CSE 331: Algorithms & Complexity

"MST Correctness"

Prof. Charlie Anne Carlson (She/Her)

Lecture 21

Wednesday October 22nd, 2025

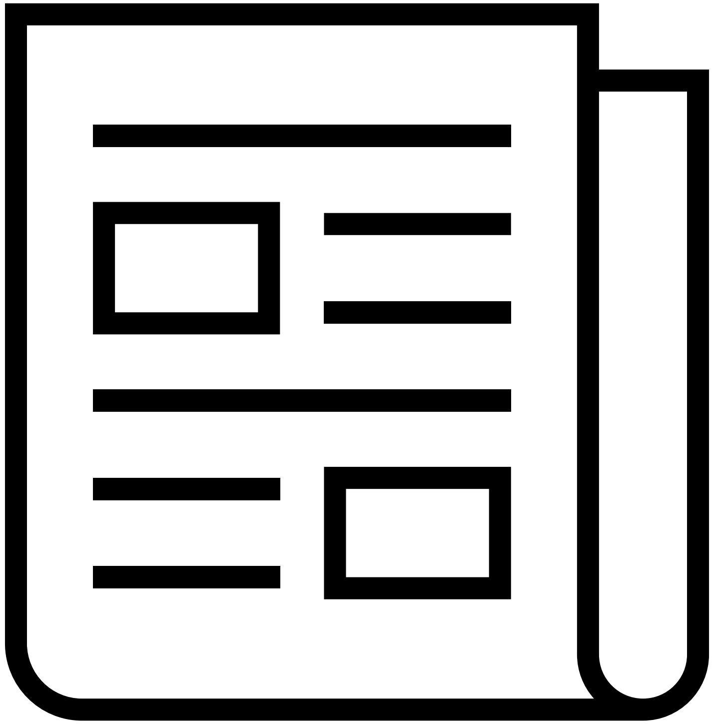
University at Buffalo

Schedule

1. Course Updates
2. Cut Property
3. Kruskal’s Algorithm
4. Prim’s Algorithm
5. Divide & Conquer

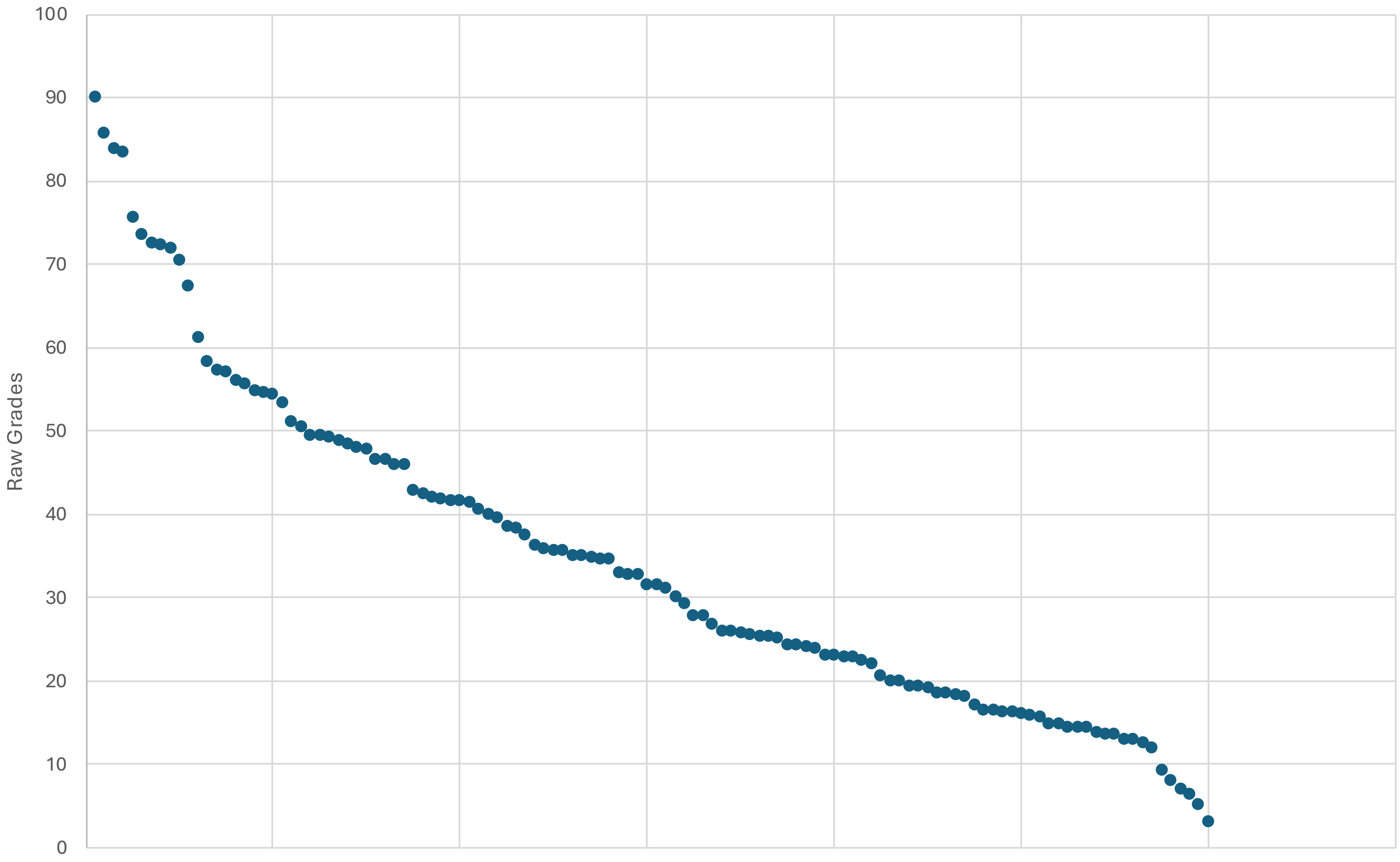
Course Updates

- Midterm Out
- Post Midterm Grades
- HW 5 Out
- Group Project
  - First Problems Oct 31st



Midterm Grades

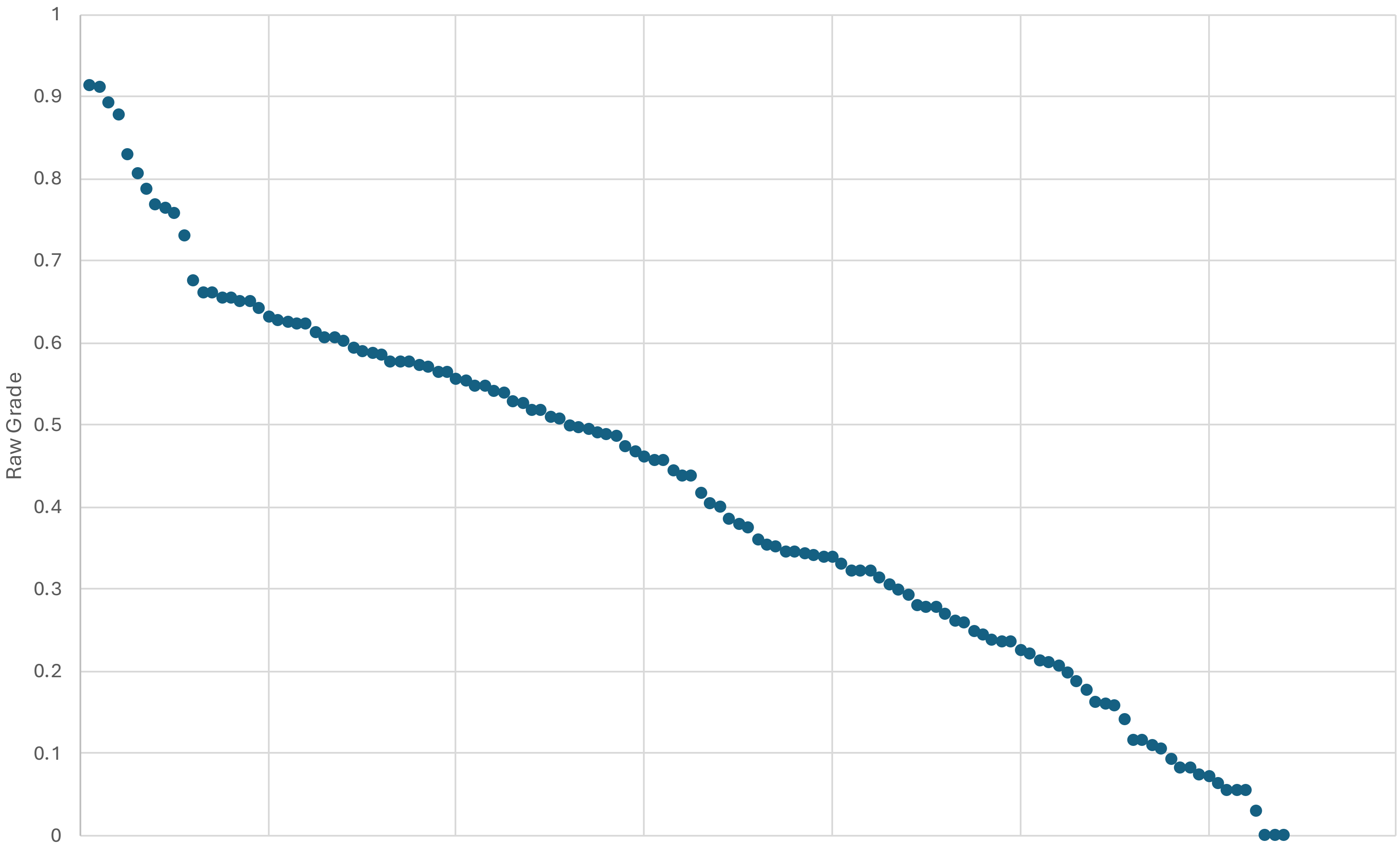
Grade = (P1+P2)



