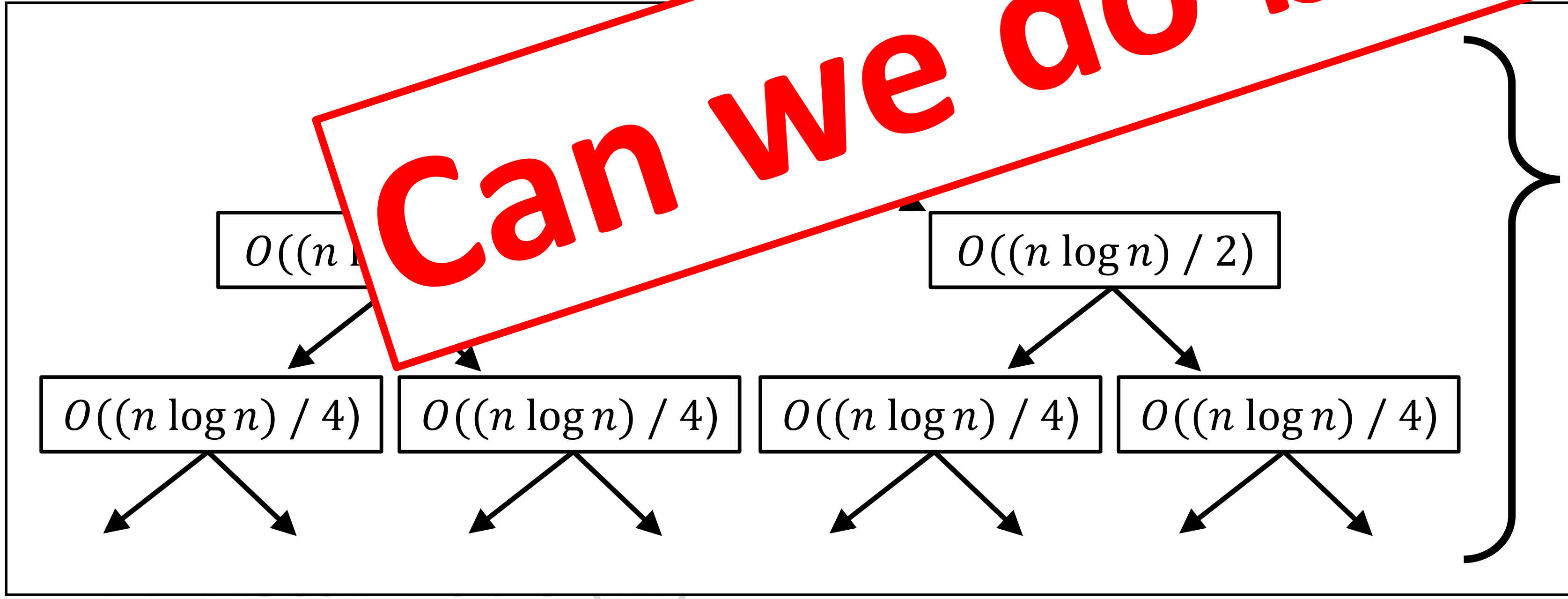


Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P … $\log n)$
2. Determine d_{left} and d …

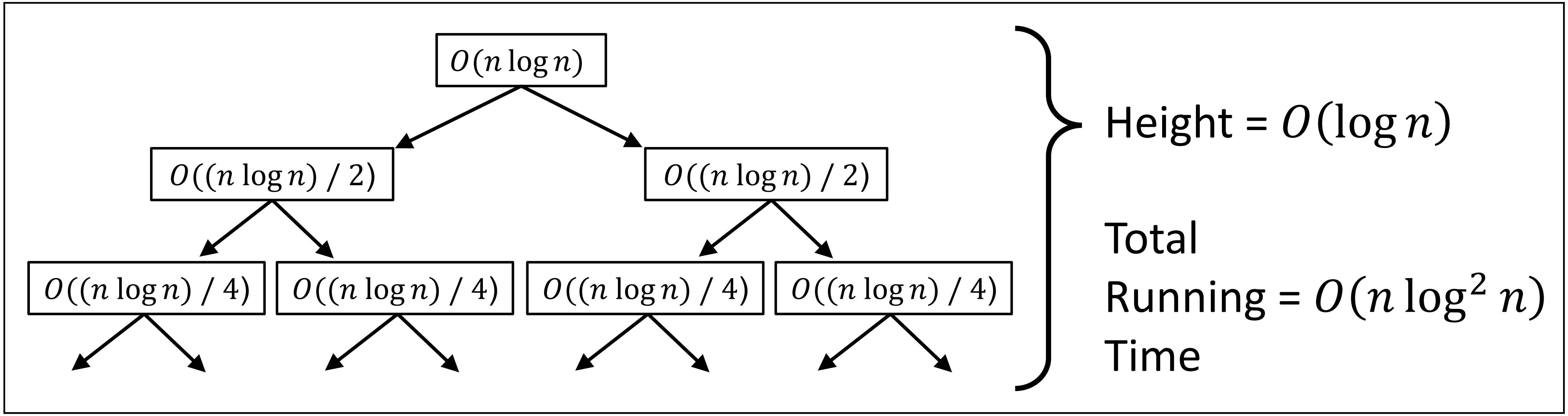
Can we do better?



Height = $O(\log n)$

Total Running Time = $O(n \log^2 n)$

Closest Pair Problem – Algorithm



Option 1: (Significantly) Reduce the height of the recursion tree.

Option 2: (Significantly) Reduce the amount of work done at each level.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

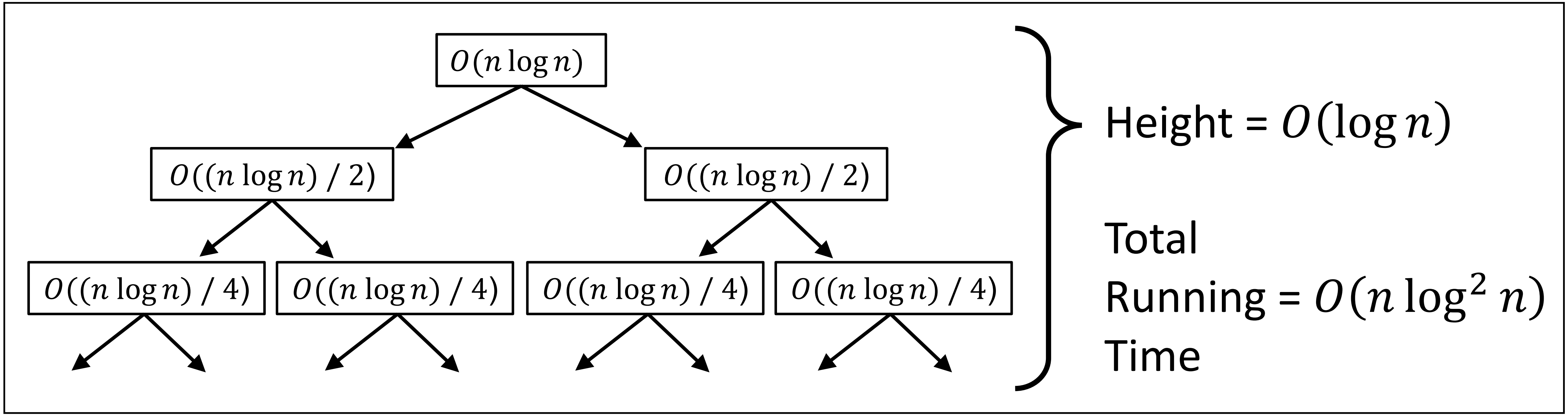
Closest Pair Problem – Algorithm

`findClosestPair`(P):

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Maybe we don't need to sort so often??

Closest Pair Problem – Algorithm



Plan:

- Presort by x-coordinate (X)
- Presort by y-coordinate (Y)
- Split X and Y by comparing to L.

