Kinds of graphs
A graph (undirected)

\[ G = (V, E) \]

\[ V = \{a, b, c, d, e, f, g, h, i, j, k, \ell\} \]

\[ E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, f\}, \{b, d\}, \{c, f\}, \]
\[ \{c, h\}, \{c, j\}, \{d, g\}, \{e, g\}, \{e, i\}, \{e, m\}, \]
\[ \{f, g\}, \{f, j\}, \{g, j\}, \{g, k\}, \{h, i\}, \{h, j\}, \{i, j\}\} \]
A directed graph

\[ G = (V, E) \]

\[ V = \{a, b, c, d, e, f\} \]

\[ E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\} \]
A directed graph

\[ G = (V, E) \]

\[ V = \{a, b, c, d, e, f\} \]

\[ E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\} \]
A directed graph with cycles

\[ G = (V, E) \]

\[ V = \{a, b, c, d, e, f\} \]

\[ E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\} \]
A directed graph with cycles

\[ G = (V, E) \]
\[ V = \{a, b, c, d, e, f\} \]
\[ E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\} \]
A DAG (directed acyclic graph)
A weighted, directed graph

\[ G = (V, E, w) \]

\[ V = \{a, b, c, d, e, f\} \]

\[ E = \{(a, b), (b, c), (c, d), (c, f), (d, d), (d, e), (e, f), (f, e)\} \]

\[ w = \{(a, b) \mapsto 1, (b, c) \mapsto 2, (c, d) \mapsto 1, (c, f) \mapsto 12, \ldots\} \]
A little graph theory
Some graph definitions

If \( \{v, u\} \in E \) then \( v \) and \( u \) are adjacent.

If \( \{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\} \in E \) then there is a path from \( v_0 \) to \( v_k \), and we say \( v_0 \) and \( v_k \) are connected.
Some graph definitions

If \( \{v, u\} \in E \) then \( v \) and \( u \) are \textit{adjacent}
Some graph definitions

If \( \{v, u\} \in E \) then \( v \) and \( u \) are adjacent

If \( \{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\} \in E \) then there is a path from \( v_0 \) to \( v_k \), and we say \( v_0 \) and \( v_k \) are connected
A subgraph of nodes all connected to each other is a *connected component*; here we have two
Degree

The degree of a vertex is the number of adjacent vertices:

$$\text{degree}(\nu, G) = |\{u \in V : \{u, \nu\} \in E\}|$$ where $G = (V, E)$
Degree

The degree of a vertex is the number of adjacent vertices:

$$\text{degree}(v, G) = \left| \{ u \in V : \{u, v\} \in E \} \right| \text{ where } G = (V, E)$$

The degree of a graph is the maximum degree of any vertex:

$$\text{degree}(G) = \max_{v \in V} \text{degree}(v, G) \text{ where } G = (V, E)$$
Degree

The degree of a vertex is the number of adjacent vertices:

\[
\text{degree}(v, G) = \left| \{u \in V : \{u, v\} \in E\} \right| \text{ where } G = (V, E)
\]

The degree of a graph is the maximum degree of any vertex:

\[
\text{degree}(G) = \max_{v \in V} \text{degree}(v, G) \text{ where } G = (V, E)
\]

Sometimes we will refer to the degree as \(d\), such as when we say that a particular operation is \(\mathcal{O}(d)\).
Some digraph definitions

If \((v, u) \in E\), \(v\) is the direct predecessor of \(u\) and \(u\) is the direct successor of \(v\).

If \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E\) then there is a path from \(v_0\) to \(v_k\); we say that \(v_k\) is reachable from \(v_0\).

If \(v_k\) and \(v_0\) are mutually reachable from each other, they are strongly connected.
Some digraph definitions

If \((v, u) \in E\), \(v\) is the direct predecessor of \(u\) and \(u\) is the direct successor of \(v\)
If \((v, u) \in E\), \(v\) is the direct predecessor of \(u\) and \(u\) is the direct successor of \(v\)

If \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E\) then there is a path from \(v_0\) to \(v_k\); we say that \(v_k\) is reachable from \(v_0\)
Some digraph definitions

If \((v, u) \in E\), \(v\) is the *direct predecessor* of \(u\) and \(u\) is the *direct successor* of \(v\)

If \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E\) then there is a *path* from \(v_0\) to \(v_k\); we say that \(v_k\) is *reachable* from \(v_0\)
Some digraph definitions

If \((v, u) \in E\), \(v\) is the **direct predecessor** of \(u\) and \(u\) is the **direct successor** of \(v\).

If \((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E\) then there is a **path** from \(v_0\) to \(v_k\); we say that \(v_k\) is **reachable** from \(v_0\).