Midterm Evals Grades (One HW Drop)

Grade Includes:

- Midterm (45%)
- Top HW (49%)
  - Up To HW 3
  - Per Part
- QUIZ (6%)
  - Quiz 1



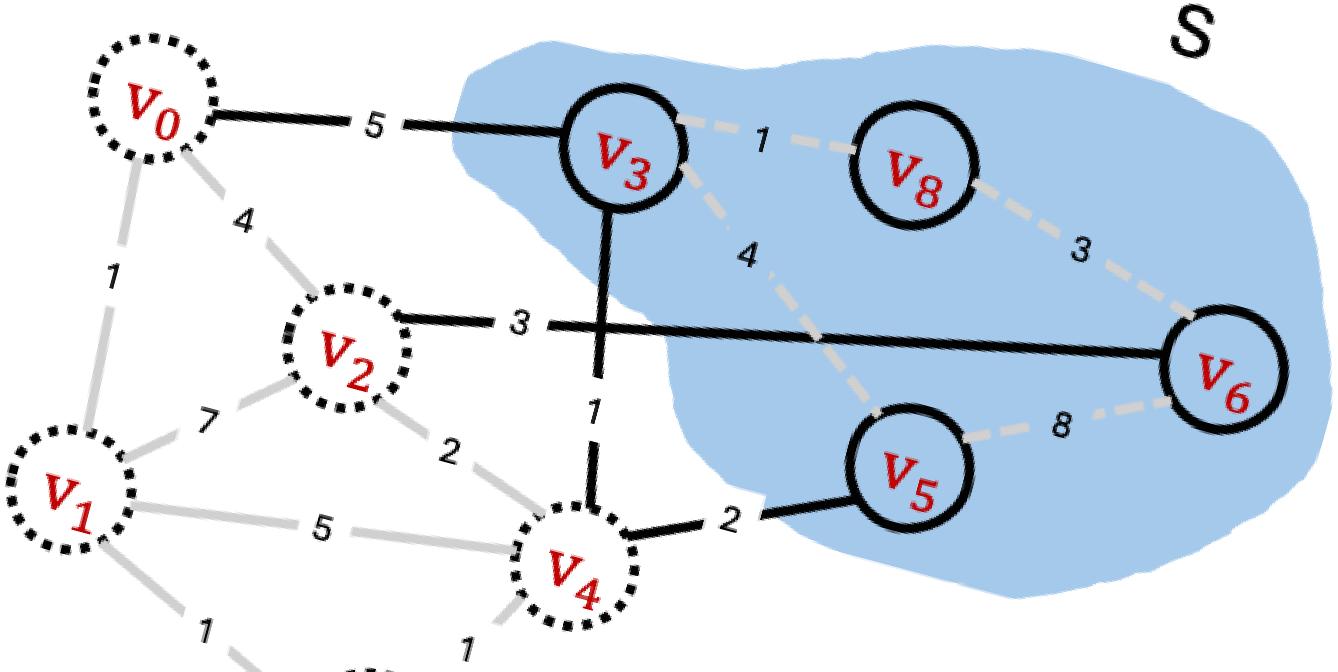
1-on-1 Meeting With Me

- I am happy to meet to discuss your course performance and help make a plan on how to move forward.
- When I make a plan for pushing out midterm eval grades, I will also make a piazza post with specific instructions on how to set up a meeting with me.
- You are free to email me now but please include a long list of times you are available so we can schedule a zoom chat.

Feedback Survey

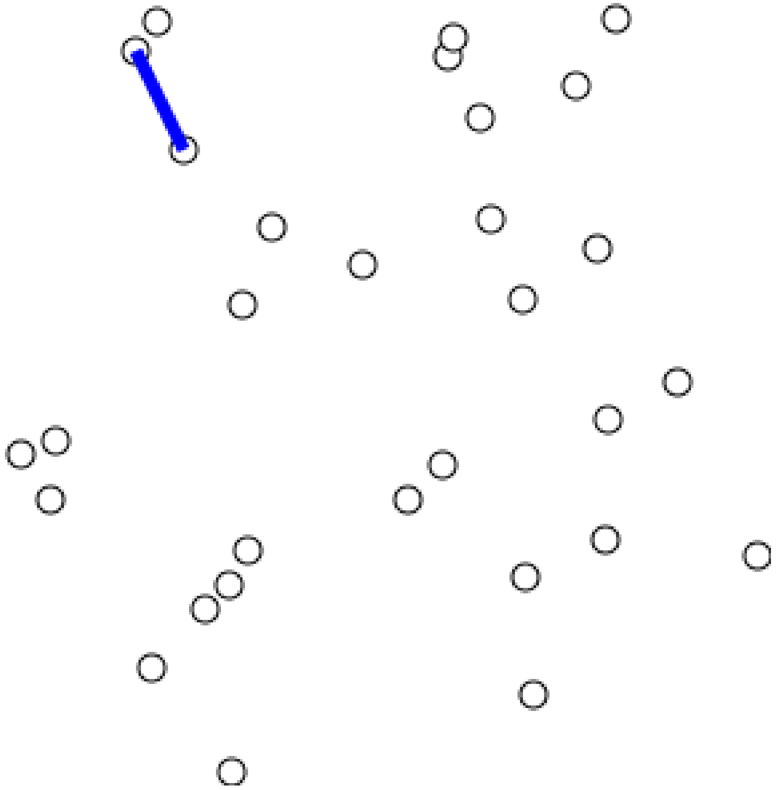
Cut Property

Lemma: Fix a graph $G = (V, E)$ with edge weights ℓ. Assume that all edges are distinct. Let S be any subset of nodes that is neither empty or equal to all of V, and let $e = (u, v)$ be the minimum-cost edge with on end in S and the other in $V \setminus S$. Then every minimum spanning tree contains the edge e.



Prim's Algorithm

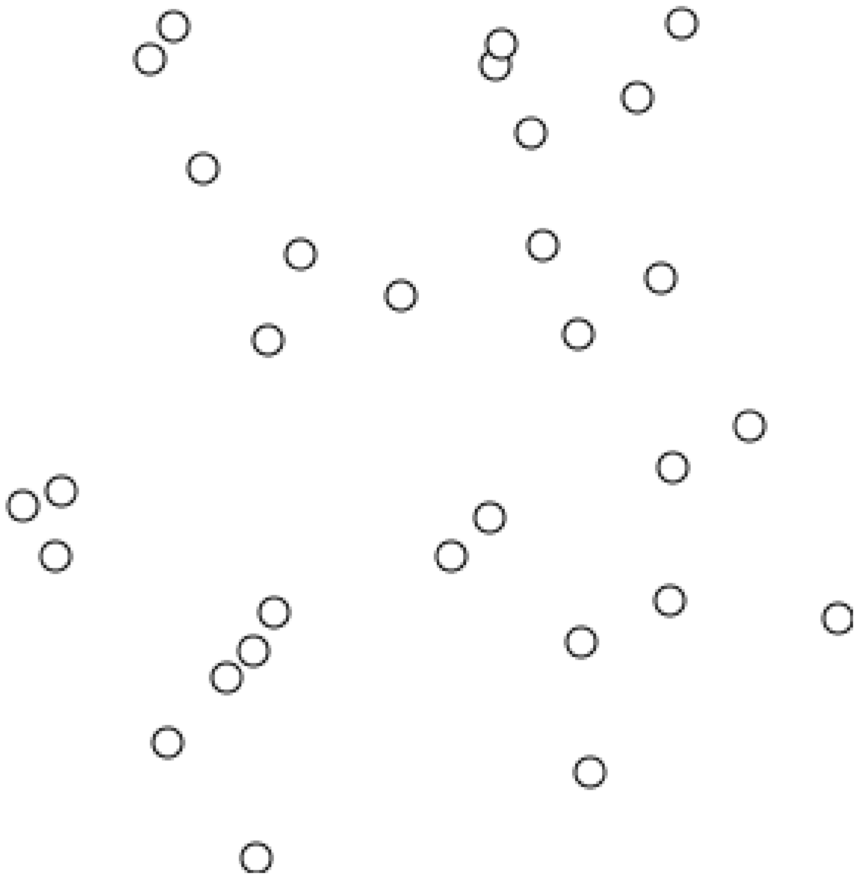
- **Input:** Undirected graph G = (V,E) and weights L
- **Output:** MST of G
  - Pick s in V arbitrarily
  - Let S = {s}
  - While S != V:
    - Find minimum weight edge e = (u,v) where u is in S but v is not.
    - Add v to S



https://en.wikipedia.org/wiki/File:PrimAlgDemo.gif

Kruskal's Algorithm

- **Input:** Undirected graph G = (V,E) and weights L
- **Output:** MST of G
  - Sort E using values in L
    - Break ties arbitrarily
  - Let T be an empty graph
  - For e in E:
    - If adding e to T doesn't case a cycle, add it.



https://en.wikipedia.org/wiki/File:KruskalDemo.gif

Claim: Kruskal’s Algorithm is Correct

Proof:

- Let e = (u,v) be an edge added by Kruskal’s algorithm
- Consider the T just before adding e.
  - Let S be the connected component of T that contains u.
- Then e was the minimum weight edge leaving S and by the Cut Property it must be in the MST.
- Hence, Kruskal’s algorithm only adds edges that must be in the MST.

