administrivia...
- assignment 9 is due on Wednesday
  - it is another pair-programming assignment

- my office hours are back to Tuesdays 2-4pm

- midterm 2 next Monday
refresh...
-graphs have no root; must store all nodes

```java
class Graph<E> {
    List<Node> nodes;
    ...
}
```

-implementation is more general than a tree

```java
class Node{
    E Data;
    List<Node> neighbors;
    ...
}
```

-the order in which neighbors appear in the list is unspecified

- a different order still make the same graph!

-called an adjacency list
depth-first search
we want to find a path from A to C

SO... start from A, traverse its first edge, save where we came from, and recurse

A.visited = true
B.cameFrom = A
traverse the first unvisited node in the edge list recursively, save where we came from

B.visited = true
E.cameFrom = B
traverse the first unvisited node in the edge list recursively, save where we came from

E.visited = true

look at the first edge; node A has already been visited, so skip
Look at next edge; **C** has not been visited yet

C.cameFrom = E
node **C** is our goal. we are done!

C.visited = true

follow each node's **cameFrom** to reconstruct the path

C.cameFrom = E, E.cameFrom = B, B.cameFrom = A

path: **A** — **B** — **E** — **C**
is there a better (shorter) path from A to C?

what determines which path DFS finds?

DFS is not guaranteed to find the shortest path, just a path.
DFS(Node curr, Node goal)
{
    curr.visited = true

    if(curr.equals(goal))
        return

    for(Node next : curr.neighbors)
        if(!next.visited)
        {
            next.cameFrom = curr
            DFS(next, goal)
        }
}

// path is now saved in nodes' .cameFrom
let’s talk about DFS complexity…
DFS(Node curr, Node goal)
{
    curr.visited = true

    if(curr.equals(goal))
        return

    for(Node next : curr.neighbors)
        if(!next.visited)
            next.cameFrom = curr
            DFS(next, goal)
}

// path is now saved in nodes’ .cameFrom
DFS(Node curr, Node goal) {
    curr.visited = true

    if(curr.equals(goal))
        return

    for(Node next : curr.neighbors)
        if(!next.visited)
            {
                next.cameFrom = curr
                DFS(next, goal)
            }
}

// path is now saved in nodes’ .cameFrom
DFS(Node curr, Node goal) {
    curr.visited = true

    if(curr.equals(goal))
        return

    for(Node next : curr.neighbors)
        if(!next.visited)
            next.cameFrom = curr
            DFS(next, goal)

} // path is now saved in nodes’ .cameFrom

complexity?
-worst-case complexity of DFS
  -we visit each node once (and only once)
  -we cross each edge once (and only once)
  -$O(V + E)$

-what if you have to find the start node?

-does $E$ relate to $V$?
sparse vs dense

- most graphs have *at most* one edge between any pair of vertices

-*dense graphs* have edges between almost every pair of vertices
  - this is the rare case
  - \( E \approx V^2 \)

- more commonly, graphs are *sparse*
  - \( E \approx V \)

- most graph algorithms are efficient for sparse graphs
breadth-first search
we want to find a path from A to C

mark and enqueue the start node A

A.visited = true

queue: A

visited

unvisited
Deque the first node in the queue (A)

Mark and enqueue A’s unvisited neighbors

B.cameFrom = A
D.cameFrom = A
B.visited = true
D.visited = true

queue: B D

visited 
unvisited
Dequeue the first node in the queue (**B**)

mark and enqueue **B**’s unvisited neighbors

E.cameFrom = B
E.visited = true
dequeue the first node in the queue (D)

mark and enqueue D’s unvisited neighbors

C.cameFrom = D
C.visited = true
Dequeue the first node in the queue (E)

mark and enqueue E’s unvisited neighbors

(no unvisited neighbors!)
dequeue the first node in the queue (C)

C is the goal! reconstruct the path with *cameFrom* references

C.cameFrom = D,
D.cameFrom = A

path: A — D — C
is this the shortest path?
path: \textbf{A} \rightarrow \textbf{D} \rightarrow \textbf{C}

BFS visits nodes closest to the start-point first

therefore, the first path found is the shortest path (closest to the start node)
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
        {
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
        }
    }
}
let’s talk about BFS complexity…
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
BFS(Node start, Node goal)
{
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty())
    {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
        {
            if(!next.visited)
            {
                next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
        }
    }
}
BFS(Node start, Node goal) {
    start.visited = true
    Q.enqueue(start)

    while(!Q.empty()) {
        Node curr = Q.dequeue()
        if(curr.equals(goal))
            return

        for(Node next : curr.neighbors)
            if(!next.visited)
                {next.visited = true
                next.cameFrom = curr
                Q.enqueue(next)
            }
    }
}
-worst-case complexity of BFS
  - we visit each node once (and only once)
  - we cross each edge once (and only once)
- $O(V + E)$