If \(v_k\) and \(v_0\) are mutually reachable from each other, they are **strongly connected**.
In a digraph, a subgraph of vertices all strongly connected to each other is a *strongly connected component*; here we have a connected graph with two SCCs.
Dense versus sparse

1
2
3
4
5
6
7

1
2
3
4
5
6
7

1
2
3
4
5
6
7
Programming with graphs
A graph ADT

Looks like $(V, E)$ (as above)

Operations:

```python
interface GRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, u: nat?, v: nat?) -> VoidC
    def has_edge?(self, u: nat?, v: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_neighbors(self, v: nat?) -> VertexSet
```
A graph ADT

Looks like \((V, E)\) (as above)

Operations:

```
interface GRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, u: nat?, v: nat?) -> VoidC
    def has_edge?(self, u: nat?, v: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_neighbors(self, v: nat?) -> VertexSet
```

Invariants:

- \(V = \{0, 1, \ldots, |V| - 1\}\)
- \(\bigcup E \subseteq V\)
Graph ADT laws

1. \( \{g = (V, E)\} \ g.\text{new\_vertex}() = n \ \{g = (V \cup \{n\}, E)\} \)
   where \( n = \max(V) + 1 \)

2. \( \{g = (V, E) \land n, m \in V\} \ g.\text{add\_edge}(n, m) \ \{g = (V, E \cup \{\{n, m\}\})\} \)

3. \( \{g = (V, E) \land \{n, m\} \in E\} \ g.\text{has\_edge?}(n, m) = \top \)

4. \( \{g = (V, E) \land \{n, m\} \notin E\} \ g.\text{has\_edge?}(n, m) = \bot \)

5. \( \{g = (V, E)\} \ g.\text{get\_vertices}() = V \)

6. \( \{g = (V, E)\} \ g.\text{get\_neighbors}(n) = \{m \in V : \{m, n\} \in E\} \)
A digraph ADT

Looks like $(V, E)$ (as above, $E$ contains ordered pairs of vertices)

Operations:

```
interface DIGRAPH:
    def new_vertex(self) -> nat?
    def add_edge(self, src: nat?, dst: nat?) -> VoidC
    def has_edge?(self, src: nat?, dst: nat?) -> bool?
    def get_vertices(self) -> VertexSet
    def get_succs(self, v: nat?) -> VertexSet
    def get_preds(self, v: nat?) -> VertexSet
```

Invariants:

- $V = \{0, 1, \ldots, |V| - 1\}$
- $\forall (v, u) \in E. \ v \in V \land u \in V$
Digraph ADT laws

1. \( \{g = (V, E)\} \quad g.new\_vertex() = n \quad \{g = (V \cup \{n\}, E)\} \)
   where \( n = \max(V) + 1 \)

2. \( \{g = (V, E) \land n, m \in V\} \quad g.add\_edge(n, m) \quad \{g = (V, E \cup \{(n, m)\})\} \)

3. \( \{g = (V, E) \land (n, m) \in E\} \quad g.has\_edge(n, m) = \top \)

4. \( \{g = (V, E) \land (n, m) \notin E\} \quad g.has\_edge(n, m) = \bot \)

5. \( \{g = (V, E)\} \quad g.get\_vertices() = V \)

6. \( \{g = (V, E)\} \quad g.get\_succs(n) = \{m \in V : (n, m) \in E\} \)

7. \( \{g = (V, E)\} \quad g.get\_preds(n) = \{m \in V : (m, n) \in E\} \)
A weighted digraph ADT

Looks like $(V, E, w)$ (as above)

Operations:

```
let weight? = OrC(num?, inf)
```

```
interface WDIGRAPH:
    def new_vertex(self) -> nat?
    def set_edge(self, src: nat?, w: weight?,
                  dst: nat?) -> VoidC
    def get_edge(self, src: nat?, dst: nat?) -> weight?
    def get_vertices(self) -> VertexSet
    def get_succs(self, v: nat?) -> VertexSet
    def get_preds(self, v: nat?) -> VertexSet
```
Weighted digraph ADT laws