Claim: Kruskal's Algorithm is Correct

Proof:

- Let e = (u,v) be an edge added by Kruskal's algorithm
- Consider the T just before adding e.
  - Let S be the connected component of T that contains u.
- Then e was the minimum weight edge leaving S and by the Cut Property it must be in the MST.
- Hence, Kruskal's algorithm only adds edges that must be in the MST.
- Finally, we note that if T was not connected then there would have been edge that could have been added without forming a cycle.
- It follows that T is the MST at the end of the algorithm.

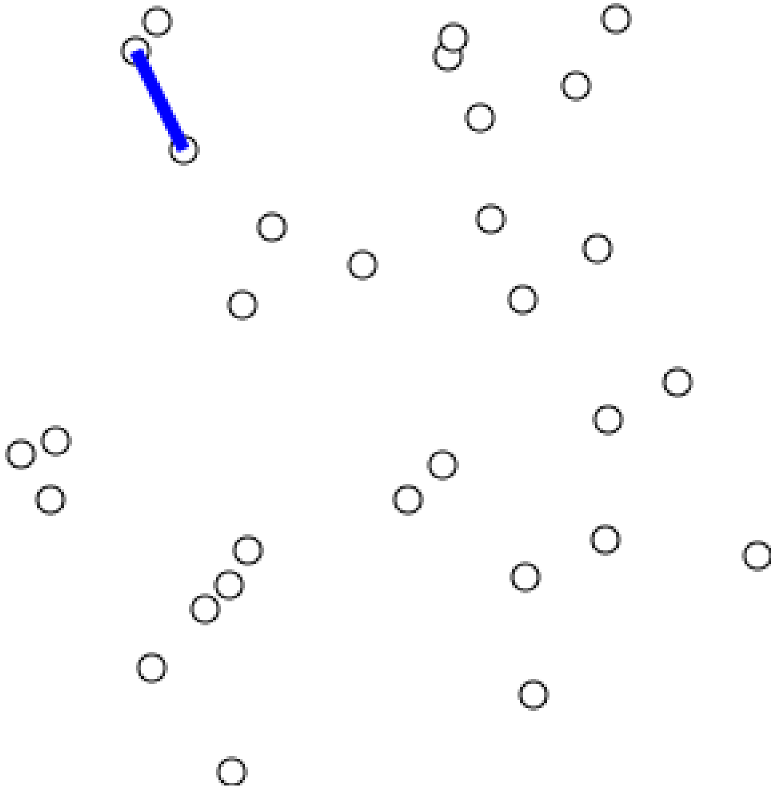
Claim: Prim's Algorithm is Correct

Proof:

- Let e = (u,v) be an edge added by Kruskal's algorithm
- Consider the T just before adding e.
  - Let S be the connected component of T that contains u.
- Then e was the minimum weight edge leaving S and by the Cut Property it must be in the MST.
- Hence, Prim's algorithm only adds edges that must be in the MST.
- Finally, we note that if T was not connected then there would have been edge that could have been added without forming a cycle.
- It follows that T is the MST at the end of the algorithm.

Prim's Algorithm Runtime

- **Input:** Undirected graph G = (V,E) and weights L
- **Output:** MST of G
  - Pick s in V arbitrarily
  - Let S = {s}
  - While S != V:
    - Find minimum weight edge e = (u,v) where u is in S but v is not.
    - Add v to S

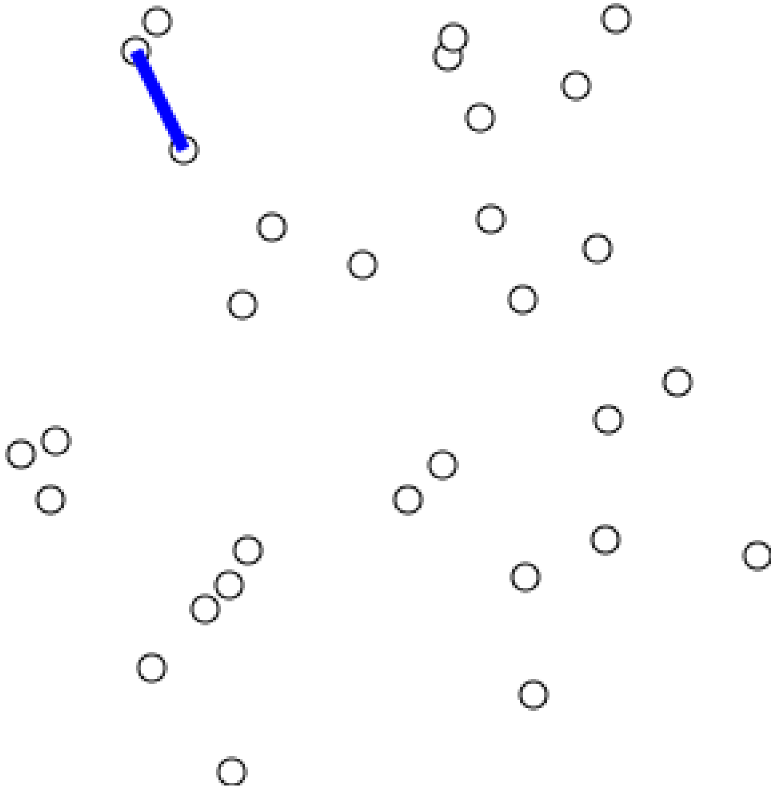


https://en.wikipedia.org/wiki/File:PrimAlgDemo.gif

Prim's Algorithm Runtime O(m log(n))

Claim: Using a priority queue, Prim's Algorithm can be implemented on a graph with n nodes and m edges to run in O(m) time, plus the time for n `PopMin` and m `DecreasePriority` operations.

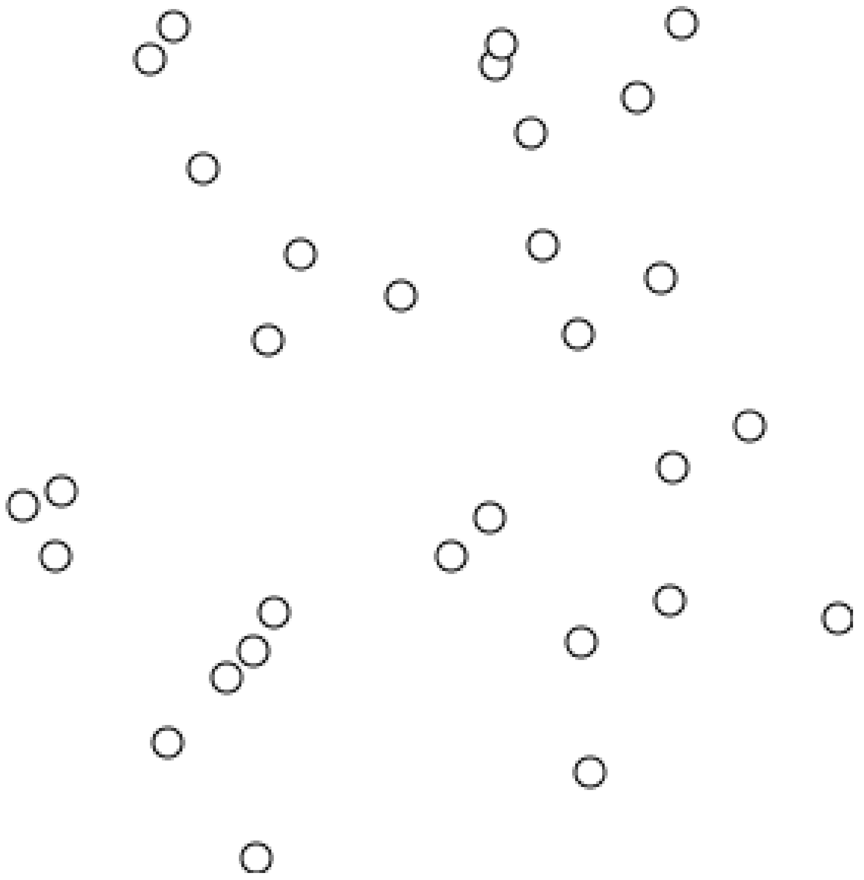
Corollary: Using a heap-based priority queue we get a running time of O(m log(n)).