Closest Pair Problem – Algorithm

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y).

```
findClosestPair(P):
```

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n\,log\,n)}$

findClosestPair(P):

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n\ log\ n)}$

`findClosestPair(`P`):`

1. Sort points by x-coord, find L, make P_{left}, P_{right}. $\boldsymbol{O(n \log n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n\ log\ n)}$

```
findClosestPair(P):
```

1. ~~Sort points by x-coord,~~ find L, split X, Y. $\boldsymbol{O(n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n \log n)}$

```
findClosestPair(P):
```

1. ~~Sort points by x-coord,~~ find L, split X, Y. $\boldsymbol{O(n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. Sort S by y-coord. $\boldsymbol{O(n \log n)}$
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n \log n)}$

```
findClosestPair(P):
```

1. ~~Sort points by x-coord,~~ find L, split X, Y. $\boldsymbol{O(n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. ~~Sort S by y-coord. $\boldsymbol{O(n \log n)}$~~
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

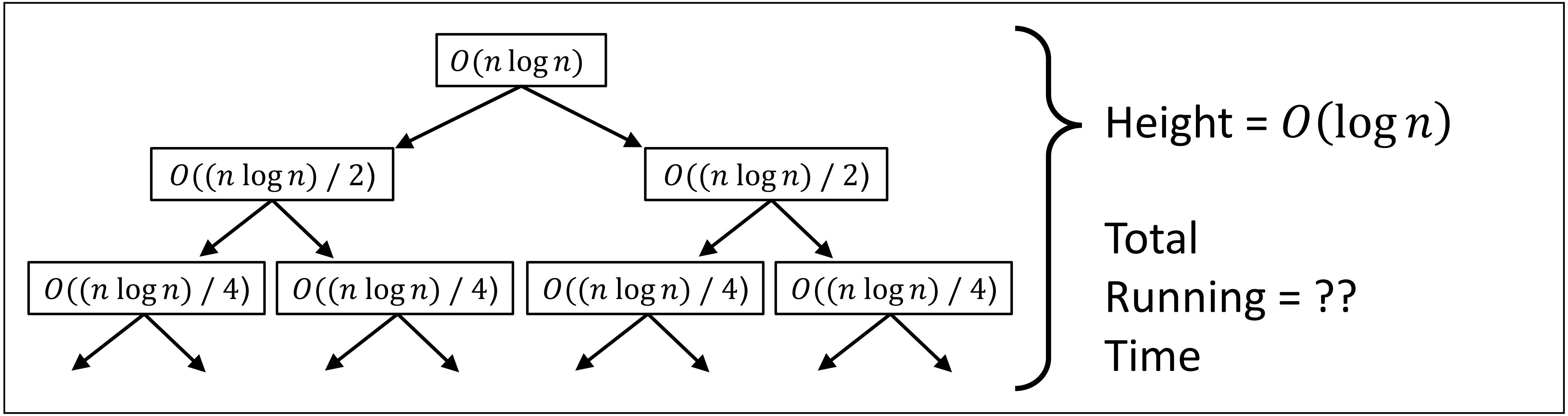
Closest Pair Problem – Algorithm

0. Sort points by x-coordinate (X) and y-coordinate (Y). $\boldsymbol{O(n \log n)}$

```
findClosestPair(P):
```

1. ~~Sort points by x-coord,~~ find L, split X, Y. $\boldsymbol{O(n)}$
2. Determine d_{left} and d_{right}.
3. Let $\delta = \min(d_{\text{left}}, d_{\text{right}})$. $\boldsymbol{O(1)}$
4. Let S be straddle points within δ of L. $\boldsymbol{O(n)}$
5. ~~Sort S by y-coord. $\boldsymbol{O(n \log n)}$~~
6. Compare points in S to next 11 points and update δ. $\boldsymbol{O(n)}$
7. Return δ. $\boldsymbol{O(1)}$

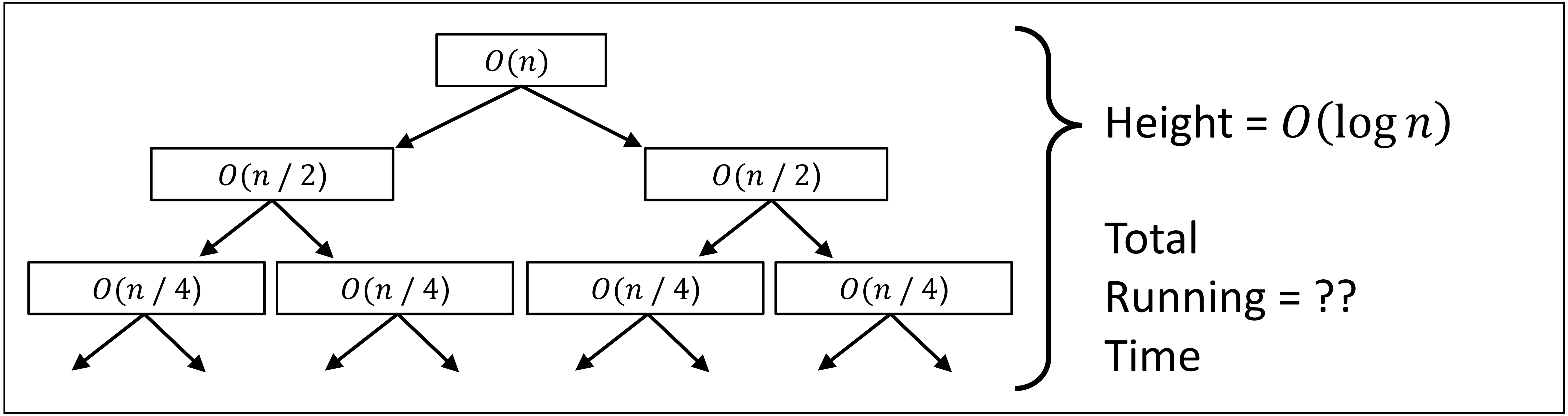
Closest Pair Problem – Algorithm



Plan:

- Presort by x-coordinate (X)
- Presort by y-coordinate (Y)
- Split X and Y by comparing to L.

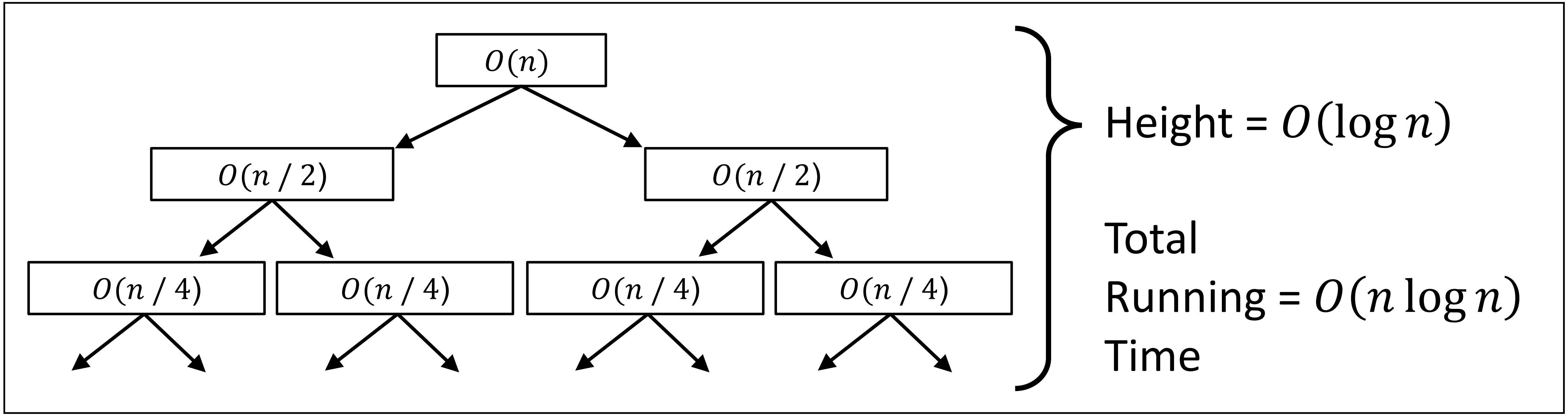
Closest Pair Problem – Algorithm



Plan:

- Presort by x-coordinate (X)
- Presort by y-coordinate (Y)
- Split X and Y by comparing to L.