1. \( \{ g = (V, E, w) \} \) \( g.\text{new\_vertex}() = n \) \( \{ g = (V \cup \{n\}, E, w) \} \)
   where \( n = \max(V) + 1 \)
2. \( \{ g = (V, E, w) \wedge n, m \in V \} \) \( g.\text{set\_edge}(n, a, m) \) \( \{ g = (V, E \cup \{(n, m)\}, w\{(n, m) \mapsto a\}) \} \)
   where \( a < \infty \)
3. \( \{ g = (V, E, w) \wedge n, m \in V \} \) \( g.\text{set\_edge}(n, \infty, m) \) \( \{ g = (V, E \setminus \{(n, m)\}, w \setminus \{(n, m)\}) \} \)
4. \( \{ g = (V, E, w) \wedge (n, m) \in E \} \) \( g.\text{get\_edge}(n, m) = w(n, m) \)
5. \( \{ g = (V, E, w) \wedge (n, m) \notin E \} \) \( g.\text{get\_edge}(n, m) = \infty \)
6. \( \{ g = (V, E, w) \} \) \( g.\text{get\_vertices}(g) = V \)
7. \( \{ g = (V, E, w) \} \) \( g.\text{get\_succe}(n) = \{ m \in V : (n, m) \in E \} \)
8. \( \{ g = (V, E, w) \} \) \( g.\text{get\_preds}(n) = \{ m \in V : (m, n) \in E \} \)
Graph representation
Two graph representations

There are two common ways that graphs are represented on a computer:

- adjacency list
- adjacency matrix
Adjacency list

In an array, store a list of neighbors (or successors) for each vertex:
Store a $|V|$-by-$|V|$ matrix of Booleans indicating where edges are present:

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0
\end{array}
\]
A directed adjacency matrix example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
With weights

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>−4</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>10</td>
<td>8</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>4</td>
<td>5</td>
<td>∞</td>
</tr>
</tbody>
</table>
Space comparison

Adjacency list—has a list for each vertex, and the total length of all the lists is the number of edges: $\mathcal{O}(V + E)$

Adjacency matrix—is $|V|$ by $|V|$ regardless of the number of edges: $\mathcal{O}(V^2)$
Space comparison

Adjacency list—has a list for each vertex, and the total length of all the lists is the number of edges:
\[ O(V + E) \]

Adjacency matrix—is \(|V|\) by \(|V|\) regardless of the number of edges:
\[ O(V^2) \]

When might we want to use one or the other?
Time comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adj. List</th>
<th>Adj. Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_edge/set_edge</td>
<td>O(setInsert(d))</td>
<td>O(1)</td>
</tr>
<tr>
<td>get_edge/has_edge?</td>
<td>O(setLookup(d))</td>
<td>O(1)</td>
</tr>
<tr>
<td>get_succs</td>
<td>O(</td>
<td>Result</td>
</tr>
<tr>
<td>get_preds</td>
<td>O(V+E)</td>
<td>O(V)</td>
</tr>
</tbody>
</table>
Time comparison

<table>
<thead>
<tr>
<th></th>
<th>adj. list</th>
<th>adj. matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_edge/set_edge</td>
<td>$\mathcal{O}(\text{setInsert}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>
## Time comparison

<table>
<thead>
<tr>
<th></th>
<th>Adj. List</th>
<th>Adj. Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add_edge/set_edge</strong></td>
<td>$\mathcal{O}(\text{setInsert}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><strong>get_edge/has_edge?</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Time comparison

<table>
<thead>
<tr>
<th></th>
<th>adj. list</th>
<th>adj. matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_edge/set_edge</code></td>
<td>$\mathcal{O}(\text{setInsert}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>get_edge/has_edge?</code></td>
<td>$\mathcal{O}(\text{setLookup}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>
### Time comparison

<table>
<thead>
<tr>
<th>Function</th>
<th>Adj. list</th>
<th>Adj. Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_edge/set_edge</td>
<td>$\mathcal{O}(\text{setInsert}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>get_edge/has_edge?</td>
<td>$\mathcal{O}(\text{setLookup}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>get_succs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Time comparison

<table>
<thead>
<tr>
<th></th>
<th>adj. list</th>
<th>adj. matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_edge/set_edge</code></td>
<td>$\mathcal{O}(\text{setInsert}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>get_edge/has_edge?</code></td>
<td>$\mathcal{O}(\text{setLookup}(d))$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>get_succs</code></td>
<td>$\mathcal{O}(</td>
<td>\text{Result}</td>
</tr>
</tbody>
</table>
## Time Comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adj. List</th>
<th>Adj. Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_edge/set_edge</td>
<td>(O(\text{setInsert}(d)))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>get_edge/has_edge?</td>
<td>(O(\text{setLookup}(d)))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>get_succs</td>
<td>(O(</td>
<td>\text{Result}</td>
</tr>
<tr>
<td>get_preds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Adj. List</td>
<td>Adj. Matrix</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>add_edge/set_edge</td>
<td>$O(setInsert(d))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get_edge/has_edge?</td>
<td>$O(setLookup(d))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get_succs</td>
<td>$O(</td>
<td>Result</td>
</tr>
<tr>
<td>get_preds</td>
<td>$O(V + E)$</td>
<td>$O(V)$</td>
</tr>
</tbody>
</table>
Next time: exam review