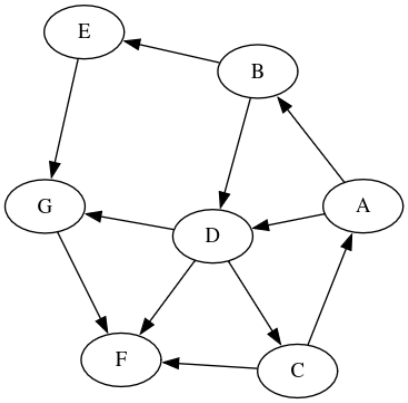
1. Carry out depth-first search on the graph shown below. Use the recursive DFS discussed in class, not a stack-based implementation. Start from vertex A. Use alphabetical order to break ties.

   Indicate the order in which vertices are marked as "discovered" and as "processed". Showing more of your work, such as a tree of the recursive calls, can help you earn partial credit in case you make a mistake.



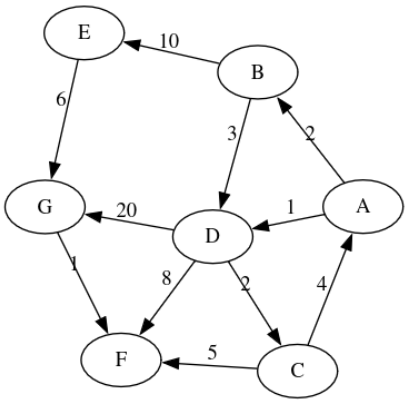

Trace:

```
dfs(A) - B, D
         |  |
         |  X
         |
        dfs(B) - D, E
                 |  |
                 | dfs(E) - G
                 |          |
                 |          X
                 |
                dfs(D) - C, F, G
                         |  |  |
                         |  X dfs(G) - F
                         |             |
                         |             X
                        dfs(C) - A, F
                                 |  |
                                 X dfs(F)
```

Discovered order: A, B, D, C, F, G, E
Processed order: F, C, G, D, E, B, A

2. Carry out Dijkstra's algorithm on the graph shown below. Start from vertex A. Use alphabetical order to break ties.

   Give the final `dist` labels for each vertex. Also show your work by indicating the order in which vertices are removed from the priority queue and showing a table of the `dist` labels as they are updated in each iteration.




   Start with `dist` labels initialized to 0 for the start vertex (A) and $\infty$ for everything else. (First row below.) All vertices are in the priority queue, which is ordered by `dist` value.

   The main loop removes the next vertex from the PQ and updates the `dist` labels of its neighbors if a new shortest path has been found. To make it easier to keep track of what is in the PQ, only the labels for things still in the PQ are shown.

| removed from PQ | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| A | | 2 | $\infty$ | 1 | $\infty$ | $\infty$ | $\infty$ |
| D | | 2 | 3 | | $\infty$ | 9 | 21 |
| B | | | 3 | | 12 | 9 | 21 |
| C | | | | | 12 | 8 | 21 |
| F | | | | | 12 | | 21 |
| E | | | | | | | 18 |
| G | | | | | | | |

   Final labels —

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 1 | 12 | 8 | 18 |

   Order of removal from the PQ: A, D, B, C, F, E, G

3. For each of the following scenarios, explain how to solve the problem by modeling it as a graph problem — address the points below.

   - A public health organization needs to set up emergency medical supply routes between a group of remote villages in a mountainous region. Each potential path between two villages takes a different amount of time and effort to maintain due to terrain and distance. The organization hopes to ensure that every village is reachable from any other by some route but has a limited budget. Given the organization's budget and the cost of each potential path between two villages, determine whether it is possible to connect all of the villages and, if so, how to do it.

     (a) Graph representation.
     - *What do the vertices of the graph represent?*
       villages
     - *What do the edges of the graph represent?*
       paths between villages
     - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
       undirected — nothing indicates that paths can only be traveled in one direction
     - *Is the graph weighted or unweighted? If weighted, what are the weights?*
       weighted — cost of the path
     - *What is the solution to the problem in terms of the graph?*
       minimum cost connectivity — minimum spanning tree

     (b) Properties of the graph. Provide a brief explanation with each answer.
     - *Is the graph simple or not simple?*
       could be either — it is not likely that there are self-loops (paths not connecting different villages) but there could be multiple routes between a given pair of villages
     - *Is the graph sparse or dense?*
       most likely sparse — paths are likely to only connect villages to their closest neighbors
     - *Is the graph cyclic or acyclic?*
       could be either, though if it is acyclic there wouldn't be any need to find the minimum spanning tree

     (c) *Algorithm.*
     Minimum cost connectivity is minimum spanning tree. If the MST is a forest rather than a tree, it is not possible to connect all of the villages. Kruskal's algorithm is more convenient than Prim's when the graph may be disconnected.

- After a storm knocks out some roads, how many separate groups of towns are left where each town is reachable from the others?
  (a) Graph representation.
    - *What do the vertices of the graph represent?*
      towns
    - *What do the edges of the graph represent?*
      the existence of at least one road between a pair of towns
    - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
      presumably undirected — even if there were one-way roads originally, exceptions would probably be made for emergency vehicles
    - *Is the graph weighted or unweighted? If weighted, what are the weights?*
      unweighted — a road is either passable or not
    - *What is the solution to the problem in terms of the graph?*
      groups of towns reachable from each other — connected components

  (b) Properties of the graph. Provide a brief explanation with each answer.
    - *Is the graph simple or not simple?*
      not simple — likely no self-loops, but potentially multiple roads between at least some towns
    - *Is the graph sparse or dense?*
      likely sparse — likely to only be direct roads from one town to nearby towns
    - *Is the graph cyclic or acyclic?*
      likely cyclic — it would be unusual to have only the barest minimum of roads between towns

  (c) *Algorithm.*
      This is connected components — determine the connected components and those are the groups of towns reachable from each other.
- Software packages may depend on components provided by other packages which must be installed first. For a collection of packages, are there circular dependencies that would prevent installation?
  (a) Graph representation.
    - *What do the vertices of the graph represent?*
      software packages
    - *What do the edges of the graph represent?*
      dependencies between packages
    - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
      directed, from a package to what depends on it e.g. $u \to v$ means that $u$ must be installed before $v$.