-same as for depth-first search
dijkstra’s algorithm
we want to find a path from **A** to **C**

this time we use a priority queue. mark nodes **after** removal from the queue.

priority queue: [ ] [ ] [ ] [ ] [ ]

visited ○

unvisited ○
we want to find a path from **A** to **C**

A.costSoFar = 0
we want to find a path from A to C

dqueue A(0), and enqueue A's neighbors with A's cost-so-far plus the edge weight

B.costSoFar = A.costSoFar + 3
D.costSoFar = A.costSoFar + 9
B.cameFrom = A
D.cameFrom = A

priority queue: B(3) D(9)
we want to find a path from **A** to **C**

dequeue **B** (3), and enqueue **B**'s neighbors with **B**'s cost-so-far plus the edge weight

E.costSoFar = B.costSoFar + 4
E.cameFrom = B

priority queue: [E(7) D(9)]
we want to find a path from A to C

Dequeue E(7), and enqueue E’s neighbors with E’s cost-so-far plus the edge weight

// A visited, so skip
// cheaper path to D found!
C.costSoFar = E.costSoFar + 4
D.costSoFar = E.costSoFar + 1
D.cameFrom = E

priority queue: D(8) C(11)
we want to find a path from A to C

dequeue D (8), and enqueue D's neighbors with D's cost-so-far plus the edge weight

// cheaper path to C found!
C.costSoFar = D.costSoFar + 2
C.cameFrom = D

priority queue: C(10)

visited ○
unvisited ○
we want to find a path from A to C
dequeue C(10). We found our goal! Final cost is 10. Reconstruct path.

A — B — E — D — C
Dijkstra(Node start, Node goal)
{
    initialize all nodes’ cost to infinity

    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal) {return} \ \ done!
        curr.visited = true

        for each unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight(curr, n))
            {
                PQ.enqueue(n) || update n’s position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight(curr, n)
            }
        }
    }
}
let’s talk about Dijkstra’s complexity…
Dijkstra(Node start, Node goal)
{
    \textit{initialize all nodes' cost to infinity}

    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal) {return} \ \texttt{done!}
        curr.visited = true

        for each unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight(curr, n))
            {
                PQ.enqueue(n) || update n's position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight(curr, n)
            }
        }
    }
}
Dijkstra(Node start, Node goal)
{
    initialize all nodes’ cost to infinity

    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal) {return} \ \done!
        curr.visited = true

        for each unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight(curr, n))
            {
                PQ.enqueue(n) || update n’s position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight(curr, n)
            }
        }
    }
}
Dijkstra(Node start, Node goal)
{
    initialize all nodes’ cost to infinity

    PQ.enqueue(start)
    while(!PQ.empty())
    {
        curr = PQ.dequeue()
        if(curr == goal) {return} \ \done!
        curr.visited = true

        for each unvisited neighbor n of curr:
        {
            if(n.cost > curr.cost + edgeweight(curr, n))
            {
                PQ.enqueue(n) || update n’s position in PQ
                n.cameFrom = curr
                n.cost = curr.cost + edgeweight(curr, n)
            }
        }
    }
}
-worst-case complexity of Dijkstra’s
  - we visit each node once (and only once)
  - we cross each edge once (and only once)
  - the work done is at each node and edge is $O(\log V)$

- $O(V + E) \times O(\log V)$

-more details later after we’ve talked about heaps
storing graphs
adjacency list

- each vertex stores an array of adjacent vertex indices, usually as a linked list

- confirming the existence of an edge for a vertex is $O(E_v)$*

- iterating through all of the edges is $O(E)$

- space complexity is $O(V + E)$

- used for sparse graphs (common case)

* $E_v$ refers to the edges for a specific vertex $V$
adjacency matrix

- matrix that represents vertices as columns and rows, with entry containing a 1 if an edge exists, and a 0 otherwise

- confirming the existence of an edge for a vertex is $O(c)$

- iterating through all of the edges is $O(V^2)$

- space complexity is $O(V^2)$

- used for dense graphs
Pacman HW

- is a sort of “implied” adjacency list
- uses a 2D array to store vertices
- adjacent vertices are computed based on indices
- useful for the assignment, but not a standard graph implementation
- you will have a chance to implement a different graph data structure in the last assignment!