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 \( \text{oct 31} = \text{dec 25}! \)

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

Adjacency Matrix

1  0  1  0  1
2  0  0  1  0
3  0  0  0  0
4  0  1  1  0
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)
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?
Depth-First Search

boolean[ ] visited;

- 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.
Depth-First Search

boolean[ ] visited;

- 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

boolean[ ] visited;

- 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\}

Nodes REACHABLE from 4: \{4, 5, 6\}
Depth-First Search

boolean[ ] visited;

• 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.

Green: visited
Blue: unvisited
Depth-First Search

boolean[ ] visited;

- 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.

**Green:** visited
**Blue:** unvisited

Nodes REACHABLE from node 1: {1, 0, 5}
Depth-First Search

boolean[ ] visited;

- Node \( u \) is visited means: \( \text{visited}[u] \) is true
- To visit \( u \) means to: set \( \text{visited}[u] \) to true
- Node \( v \) is REACHABLE from node \( u \) if there is a path \((u, \ldots, v)\) in which all nodes of the path are unvisited.

Green: visited
Blue: unvisited

Nodes REACHABLE from node 1:
\{1, 0, 5\}

Nodes REACHABLE from 4: none

Not even 4 itself, because it’s already been visited!
Depth-First Search

/** Visit all nodes that are REACHABLE from u. 
   Precondition: u is not visited */
public static void dfs(int u) {

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

1 —> 2 —> 3
0 —> 5 —> 6

1 —> 2 —> 3
0 —> 5 —> 6
/** Node \(u\) is unvisited. Visit all nodes that are REACHABLE from \(u\). */

```java
public static void dfs(int u) {
    // Implementation
}
```

Let \(u\) be 1

The nodes REACHABLE from 1 are 1, 0, 2, 3, 5
/** Node u is unvisited. Visit all nodes that are \textbf{REACHABLE} from u. */

```java
public static void dfs(int u) {
    visited[u] = true;
}
```

Let \( u \) be 1

The nodes \textbf{REACHABLE} from 1 are 1, 0, 2, 3, 5
/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */

public static void dfs(int u) {
    visited[u] = true;
}

Let u be 1 (visited)
The nodes to be visited are 0, 2, 3, 5
/** 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);
    }

Let u be 1 (visited)
The nodes to be visited are 0, 2, 3, 5
Have to do DFS on all unvisited neighbors of u!
/** Node \( u \) is unvisited. Visit all nodes that are REACHABLE from \( u \). */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges \((u, v)\) leaving \( u \):
        if \( v \) is unvisited then dfs(v);
}
```

Suppose the `for` loop visits neighbors in numerical order. Then `dfs(1)` visits the nodes in this order: 1 ...
/** 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);
}

Suppose the for loop visits neighbors in numerical order. Then \texttt{dfs(1)} visits the nodes in this order: 1, 0 ...
/** 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);
}

Suppose the for loop visits neighbors in numerical order. Then \texttt{dfs(1)} visits the nodes in this order: 1, 0, 2 ...
Depth-First Search

/** Node \textit{u} is unvisited. Visit all nodes that are \text{REACHABLE} from \textit{u}. */

public static void dfs(int \textit{u}) {
    visited[\textit{u}] = true;
    for all edges \((\textit{u}, \textit{v})\) leaving \textit{u}:
        if \textit{v} is unvisited then dfs(\textit{v});
}

Suppose the \textbf{for} loop visits neighbors in numerical order. Then \texttt{dfs(1)} visits the nodes in this order: 1, 0, 2, 3 ...
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);
}

Suppose the for loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order: 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) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}

Suppose \( n \) nodes are REACHABLE along \( e \) edges (in total). What is
• Worst-case execution?
• Worst-case space?
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

```java
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();
        }
    }
}
```

Each node of the graph is an object of type Node

No need for a parameter. The object is the node.
Depth-First Search written iteratively

/*** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s = (u);  // 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);
        }
    }
}
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)

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 0

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 = new Stack(u);
    while (!s.isEmpty()) {
        int 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 0

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 = new Stack(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 0

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 = new Stack();
    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 0

Stack s
0
2
5
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
0
2
5

0
2
5

Stack s
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (!s.isEmpty()) {
        int u = s.pop();
        if (!visited.contains(u)) {
            // Visit u
            visited.add(u);
            for (int v : neighbors.get(u)) {
                s.push(v);
            }
        }
    }
}
```

Call `dfs(1)`

Iteration 1

```
Stack s
```

2
5

Stack `s`
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
public static void dfs(int u) {
    Stack s = new Stack(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**

```
1 -> 2 -> 3
0 -> 5 -> 4
```

2
5

Stack s
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    while (!s.isEmpty()) {
        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`

```
nodes: 1, 2, 3, 4, 5, 6
edges: (1, 2), (1, 3), (1, 5), (2, 6), (3, 4)
```

2
5

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
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
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

public static void dfs(int u) {
    Stack s = new Stack(u);
    while (!s.isEmpty()) {
        u = s.pop();
        if (!u.hasBeenVisited()) {
            visit(u);
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  Iteration 2

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

Yes, 5 is put on the stack twice, once for each edge to it. It will be visited only once.

Stack s

3
5
5
Breadth-First Search

/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void bfs(int u) {
    Queue q = (u);  // Not Java!
    // inv: all nodes that have to be visited are
    //      REACHABLE from some node in 
    while (q is not empty) {
        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)

1

Queue q
Breadth-First Search

/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
public static void bfs(int u) {
    Queue q = new Queue(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 0

![Graph showing BFS traversal]

1  
Queue q
/** 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);
        }
    }
}
Breadth-First Search

/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
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 0

Queue q

Diagram:
- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7

Connections:
- 1 → 2 → 3
- 1 → 2 → 5 → 6
- 1 → 2 → 5
- 1 → 0
- 1 → 7
- 2 → 3
- 2 → 5
- 2 → 6
- 3 → 4
- 4 → 5
- 5 → 6
Breadth-First Search

/**
 * Node u is unvisited. Visit all nodes REACHABLE from u.
 */

```java
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 0

0 2
Queue q
Breadth-First Search

/** Node u is unvisited. Visit all nodes REACHABLE from u. */

public static void bfs(int u) {
    Queue q = new Queue(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 1

```
Queue q
0 2
```

```
0 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 1

Queue q

2

1

0

7

3

5

4

6
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 1

Queue q

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 1
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);
        }
    }
Breadth-First Search

/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
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

7
/** Node u is unvisited. Visit all nodes REACHABLE from u. */

```java
public static void bfs(int u) {
    Queue q = new Queue(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)`

Instruction:
- **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)

Breadth first:
(1) Node u
(2) All nodes 1 edge from u
(3) All nodes 2 edges from u
(4) All nodes 3 edges from u
...

7 3 5
Queue q