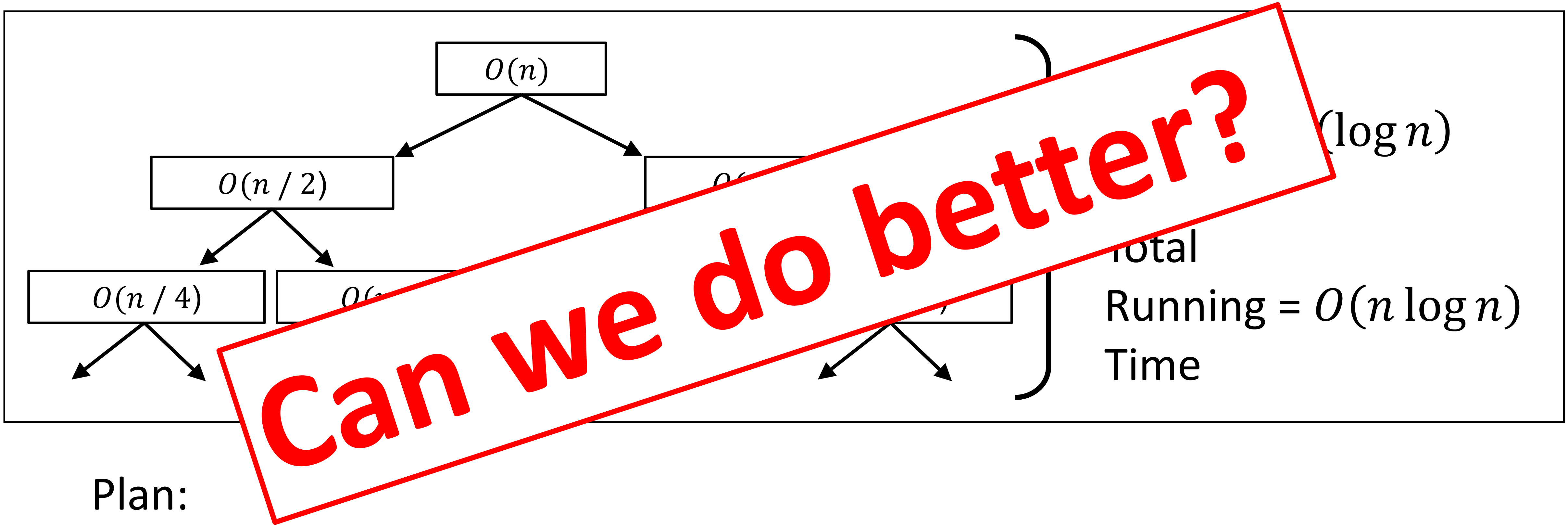
Closest Pair Problem – Algorithm



Plan:

- Presort by x-coordinate (X)
- Presort by y-coordinate (Y)
- Split X and Y by comparing to L.

Closest Pair Problem – Algorithm



Plan:

- Presort by x-coordinate (X)
- Presort by y-coordinate (Y)
- Split X and Y by comparing to L.

Dynamic Programing

Trick-or-treat planning.

Dynamic Programing

Trick-or-treating at the red house, blue house, and green house are the fewest stops you need to fill your 25-pound capacity sack.

Dynamic Programing

Trick-or-treating at the red house, blue house, and green house are the fewest stops you need to fill your 25-pound capacity sack. The blue house gives you 5 pounds of candy. What can you conclude?

Dynamic Programing

Trick-or-treating at the red house, blue house, and green house are the fewest stops you need to fill your 25-pound capacity sack. The blue house gives you 5 pounds of candy. What can you conclude?

The red house and the green house are the fewest stops you need to get 20+ pounds of candy.

Dynamic Programing

Trick-or-treating at the red house, blue house, and green house are the fewest stops you need to fill your 25-pound capacity sack. The blue house gives you 5 pounds of candy. What can you conclude?

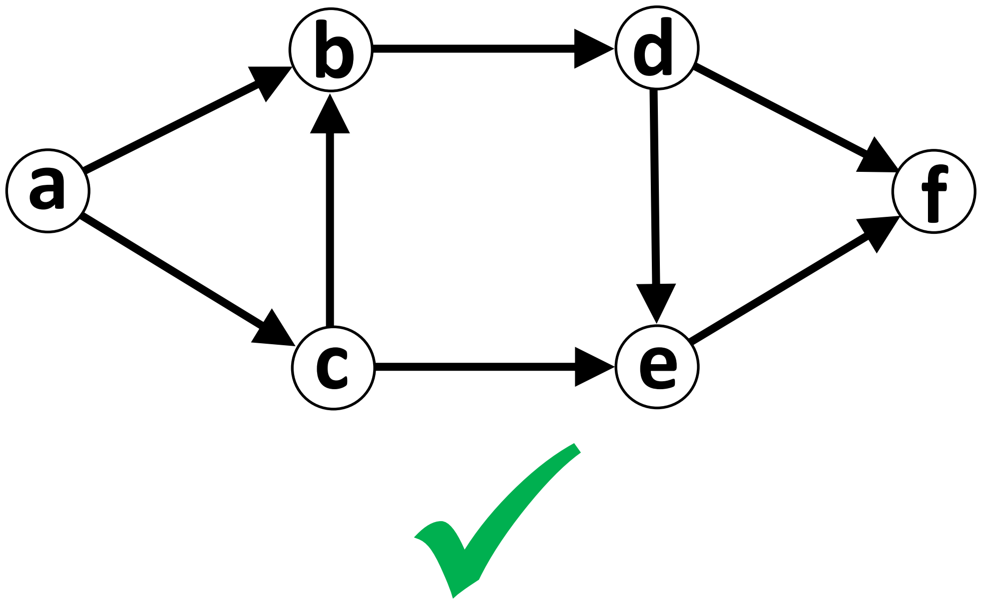
The red house and the green house are the fewest stops you need to get 20+ pounds of candy.

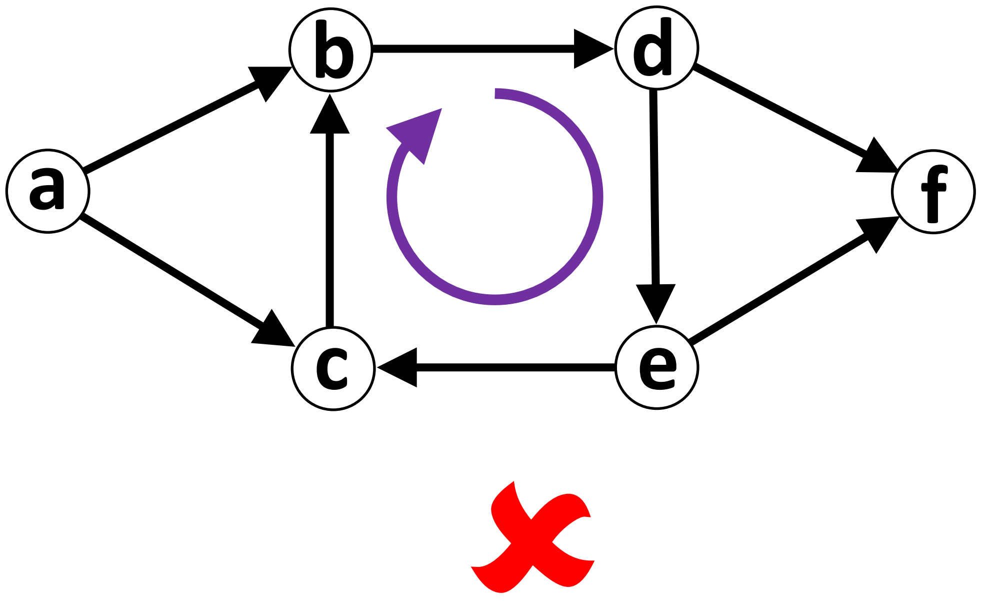
A problem exhibits optimal substructure if removing part of an optimal solution results in an optimal solution to a smaller problem.

Central tenant of Dynamic Programming: Leverage optimal sub-structure.

Directed Acyclic Graph (DAG)

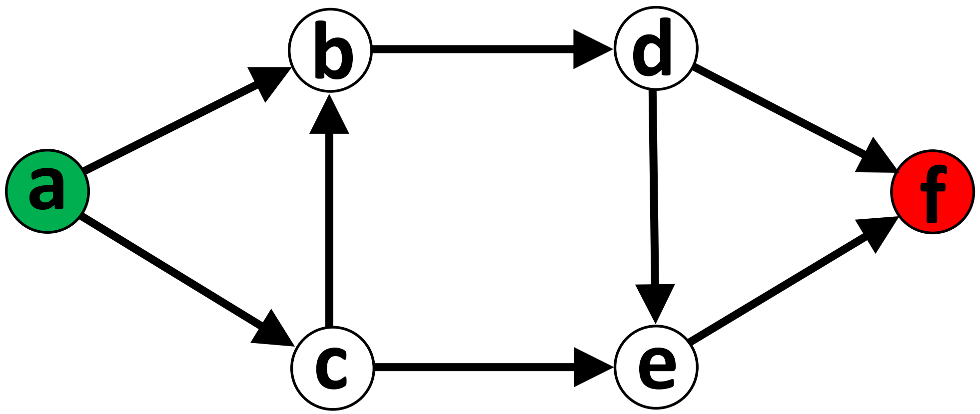
Directed Acyclic Graph (DAG) = Directed graph with no cycles.





