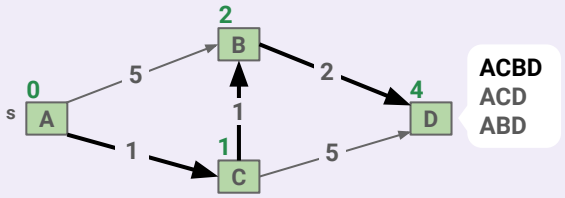
Q Arbitrary Challenging Problem of the Day

Given a weighted graph, find the **second-shortest path** from a source to a goal vertex.

Given a weighted graph, find the **kth-shortest path** from a source to a goal vertex.



Q Nearest Pseudocode

nearest(Node n, Point goal, Node best):

- If n is null: return best
- If n.distance(goal) < best.distance(goal): best = n
- If goal < n:
  - closer = n.left
  - further = n.right
- else:
  - closer = n.right
  - further = n.left
- best = nearest(closer, goal, best)
- best = nearest(further, goal, best)
- return best

Something is missing. What?

Symptoms of Complexity

Ousterhout describes three symptoms of complexity.

Change amplification. A simple change requires modification in many places. Our overly complex k-d tree was a good example of this.

Cognitive load. How much you need to know in order to make a change. Note that, often, **more** lines of code actually makes code simpler because it is more narrative.

Unknown unknowns. The worst type of complexity. This occurs when it's not even clear what you need to know in order to make modifications! Common in large code bases.

Obvious Systems

A well-designed software system is, ideally, **obvious**.

A new developer can quickly and confidently make changes in an **obvious system**.

1. Quickly understand how existing code works.
2. Come up with a proposed change without doing too much thinking.
3. Feel confident that the change should work **without knowing the rest of the system**.

Complexity as the Accumulation of Technical Debt

Every software system starts out beautiful, pure, and clean.

As they are built upon, they slowly twist into uglier and uglier shapes. This is almost inevitable in real systems. “Move fast and break things” often incurs **technical debt**.

“Complexity comes about because hundreds or thousands of small dependences and obscurities build up over time… Eventually, there are so many of these small issues that every possible change is affected by several of them.”

Ousterhout recommends a zero tolerance philosophy.

Tactical vs. Strategic Programming

Tactical programming introduces tons of little complexities and **code smells**, e.g. making two copies of a method that do something similar. These temporary patches deal with **technical debt** by taking on more **technical debt**.

“The first step towards becoming a good software designer is to realize that working code isn’t enough.”

Strategic programming. In real systems: Try to imagine how things might need to be changed in the future, and make sure your design can handle such changes.

In large, changing systems, new problems or new demands will arise. **Refactor code**.

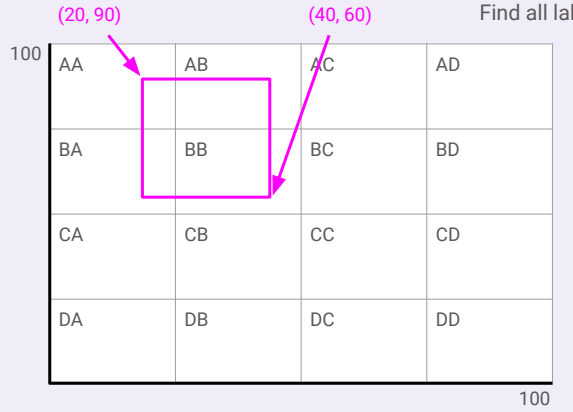
Goal: Fill in the findAnswer Method

```
public class Rasterer {
  private static final double MAX_X = 100;
  private static final double MAX_Y = 100;

  public static List<String> findAnswer(double x1, double y1,
                                        double x2, double y2) {
    // TODO
    return null;
  }
}
```

What are some potentially useful helper methods?

Q Arbitrary Challenging Problem of the Day (Redux)



Find all labels that overlap the query.

findAnswer(20, 90, 40, 60)

[“AA”, “AB”, “BA”, “BB”]