https://en.wikipedia.org/wiki/File:PrimAlgDemo.gif

Kruskal’s Algorithm Runtime

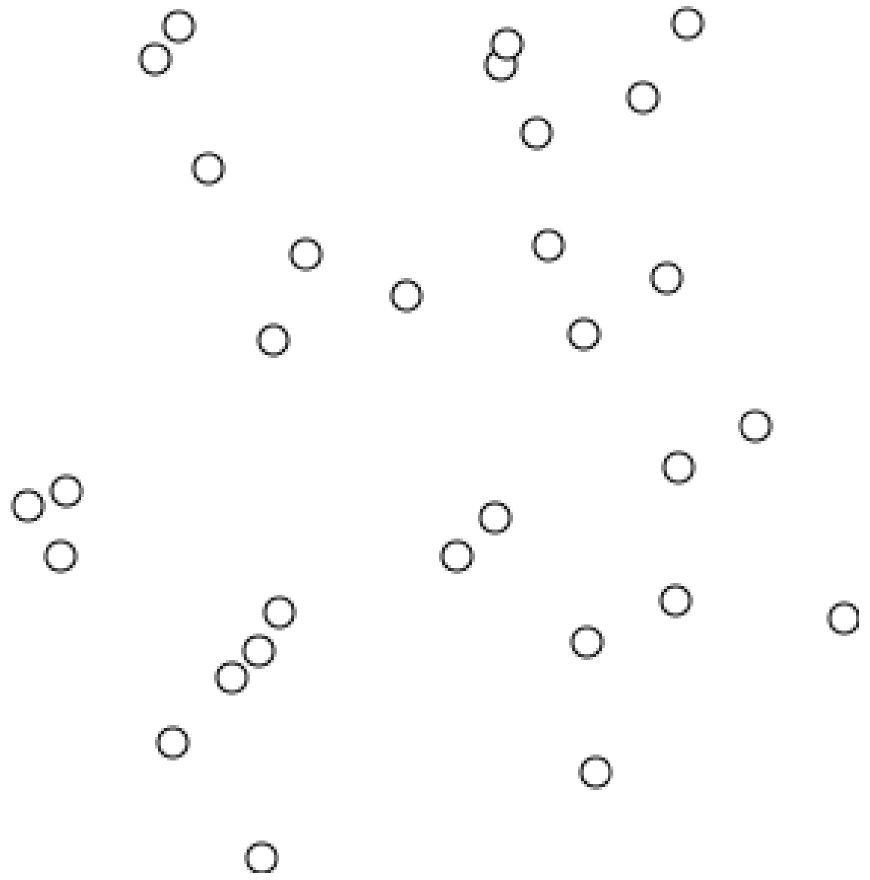
- **Input:** Undirected graph G = (V,E) and weights L
- **Output:** MST of G
  - Sort E using values in L
    - Break ties arbitrarily
  - Let T be an empty graph
  - For e in E:
    - If adding e to T doesn’t case a cycle, add it.



https://en.wikipedia.org/wiki/File:KruskalDemo.gif

Kruskal’s Algorithm Runtime O(m log(n))

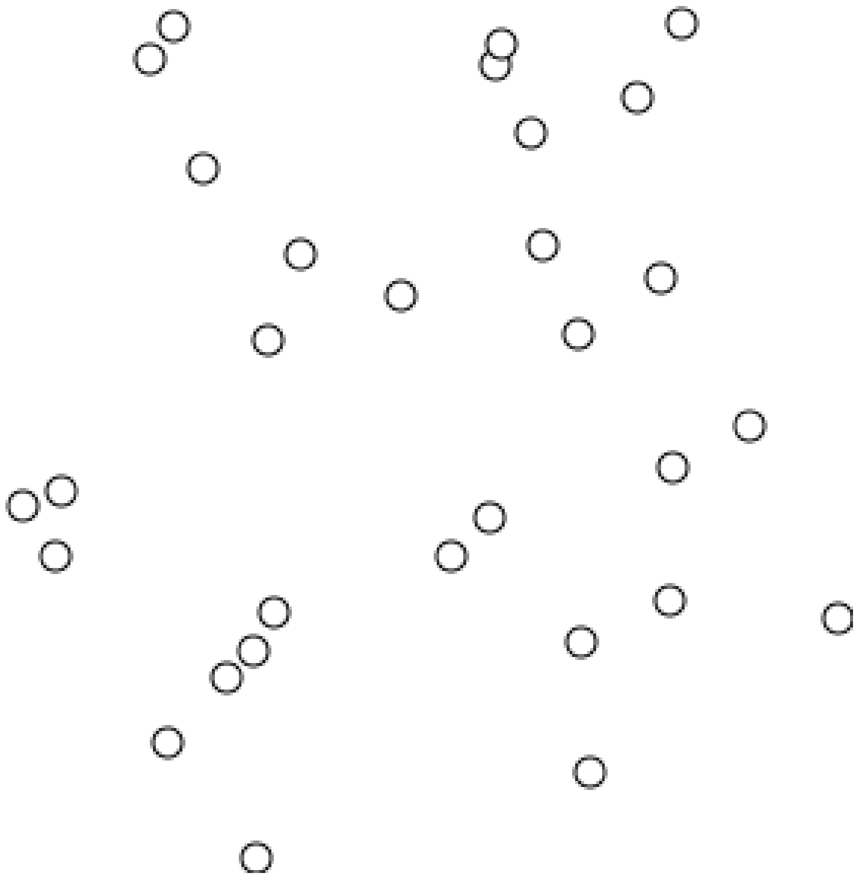
- **Input:** Undirected graph G = (V,E) and weights L
- **Output:** MST of G
  - Sort E using values in L
    - Break ties arbitrarily
  - Let T be an empty graph
  - For e in E:
    - If adding e to T doesn’t case a cycle, add it.



https://en.wikipedia.org/wiki/File:KruskalDemo.gif

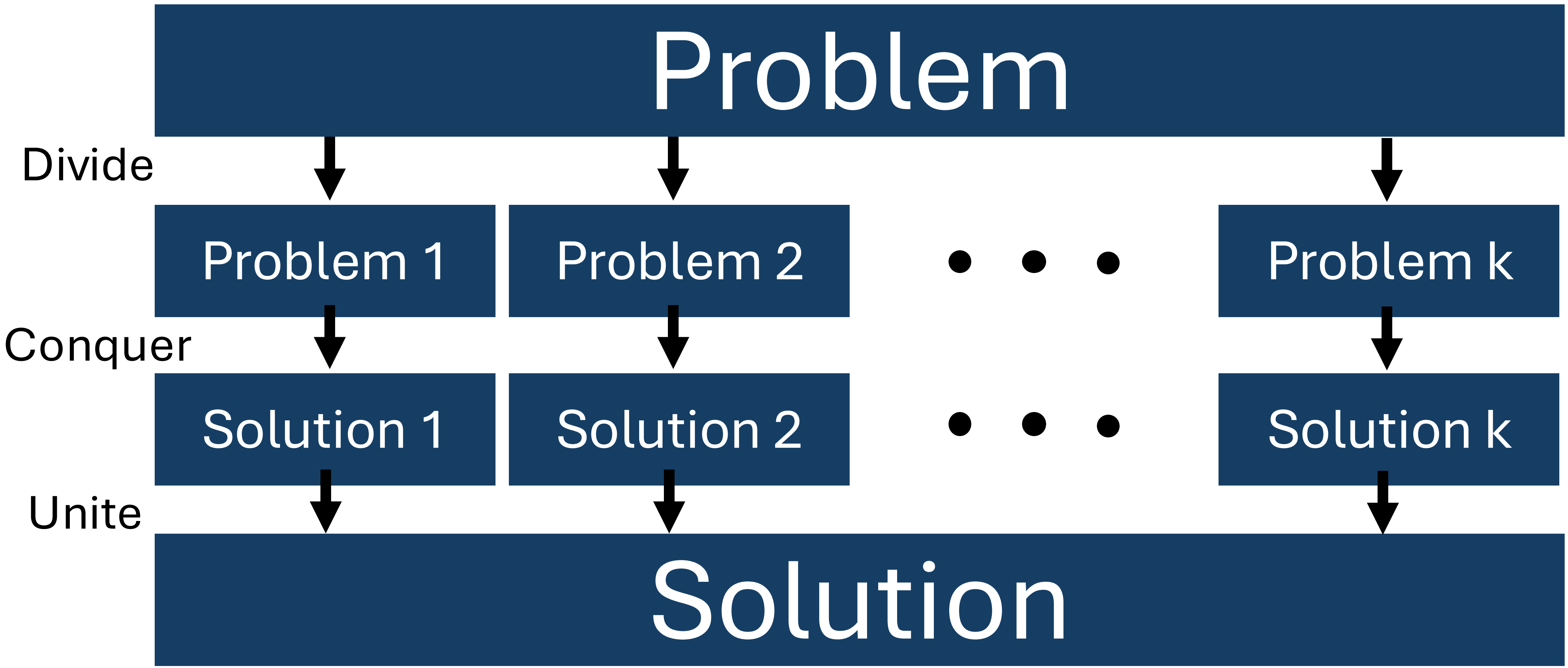
Kruskal's Algorithm Runtime O(m log(n))

- To prove this running time, we need a Union-Find data structure.
  - Keeps track of which elements in a ground set belong to the same subsets.
  - `Find(u)`: Returns name of set that contains u.
  - `Union(A,B)` combine sets A and B into one set.
  - Read KT 4.6!