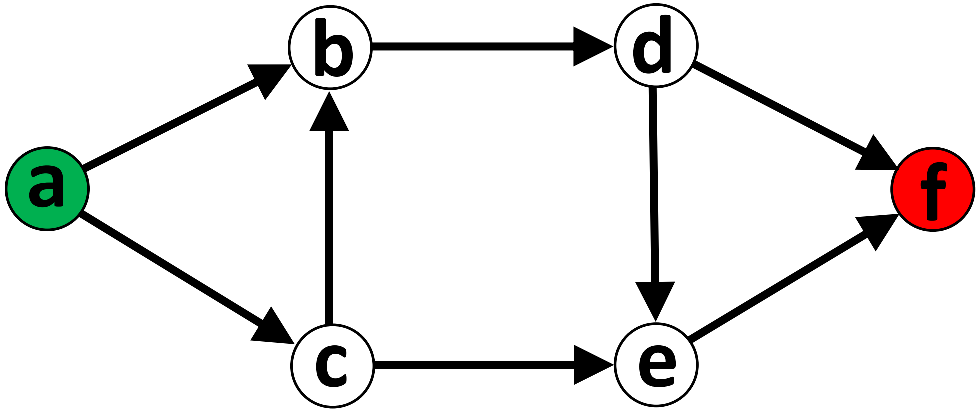
Longest Path in a DAG

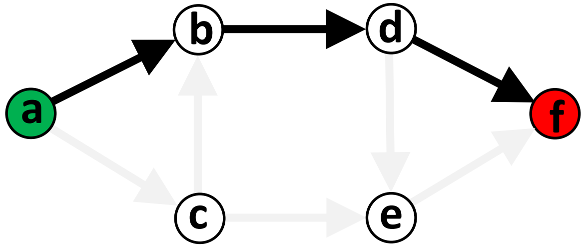
Given a DAG, find the longest path between any two vertices in the graph.



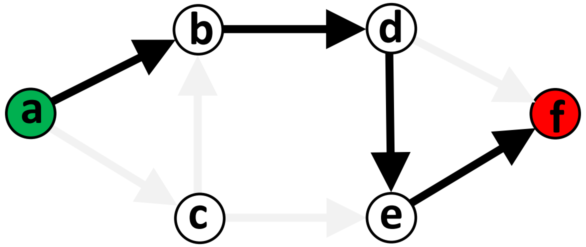
Longest Path in a DAG

Given a DAG, find the longest path between any two vertices in the graph.

