1 Levels of Algorithm Design

- **Naive Algorithms:** Turn definition into algorithm. This is easy to wirte, good first pass, but often very slow. Good way to "test."
- **Toolkit:** Algorithms designed using standard tools the main focus of this course.
- **Optimized:** Use data structures or other ideas to make algorithm especially efficient.
- **Magic:** Sometimes, an algorithm requires a surprising new insight.

2 Graph

Definition 2.1: Graph

A **graph** $G = (V, E)$ consists of two things:

- A collection V of vertices, or objects to be connected.
- A collection E of edges, each of which connects a pair of vertices.

For example, we can model the following as graphs:

- The internet, where V is the websites and E are links.
- The internet, where V are computers and E are physical connections.
- A highway system, where V are the intersections and E are the roads.

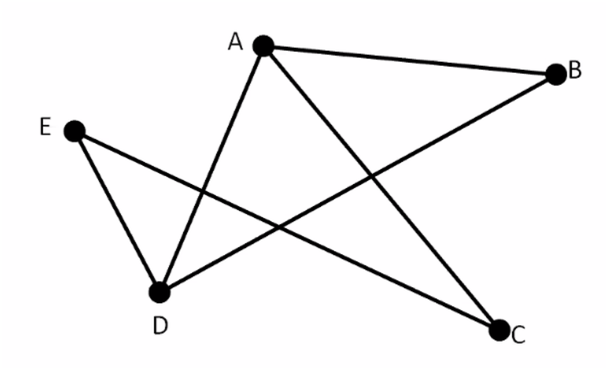
2.1 Examples of Graphs in CS

In computer science, there are a few examples of graphs:

- The internet (webpages, physical connections, etc.)
- Social networks (especially with friendship, connections).
- Transitions between states of a program.
- Road maps.

2.2 Drawing Graphs

When drawing graphs, we want to draw the vertices first. Then, for each edge, draw line segments or curves connecting those points.



This particular graph can be represented by $G = (V, E)$ where:

$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, E), (D, E)\}$$

Because this is an unordered graph, (A, B) and (B, A) mean the same thing.

2.3 Exploring Graphs

Suppose we are playing a video game and want to make sure that you've found all the areas in this level before moving on to the next one. How do we ensure that we found everything?

2.3.1 Basic Algorithm

Essentially, we want to:

```
Keep track of all areas discovered.
While there is an unexplored path:
    Follow path.
```

2.3.2 Systematize

Essentially, we need to keep track of:

- Which vertices we have discovered.
- Which edges have yet to be explored.

So, the explore algorithm will:

- Use a field `v.visted` to let us know which vertices we have seen.
- Store edges to be explored implicitly in the program stack.

```
explore(v):
    v.visited <- true
    for each edge (v, w):
        if not w.visited:
            explore(w)
            w.prev <- v     // If we want to keep track of path taken
```

2.3.3 Result

Theorem 2.1

If all vertices start unvisited, `explore(v)` marks as visited exactly the vertices reachable from `v`.

Proof. First, we note that we can only visit vertices that are reachable from v. If u is visited, then eventally we will visit every adjacent w. If there is a chain of vertices, then v will visit u_1 which will visit u_2 which will visit everything up to and including w. □

2.4 Depth First Search

`explore` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times. This introduces an algorithm known as **depth first search**:

```
DepthFirstSearch(v):
    Mark all v in G as unvisited.
    For v in G:
        if not v.visited:
            explore(v)
```

Here, this runs in $O(|V|)$ time: we need to iterate over vertex.

2.5 Runtime of DFS

```
explore(v):                 // Run once per vertex O(|V|)
    v.visited <- true       // Run once per vertex
    for each edge (v, w):   // Run once per neighboring vertex O(|E|)
        if not w.visited:   // Run once per neighboring vertex
            explore(w)      // Run once per neighboring vertex
            w.prev <- v     // Run once per neighboring vertex
```

So, our final runtime is:

$$O(2|V| + |E|) = O(|V| + |E|)$$