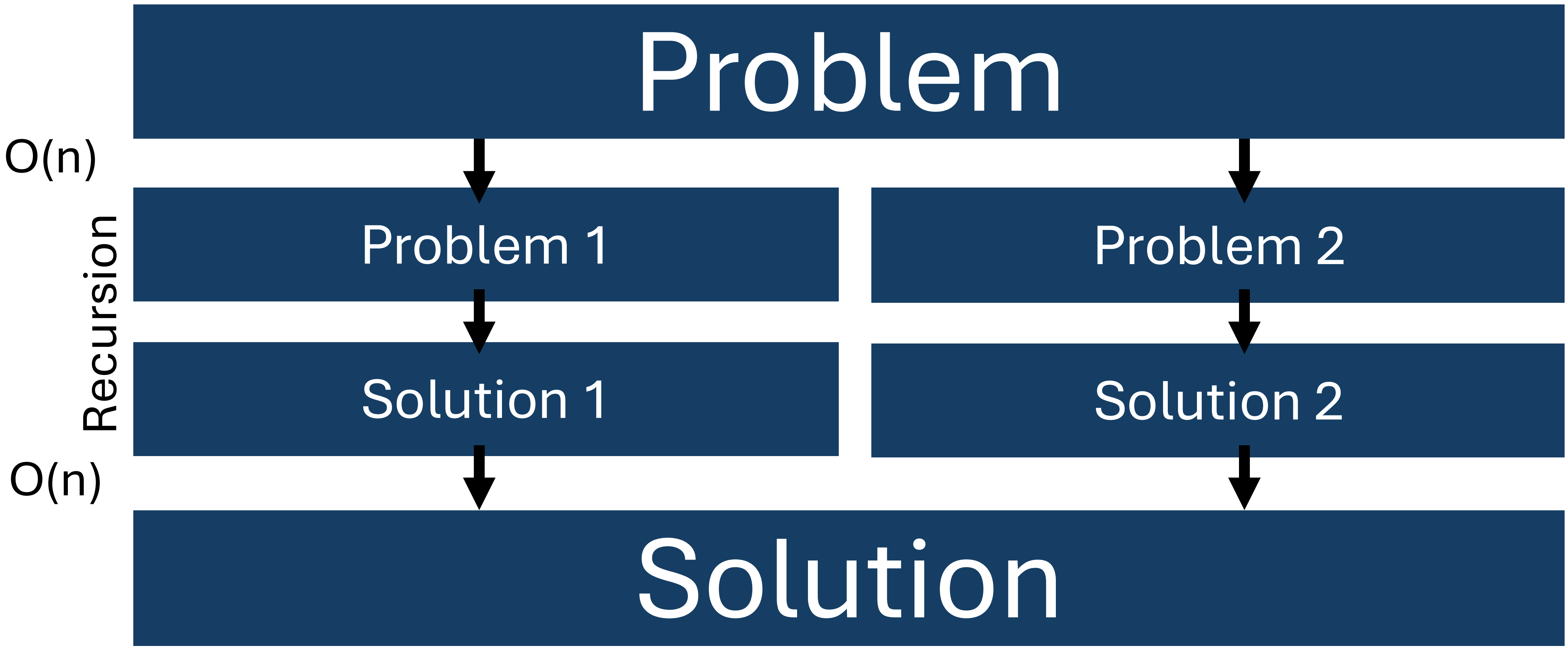


https://en.wikipedia.org/wiki/File:KruskalDemo.gif

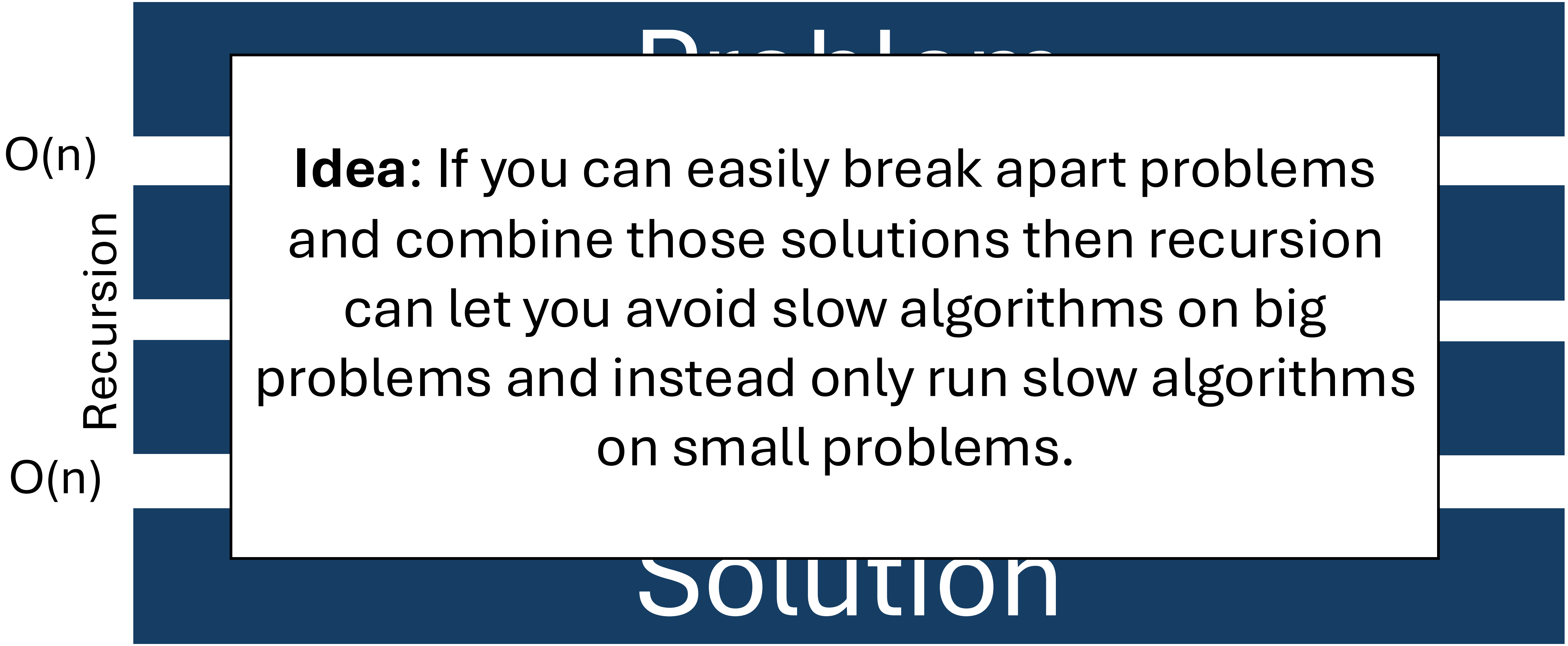
Divide & Conquer (KT 5.1 and KT 5.2)



Why Divide & Conquer?

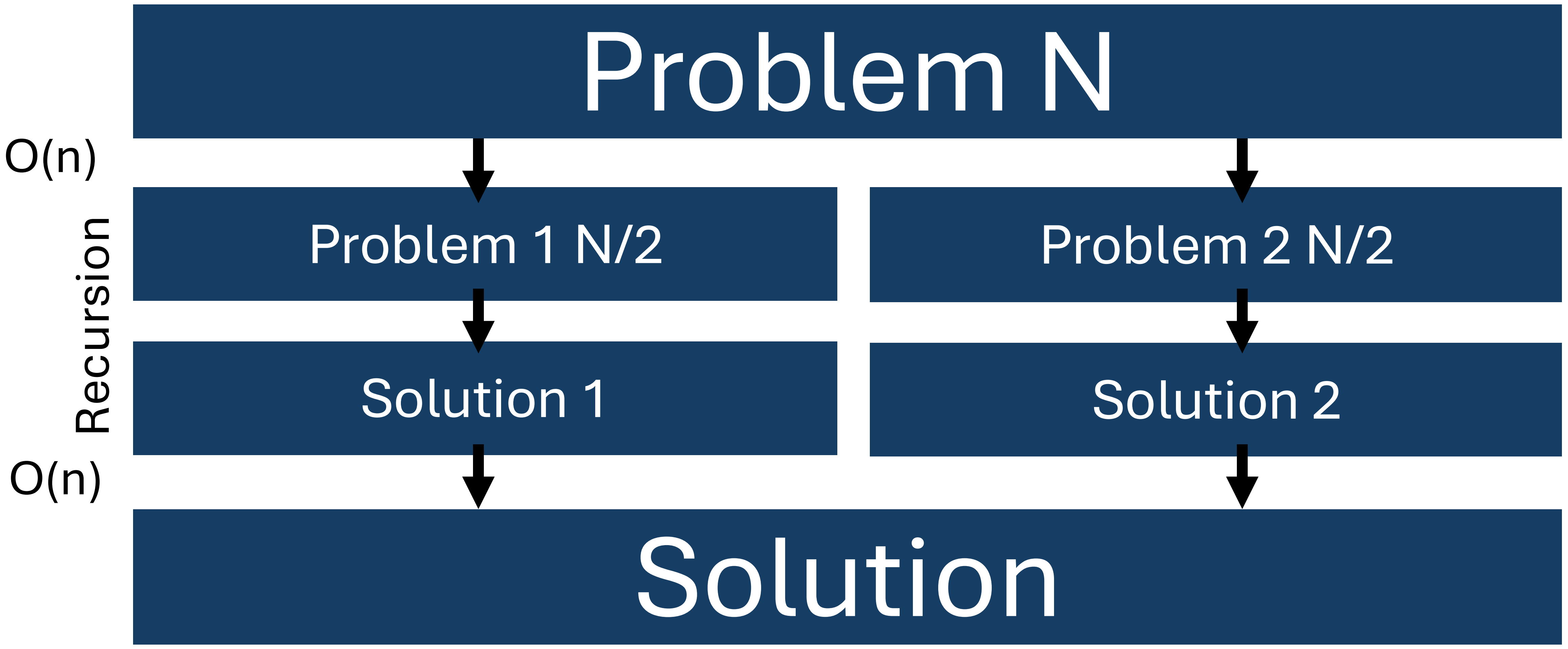


Why Divide & Conquer?