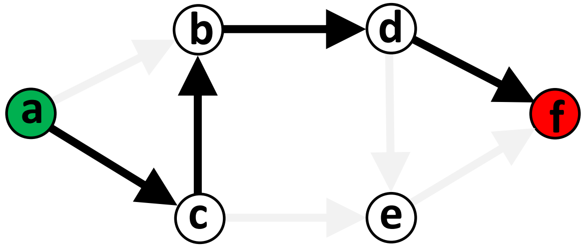




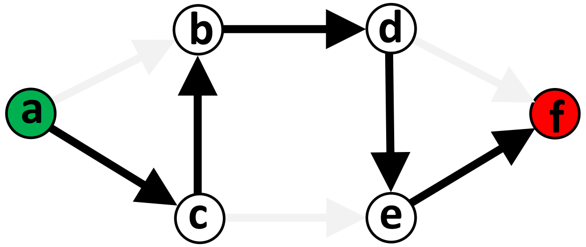
length = 3



length = 4



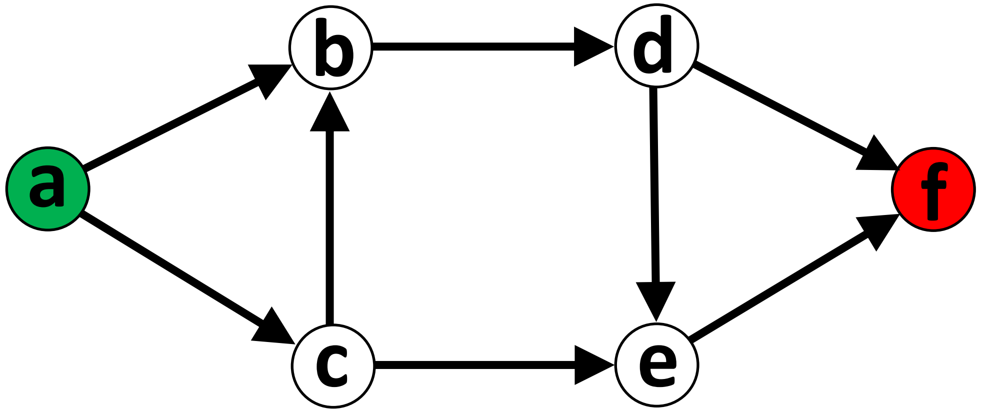
length = 4



length = 5

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day

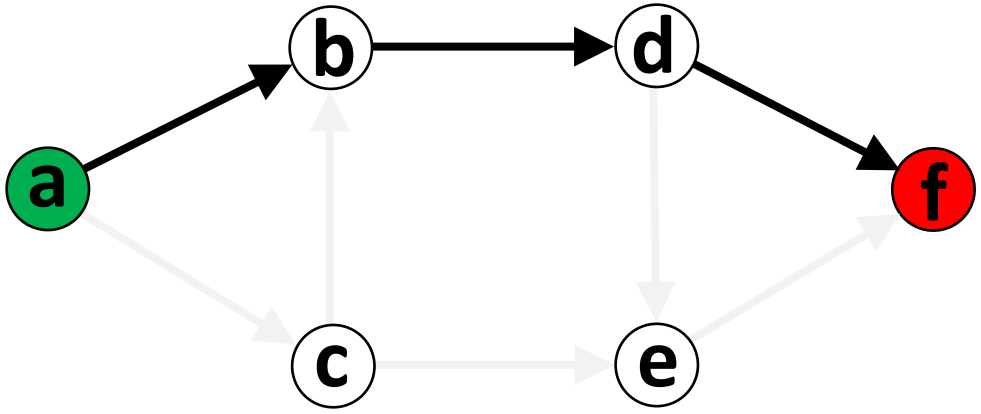


Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day

Length = 2 + 4 + 2 + 1 = 9 days

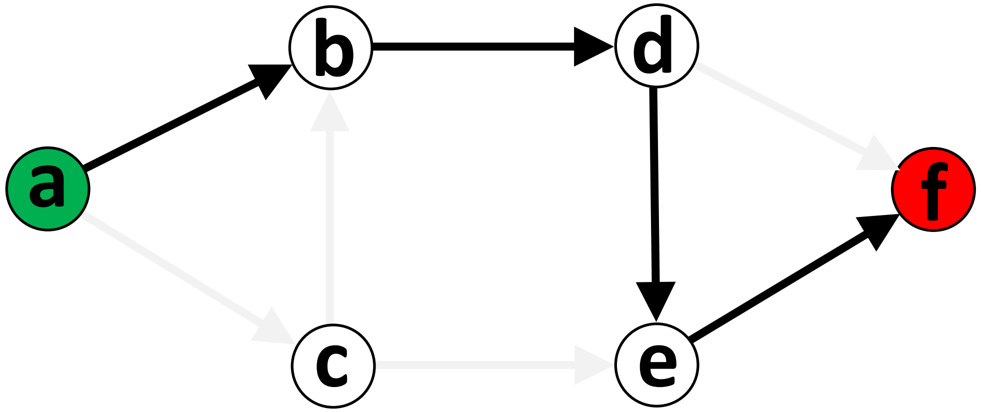


Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day

Length = 2 + 4 + 2 + 1 +1 = 10 days

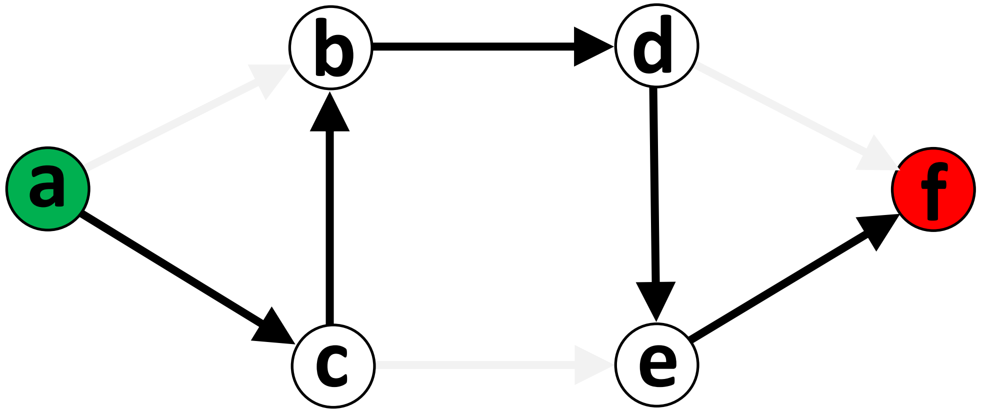


Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day

Length = 2 + 1 + 4 + 2 + 1 +1 = 11 days

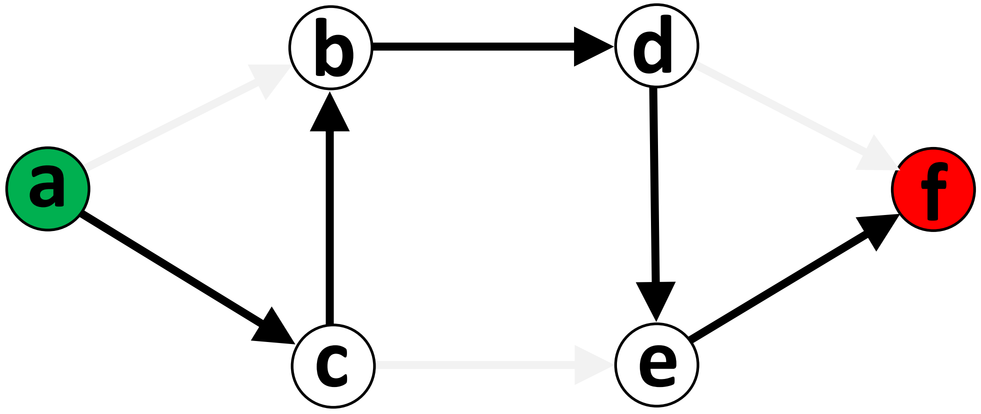


Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day

Length = 2 + 1 + 4 + 2 + 1 +1 = 11 days

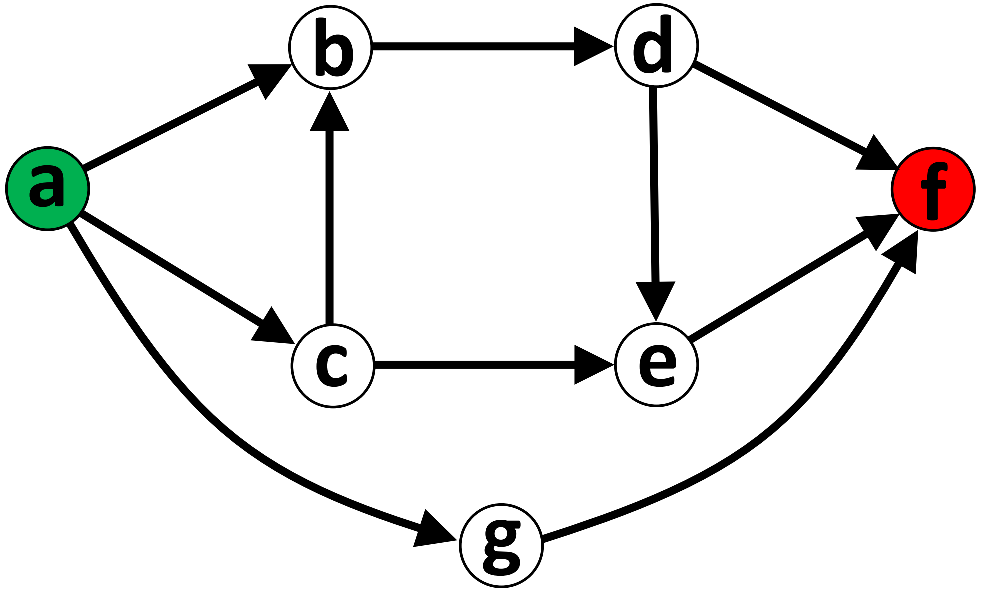


Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

If any task on critical path is delayed, project is delayed.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day
g	Photograph venue	1 day



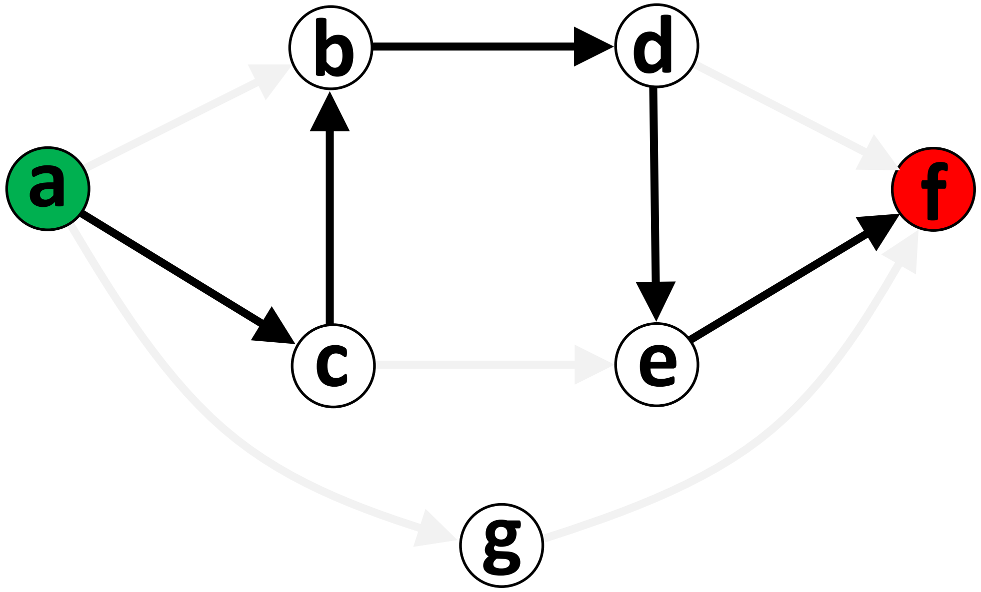
Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

If any task on critical path is delayed, project is delayed.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day
g	Photograph venue	1 day

Length = 2 + 1 + 4 + 2 + 1 +1 = 11 days



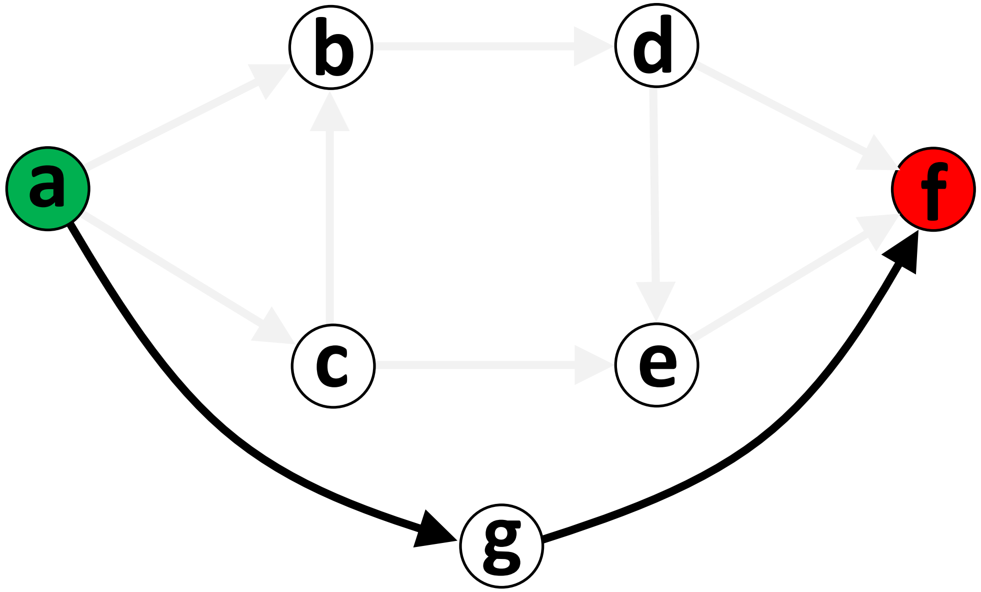
Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

If any task on critical path is delayed, project is delayed.

