Shortest Paths
Part One
Recap from Last Time
A graph is a mathematical structure for representing relationships.

A graph consists of a set of nodes connected by edges.
Breadth-First Search
Breadth-First Search
BFS Pseudocode

breadth-first-search() {
    make a queue of nodes.
    enqueue start node.
    color the start node yellow.

    while (the queue is not empty) {
        dequeue a node from the queue.
        color that node green.

        for (each neighboring node) {
            if (that node is gray) {
                color the node yellow.
                enqueue it.
            }
        }
    }
}
The Limits of Breadth-First Search
Each intersection is a node.

Edges represent roads.

Different roads have different lengths.

**Question:** What’s the best way to get from point A to point B?
The Model

- We have a graph in which each edge has a nonnegative cost or weight associated with it.
- We want to find the lowest-cost path from point A to point B.
- BFS does not take edge weights into account.
- How might we go about solving this problem?
Option 1: Brute-Force!

- We could conceivably solve this problem using brute force and a backtracking recursion.

- **Problem:** There can be a lot of different paths in a graph!

- This is way too inefficient to use in practice.
Option 2: Expand the Graph

- BFS works in the case where each edge has equal weight.
- **Idea:** What if we split each edge of length $k$ into $k$ smaller edges?
Option 2: Expand the Graph

- BFS works in the case where each edge has equal weight.
- **Idea:** What if we split each edge of length $k$ into $k$ smaller edges?
- What if there are fractional edges? Or large weights?
Option 2: Expand the Graph

- BFS works in the case where each edge has equal weight.

- **Idea:** What if we split each edge of length $k$ into $k$ smaller edges?

- What if there are fractional edges? Or large weights?
Option 3: Look at the problem more closely
The Pattern

All yellow nodes (nodes we've seen, but don't know the distance to.)
The Pattern

Look at the lowest-cost yellow node.
The Pattern

No other path to this node can be better than the one we already know about!
At a Glance

- The approach suggested here gives rise to *Dijkstra’s algorithm*, a fast, powerful, and famous algorithm for computing shortest paths.

- **Key idea:** As in BFS, split nodes into
  - *gray nodes* we haven’t seen,
  - *yellow nodes* that are on the frontier, and
  - *green nodes* we have the best path to,
  then repeatedly turn the lowest-cost yellow node into a green node.
Implementing Dijkstra’s Algorithm
The Finished Product

dijkstra's-algorithm() {
    make a priority queue of nodes.
enqueue start node at distance 0.
color the start node yellow.

    while (the queue is not empty) {
        dequeue a node from the queue.
        if (that node isn't green) {
            color that node green.

            for (each neighboring node) {
                if (that node is not green) {
                    color the node yellow.
enqueue it at the new distance.
                }
            }
        }
    }
}
dijkstra's-algorithm() {
    make a priority queue of nodes.
    enqueue start node at distance 0.
    color the start node yellow.

    while (the queue is not empty) {
        dequeue a node from the queue.
        if (that node isn't green) {
            color that node green.

            for (each neighboring node) {
                if (that node is not green) {
                    color the node yellow.
                    enqueue it at the new distance.
                }
            }
        }
    }
}
The Finished Product

dijkstra's-algorithm() {
    make a priority queue of nodes.
    enqueue start node at distance 0.
    color the start node yellow.

    while (the queue is not empty) {
        dequeue a node from the queue.
        if (that node isn't green) {
            color that node green.
        }

        for (each neighboring node) {
            if (that node is not green) {
                color the node yellow.
                enqueue it at the new distance.
            }
        }
    }
}

As a consequence, when dequeuing nodes, make sure we're not visiting something we've already processed.