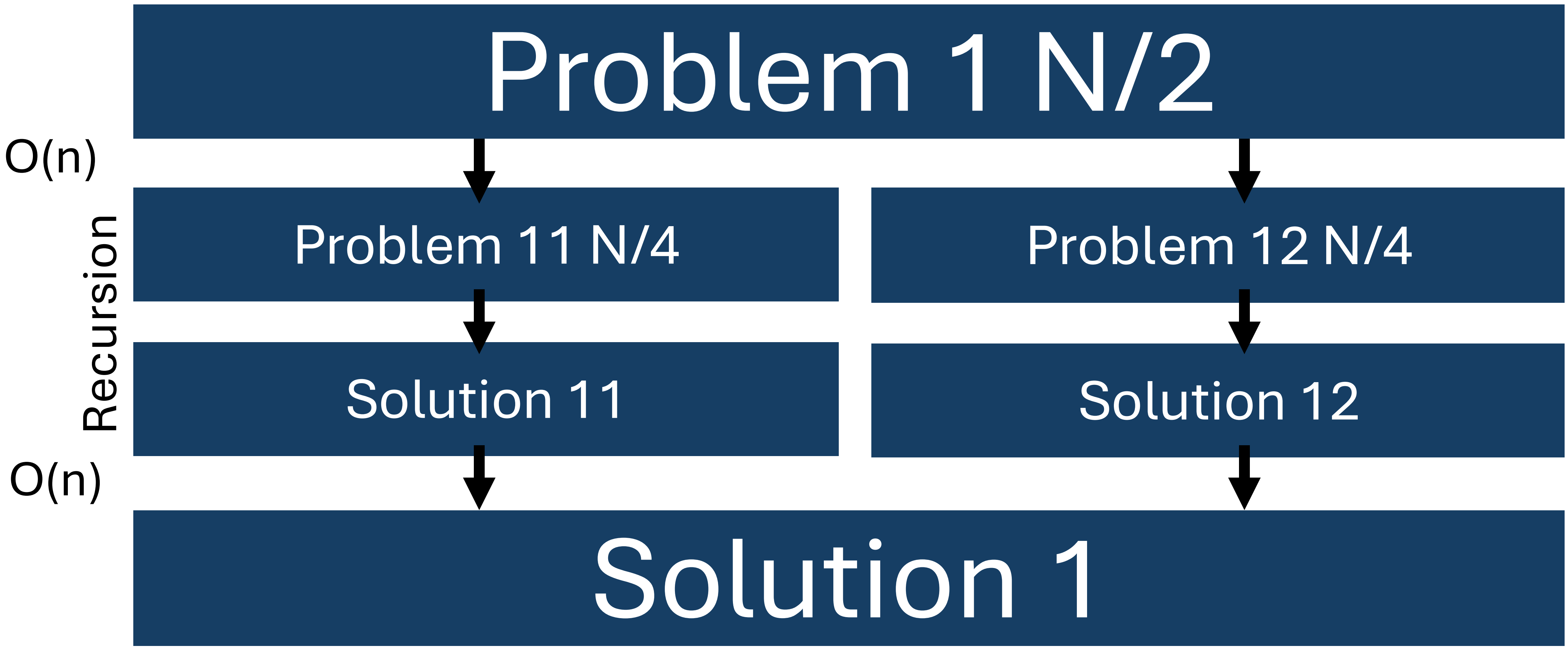


Idea: If you can easily break apart problems and combine those solutions then recursion can let you avoid slow algorithms on big problems and instead only run slow algorithms on small problems.

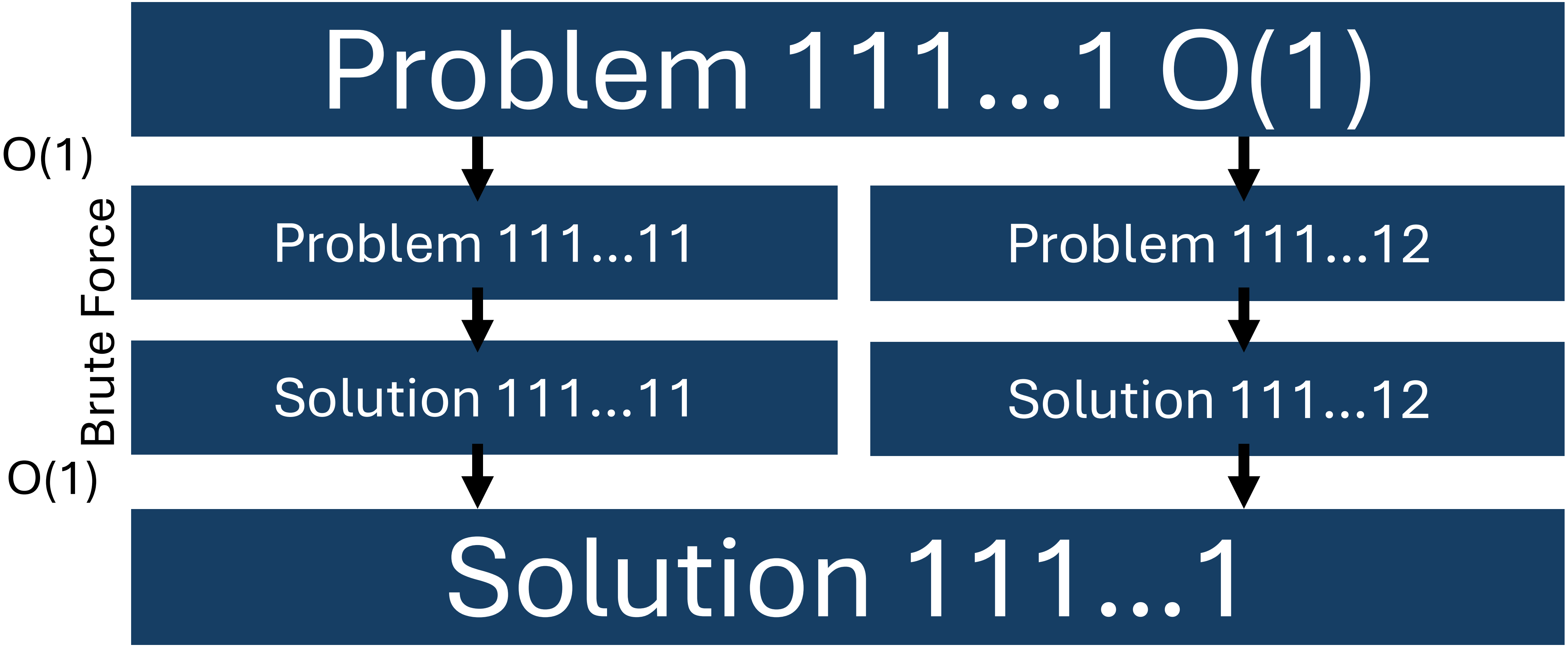
Why Divide & Conquer?