Find the Longest Path in a DAG

Task	Description	Duration
a	Select location	2 days
b	Get permits	4 days
c	Select date/time	1 day
d	Hire vendors	2 days
e	Make flyers	1 day
f	Market event	1 day
g	Photograph venue	**10 days**

Length = 2 + 10 +1 = 13 days



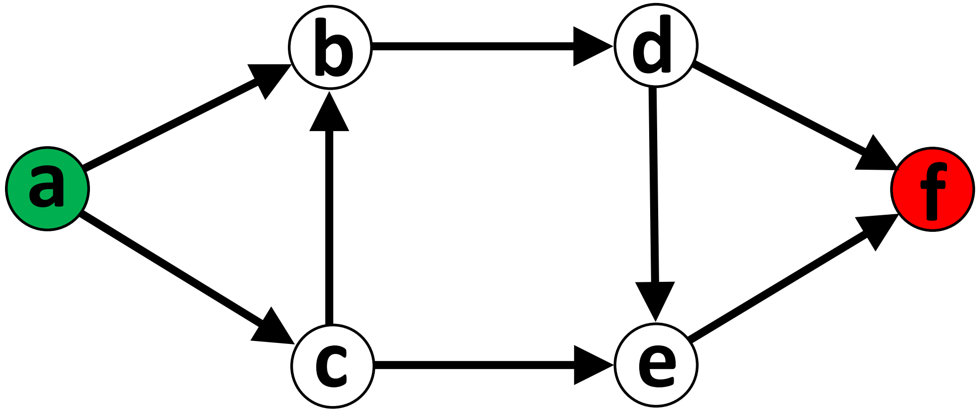
Critical Path: Sequence of dependent tasks that determines the minimum time to complete project.

If any task on critical path is delayed, project is delayed.

Find the Longest Path in a DAG

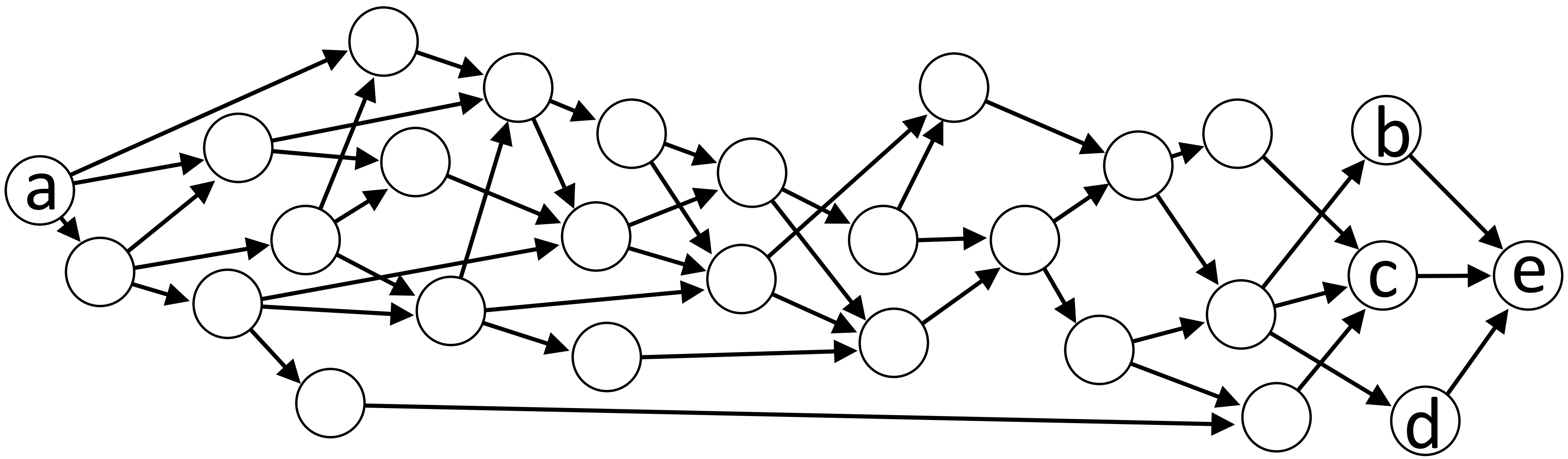
Number of edges.
No vertex weights.

Given a DAG, find the longest path between any two vertices in the graph.



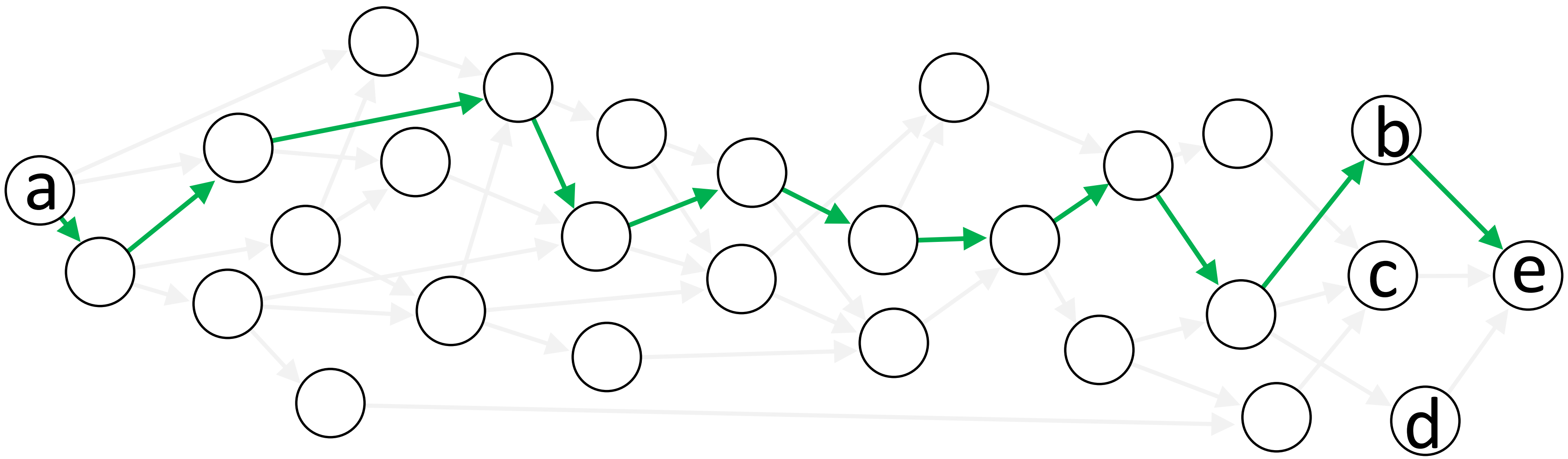
How do we do this?

Find the Longest Path in a DAG



Interesting observations?

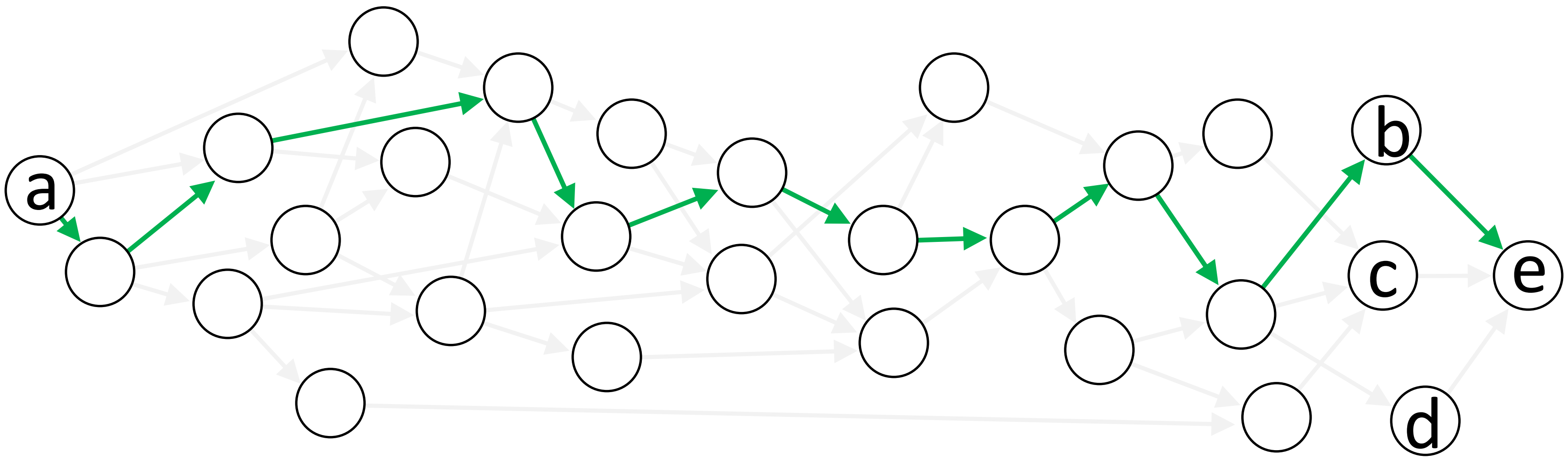
Find the Longest Path in a DAG



Interesting observations?

