Announcements

- A6 will be available tonight or tomorrow.
- Gries lunch canceled today
- Gries office hours today from 1 to 2 only

Q. Why do programmers confuse Halloween and Christmas?
   Answer. Because oct 31 = dec 25!

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
</tr>
<tr>
<td>05</td>
<td>05</td>
</tr>
<tr>
<td>06</td>
<td>06</td>
</tr>
<tr>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>25</td>
</tr>
<tr>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>28</td>
<td>30</td>
</tr>
<tr>
<td>29</td>
<td>31</td>
</tr>
</tbody>
</table>

Where did I leave that book?

Go as far down a path as possible before backtracking – Depth-First Search
Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra’s algorithm
- Minimum spanning trees
  - Prim’s algorithm
  - Kruskal’s algorithm

Representations of Graphs

Adjacency Matrix or Adjacency List?

- Definitions:
  - \( n \) = number of vertices
  - \( m \) = number of edges
  - \( d(u) \) = degree of \( u \) = number of edges leaving \( u \)
- Adjacency Matrix
  - Uses space \( O(n^2) \)
  - Can iterate over all edges in time \( O(n^2) \)
  - Can answer “Is there an edge from \( u \) to \( v \)?” in \( O(1) \) time
  - Better for dense graphs (lots of edges)
- Adjacency List
  - Uses space \( O(m + n) \)
  - Can iterate over all edges in time \( O(m + n) \)
  - Can answer “Is there an edge from \( u \) to \( v \)?” in \( O(d(u)) \) time
  - Better for sparse graphs (fewer edges)

Depth-First Search

- Given a graph and one of its nodes \( u \)
  (say node 1 below)
- We want to “visit” each node reachable from \( u \)
  (nodes 1, 0, 2, 3, 5)

There are many paths to some nodes.
How do we visit all nodes efficiently, without doing extra work?

Suppose all nodes are unvisited.

- Node \( u \) is visited means visited[\( u \)] is true
- To visit \( u \) means to: set visited[\( u \)] to true
- Node \( v \) is REACHABLE from node \( u \) if there is a path \((u, ..., v)\) in which all nodes of the path are unvisited.
Suppose all nodes are unvisited.
 Nodes REACHABLE from node 1: {1, 0, 2, 3, 5}
Depth-First Search

/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */
public static void dfs(int u) {
}

Let u be 1
The nodes REACHABLE from 1 are 1, 0, 2, 3, 5

Suppose the for loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order: 1, ...
Depth-First Search

/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}

Example: Use different way (other than array visited) to know whether a node has been visited
Example: We really haven’t said what data structures are used to implement the graph

That’s all there is to basic DFS. You may have to change it to fit a particular situation.
If you don’t have this spec and you do something different, it’s probably wrong.

Depth-First Search in OO fashion

public class Node {
    boolean visited;
    List<Node> neighbors;

    /** This node is unvisited. Visit all nodes REACHABLE from this node */
    public void dfs() {
        visited= true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
Depth-First Search written iteratively

```java
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s = new Stack(); // Not Java!
    // inv: all nodes that have to be visited are
    // REACHABLE from some node in s
    while (s is not empty) {
        u = s.pop(); // Remove top stack node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call `dfs(1)` Iteration 0

Stack s

```
Depth-First Search written iteratively

```
Depth-First Search written iteratively
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s = (u);
    while (s is not empty) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1) Iteration 1
Stack s

Depth-First Search written iteratively
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s = (u);
    while (s is not empty) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1) Iteration 2
Stack s
Depth-First Search written iteratively

```java
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s = (u);
    while (!s.isEmpty()) {  // Not Java!
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Breadth-First Search

```java
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);
    while (!q.isEmpty()) {  // Not Java!
        u = q.popFirst();  // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);  // Add to end of queue
        }
    }
}
```

Breadth-First Search

```java
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);
    while (!q.isEmpty()) {  // Not Java!
        u = q.popFirst();  // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);  // Add to end of queue
        }
    }
}
```
Breadth-First Search
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u)
{
    Queue q = (u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)
Queue q
0 2
Iteration 0

Call bfs(1)
Queue q
2
Iteration 1

Call bfs(1)
Queue q
2 7
Iteration 2
Breadth-First Search
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1) Iteration 2
Queue q

Breadth-First Search
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1) Iteration 2
Queue q

Breadth-First Search
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1) Iteration 2
Queue q