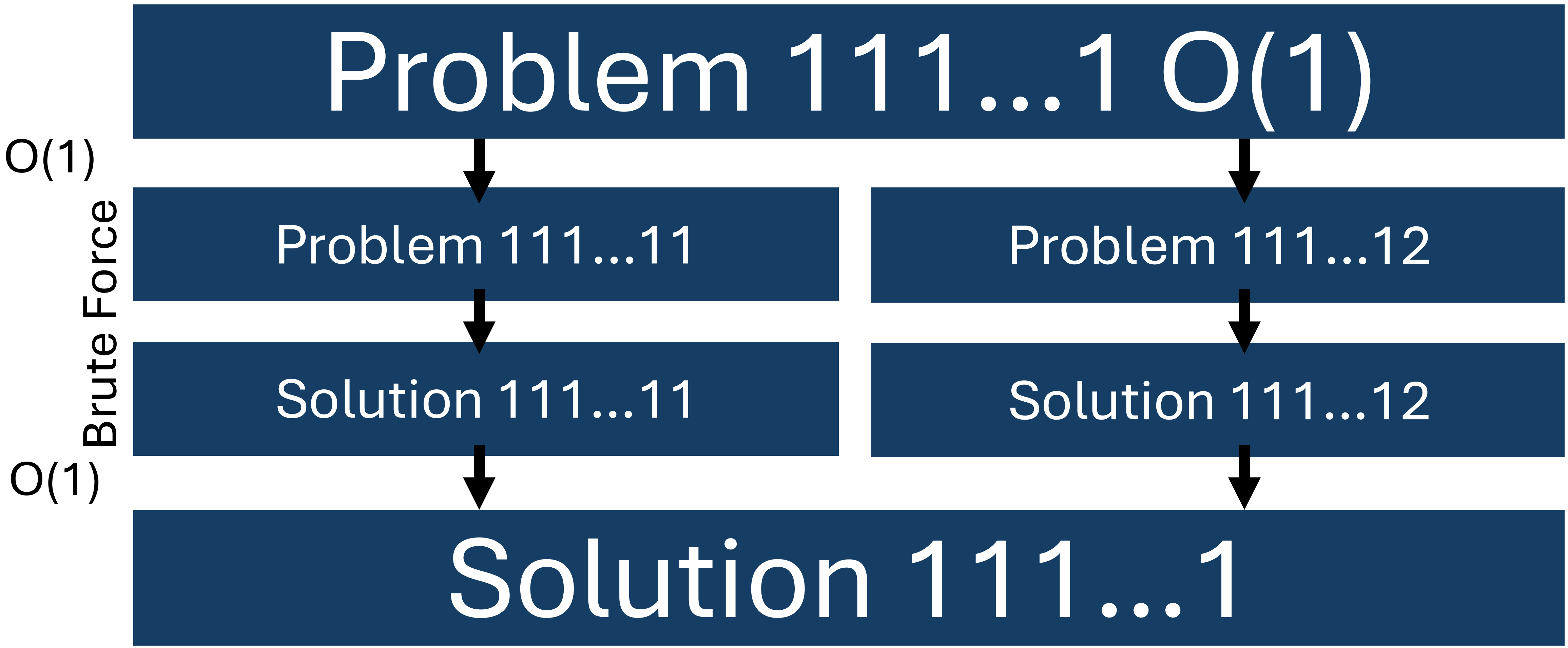
Why Divide & Conquer?



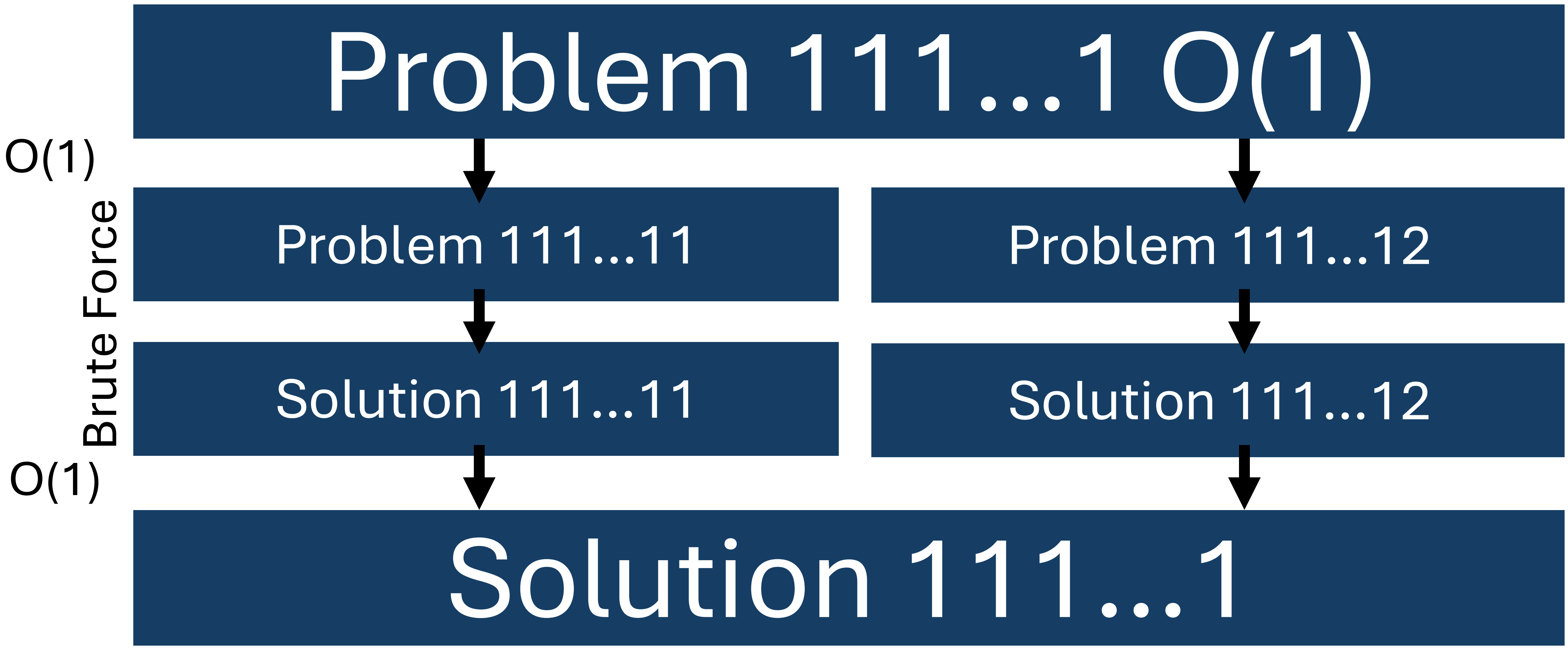
Why Divide & Conquer?



Q: How many times can you split in half?



A: After log times, you will get a constant!



Sorting

- **Problem**: Given a list of n numbers L, rearrange them in ascending order.

- E.g.
  - **Input**: [3,2,5,5,1,6,7,8]
  - **Output**: [1,2,3,5,5,6,7,8]

Sorting

- **Problem**: Given a list of n numbers L, rearrange them in ascending order.

- Sorting Algorithms:
  - Bubble Sort
  - Insertion Sort
  - Mergesort
  - Radix Sort
  - Quicksort
  - Introsort

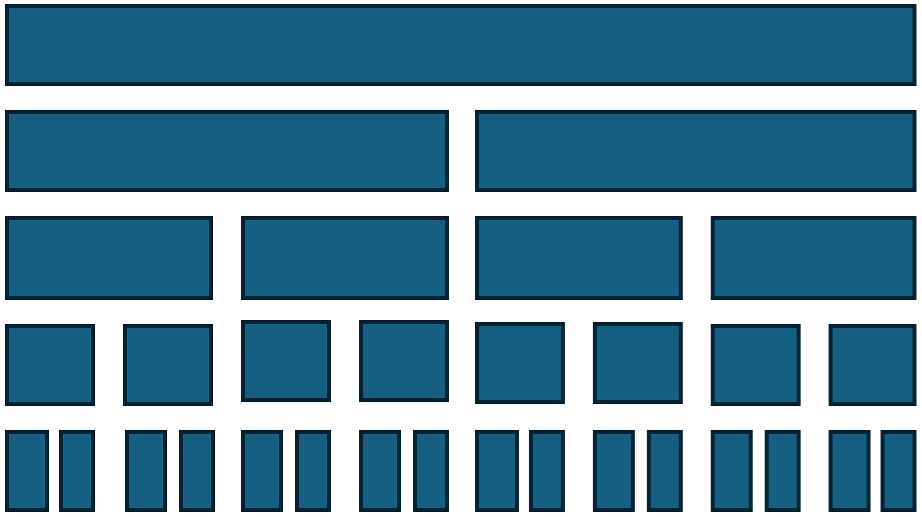
Sorting

- **Problem**: Given a list of n numbers L, rearrange them in ascending order.

- Sorting Algorithms:
  - Bubble Sort
  - Insertion Sort
  - **Mergesort**
  - Radix Sort
  - Quicksort
  - Introsort

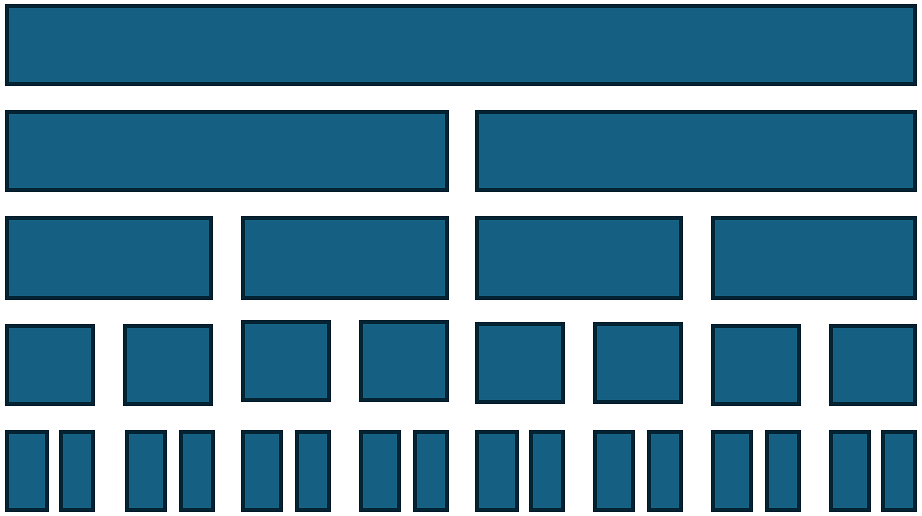
Mergesort

- **Divides**: Divides input into two pieces of equal size in linear time.
    - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece.
- **Unite**: Merges the two sorted lists in linear time.