If the longest path goes from **a** to **e** and passes through **b**, what could we say about the part of that path to **b**?

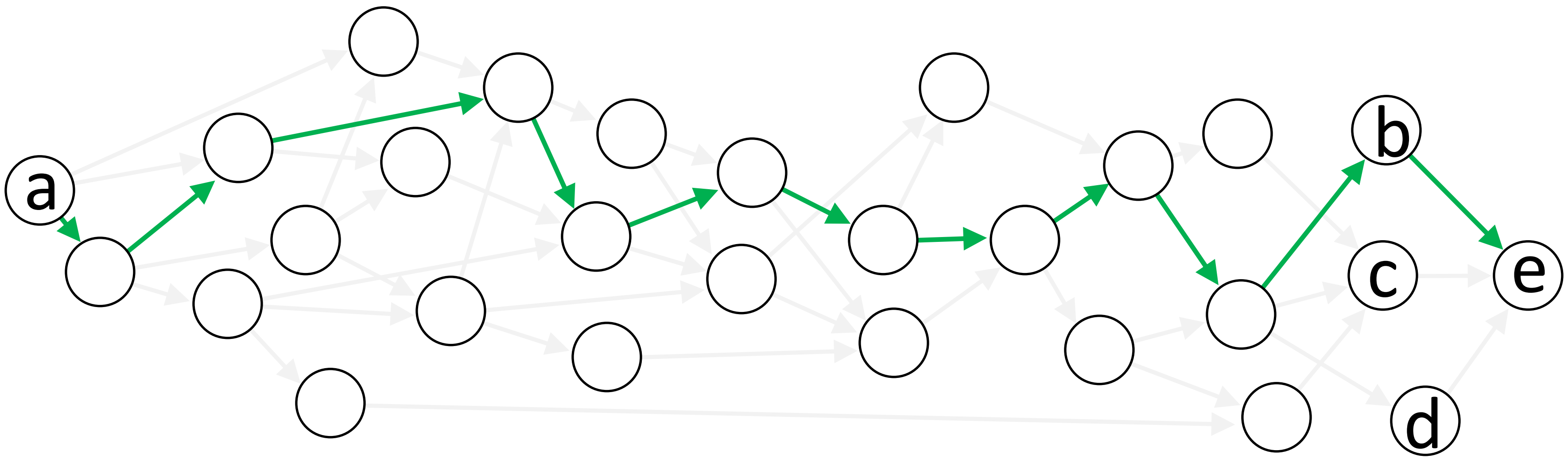
Find the Longest Path in a DAG



Interesting observations?

If the longest path goes from **a** to **e** and passes through **b**, that must be the longest path that ends at **b**.

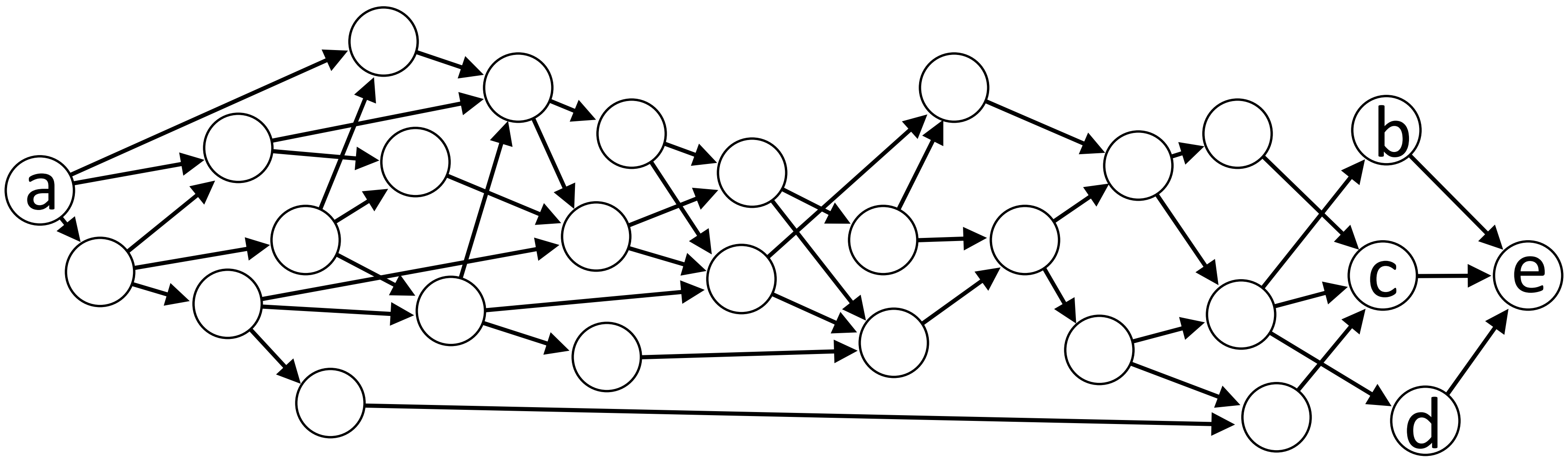
Find the Longest Path in a DAG



Interesting observations?

If the longest path goes from **a** to **e** and passes through **b**, that must be the longest path that ends at **b**. If not, then we could make a longer path.

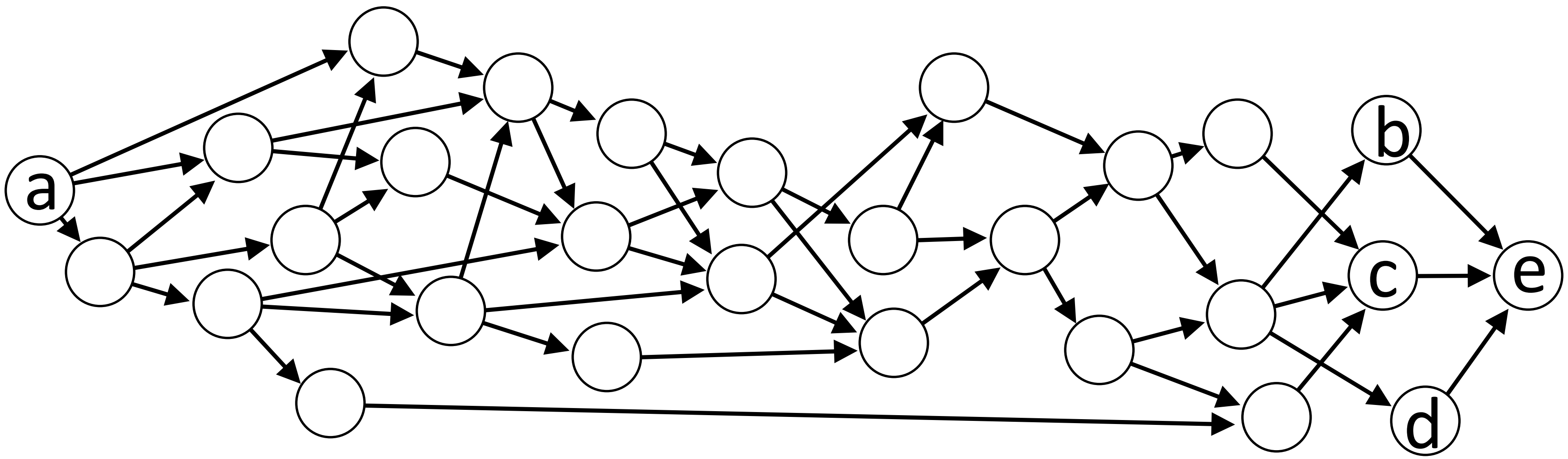
Find the Longest Path in a DAG



Interesting observations?