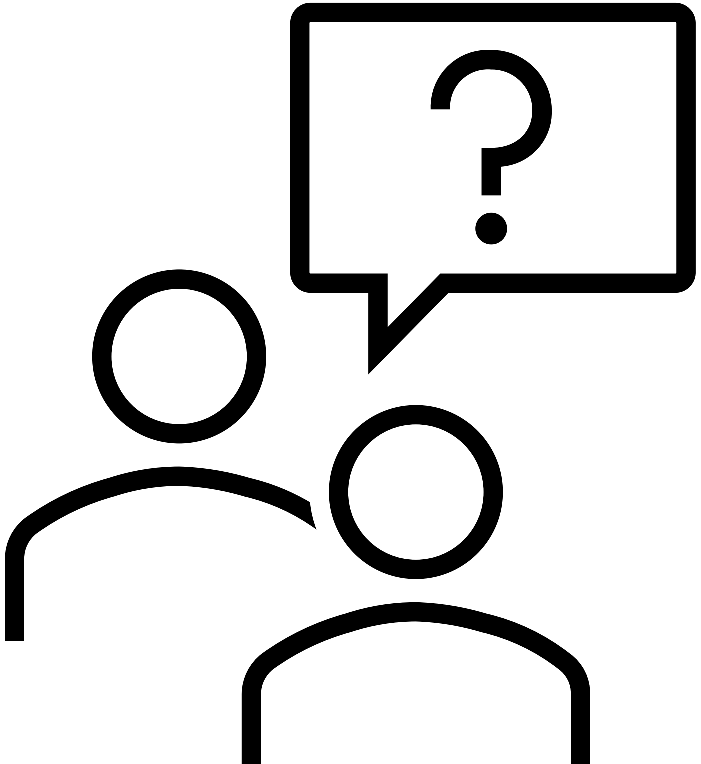
Mergesort

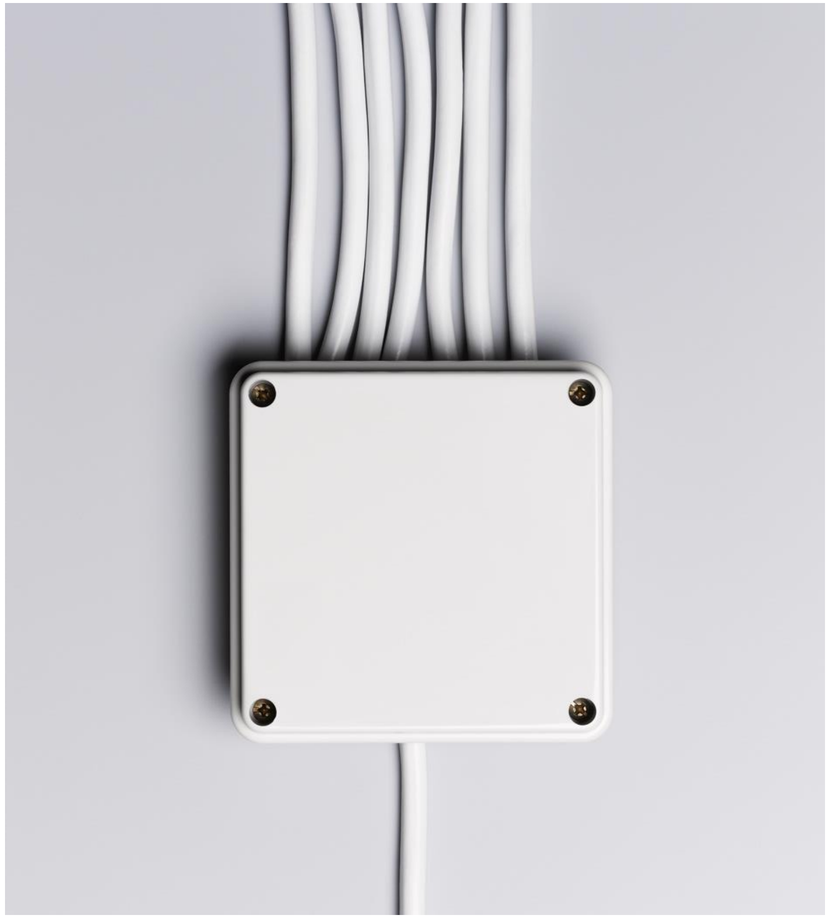
- **Base Case:** If array has length less than 2, brute force.
- **Divides**: Divides input into two pieces of equal size in linear time.
  - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece.
- **Unite**: Merges the two sorted lists in linear time.



Sorting

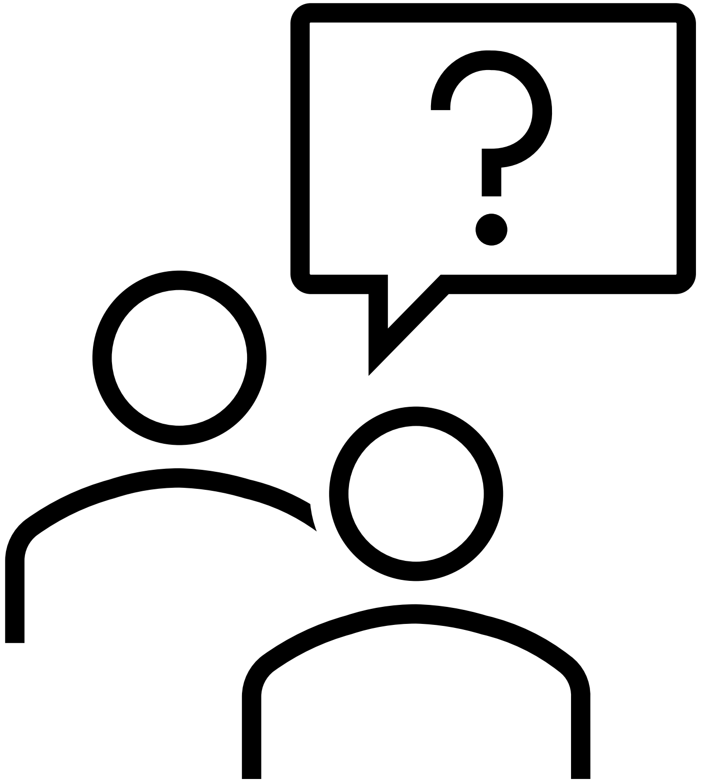
- **Problem**: Given two sorted lists A and B, find a sorted list of their union.





Merging

- **Input**: Two sorted lists A and B of length n/2
- **Output**: Sorted list of A and B
- Initialize list C to be empty
- Let i = 0 and j = 0
- While (i < n/2 or j < n/2):
  - If j == n/2 or A[i] <= B[j]:
    - C.append(A[i])
    - i += 1
  - Else:
    - C.append(B[j])
    - j += 1



Mergesort Runtime?

- **Base Case:** If array has length less than 2, brute force.
- **Divides**: Divides input into two pieces of equal size in linear time.
  - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece.
- **Unite**: Merges the two sorted lists in linear time.

Let T(n) be runtime of Mergesort.

- **Base Case:** If array has length less than 2, brute force. **O(1)**
- **Divides**: Divides input into two pieces of equal size in linear time. **O(n)**
  - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece. **T(n/2)**
- **Unite**: Merges the two sorted lists in linear time. **O(n)**

Let T(n) be runtime of Mergesort.

- **Base Case:** If array has length less than 2, brute force. **O(1)**
- **Divides**: Divides input into two pieces of equal size in linear time. **O(n)**
  - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece. **T(n/2)**
- **Unite**: Merges the two sorted lists in linear time. **O(n)**

$$\boldsymbol{T(n)} \leq ?$$

Let T(n) be runtime of Mergesort.

- **Base Case:** If array has length less than 2, brute force. **O(1)**
- **Divides**: Divides input into two pieces of equal size in linear time. **O(n)**
  - Assume even length for now.
- **Conquer**: Recursively calls mergesort on each piece. **T(n/2)**
- **Unite**: Merges the two sorted lists in linear time. **O(n)**

$$\boldsymbol{T(n)} \leq \begin{cases} O(1) & \text{if } n \leq 2 \\ 2T(n/2) + O(n) & \text{o.w.} \end{cases}$$

Let T(n) be runtime of Mergesort.

- **Base Case:** If array has length less than 2, brute force. **O(1)**
- **Divides**: Divides input into two pieces of equal size in linear time. **O(n)**
- **Conquer**: Recursively calls mergesort on each piece. **T(n/2)**
- **Unite**: Merges the two sorted lists in linear time. **O(n)**

$$\boldsymbol{T(n)} \leq \begin{cases} c & \text{if } n \leq 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil + c'n & \text{o.w.} \end{cases}$$

How do you solve a recurrence?

- **Unrolling:** We analyze the first few ”levels” of the recursion, find a pattern and then prove that the pattern is correct.
- **Guess and Check:** We guess what the answer and the substitute it in to check that it works. That is, we prove it works.
- We will talk about these next time but read KT 5.1 and KT 5.2!

$$\boldsymbol{T(n)} \leq \begin{cases} c & \text{if } n \leq 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil + c'n & \text{o.w.} \end{cases}$$