The longest path to e = ??

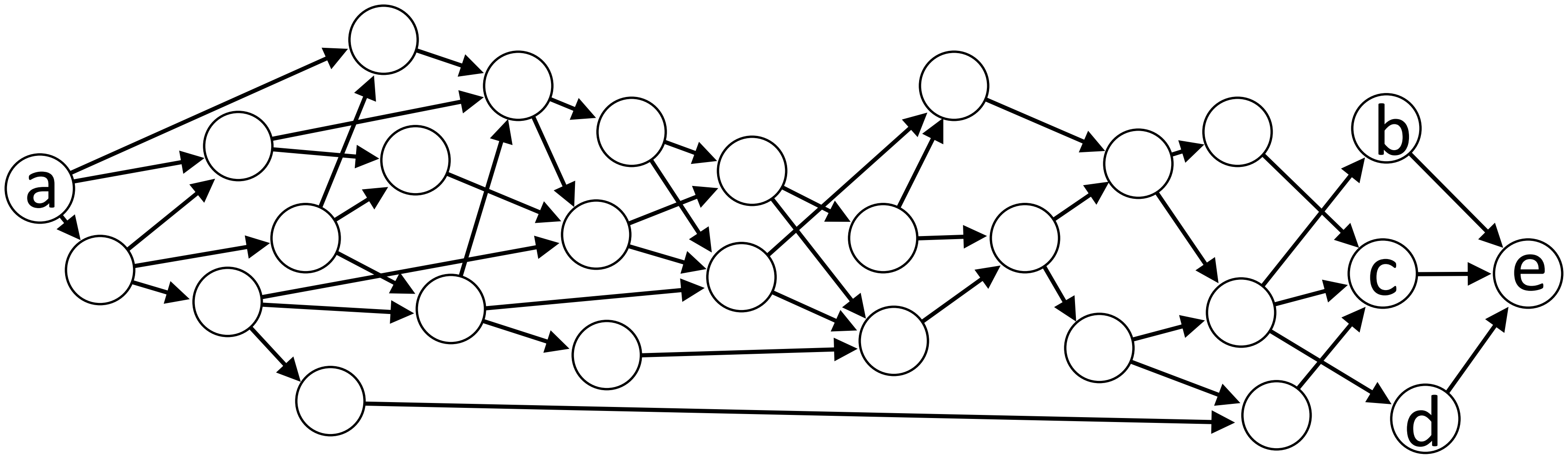
Find the Longest Path in a DAG



Interesting observations?

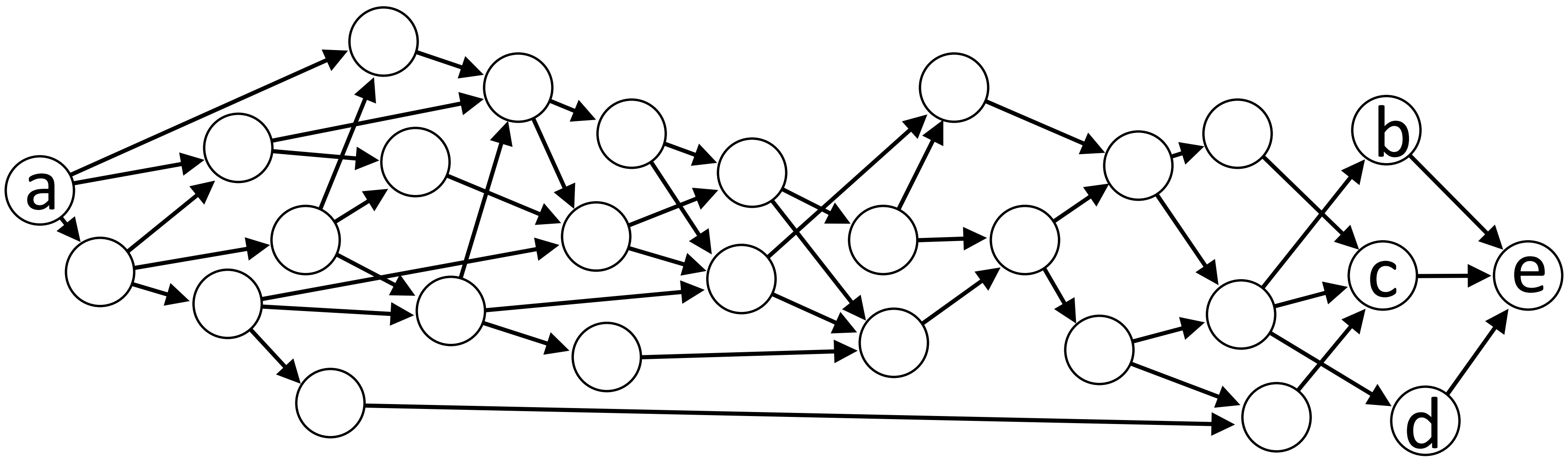
$$\text{The longest path to e} = \max\begin{pmatrix}\text{longest path to b}\\ \text{longest path to c}\\ \text{longest path to d}\end{pmatrix} + 1$$

Find the Longest Path in a DAG



$$\text{longest path to e} = \max\begin{pmatrix}\text{longest path to b}\\ \text{longest path to c}\\ \text{longest path to d}\end{pmatrix} + 1$$

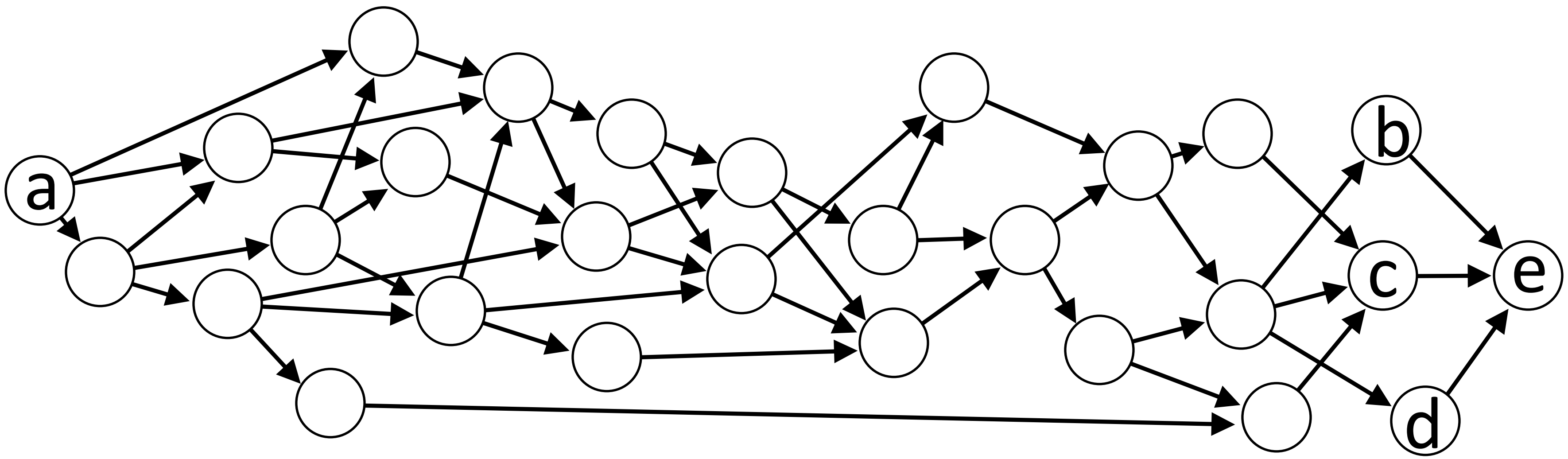
Find the Longest Path in a DAG



When are we ready to calculate the longest path to e?

$$\text{longest path to e} = \max\begin{pmatrix}\text{longest path to b}\\ \text{longest path to c}\\ \text{longest path to d}\end{pmatrix} + 1$$

Find the Longest Path in a DAG



When are we ready to calculate the longest path to e?

When we have the longest paths to b, c, and d.

$$\text{longest path to e} = \max\begin{pmatrix}\text{longest path to b}\\ \text{longest path to c}\\ \text{longest path to d}\end{pmatrix} + 1$$