  - *Is the graph weighted or unweighted? If weighted, what are the weights?*
    unweighted — either a dependency exists or it doesn't
  - *What is the solution to the problem in terms of the graph?*
    whether or not there is a directed cycle

(b) Properties of the graph. Provide a brief explanation with each answer.

  - *Is the graph simple or not simple?*
    simple — packages can't depend on themselves, and there's either a dependency between two packages or there isn't (mo multiedges)
  - *Is the graph sparse or dense?*
    presumably sparse — if every package depends on most of the other packages, it is unlikekly there will be a successful installation order (not to mention a very poorly-designed system)
  - *Is the graph cyclic or acyclic?*
    that is what is to be determined

(c) *Algorithm.*
Circular dependencies = directed cycle. Topological sort would find an ordering of the packages respecting the dependencies if there is one, so report that there are circular dependencies if topological sort fails.

- In a communications network, are there any critical elements — a server or a cable — which would split the network if they failed or were removed?

(a) Graph representation.

  - *What do the vertices of the graph represent?*
    servers
  - *What do the edges of the graph represent?*
    cables between servers
  - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
    undirected — presumably traffic goes both ways over network links
  - *Is the graph weighted or unweighted? If weighted, what are the weights?*
    unweighted — only the existence (or not) of a link matters
  - *What is the solution to the problem in terms of the graph?*
    vertices or edges whose removal would disconnect the graph

(b) Properties of the graph. Provide a brief explanation with each answer.

  - *Is the graph simple or not simple?*
    not simple — it is possible there are multiple cables between the same set of servers, and that is important info to capture because the failure of one cable wouldn't then disconnect those servers

  - *Is the graph sparse or dense?*
    likely sparse — typically only nearby servers would be connected directly
  - *Is the graph cyclic or acyclic?*
    presumably cyclic, otherwise every edge is a single point of failure, but nothing requires it to be

(c) *Algorithm.*
Single points of failure = cut vertices and cut edges. Algorithms for both of those were discussed in class.

- In a city's public transportation network, the cost of a route depends on the number of transfers rather than the distance traveled. Find routes with the minimum number of transfers.

  (a) Graph representation.
  - *What do the vertices of the graph represent?*
    stops on particular bus/train routes
  - *What do the edges of the graph represent?*
    both the routes travelled by buses/trains and transfers — connect consecutive stops on a particular bus/train routes and geographically co-located stops on different routes that require a transfer
  - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
    directed — while there may well be bus/train routes connecting adjacent stops in both directions, there are often different stops or platforms for buses/trains travelling in different directions
  - *Is the graph weighted or unweighted? If weighted, what are the weights?*
    weighted — 1 for transfer edges, 0 for route edges
  - *What is the solution to the problem in terms of the graph?*
    since only transfer edges incur a cost, a route with the minimum number of transfers = a min-cost path between stops

  (b) Properties of the graph. Provide a brief explanation with each answer.
  - *Is the graph simple or not simple?*
    simple — since we duplicate stops to account for transfers when different bus/train routes pass through the same place, there won't be multiedges
  - *Is the graph sparse or dense?*
    sparse — each bus/train route generally only includes a stop once, and while the transfer edges introduce dense subgraphs around co-located stops, most stops are typically only served by one bus/train route

  - *Is the graph cyclic or acyclic?*
    most likely cyclic, but it depends — bus routes are more likely to be loops while train routes are more likely to be out-and-back branches, but there can also be central loops on trains; transfer points can also introduce cycles if more than two routes share the same stop

(c) *Algorithm.*
By setting up the graph so that the desired cost of a path is the sum of its weights, this is now a straightforward weighted shortest path task solvable with Dijkstra's algorithm.

- You are organizing a conference with a number of sessions. When registering for the conference, participants were asked for up to two sessions that they most wanted to attend. Sessions can be scheduled in either the morning or the afternoon, and participants can only attend one session in each time slot. Determine how to schedule the sessions so that every participant is able to attend their preferred sessions, or determine that such a schedule is not possible.

  (a) Graph representation.
  - *What do the vertices of the graph represent?*
    sessions
  - *What do the edges of the graph represent?*
    a pair of sessions at least one person wants to attend
  - *Is the graph undirected or directed? If directed, what does the direction of the edges reflect?*
    undirected — edges are the presence of a constraint (preference)
  - *Is the graph weighted or unweighted? If weighted, what are the weights?*
    unweighted — edges are the presence of a constraint (preference)
  - *What is the solution to the problem in terms of the graph?*
    two groups of vertices (morning and afternoon) so that edges only connect vertices in different groups

  (b) Properties of the graph. Provide a brief explanation with each answer.
  - *Is the graph simple or not simple?*
    simple — only include an edge if a participant has two different sessions they want to attend, and edges represent at least one participant wanting those two sessions rather than individual participants (so no multiedges)
  - *Is the graph sparse or dense?*
    depends on the number of participants and their preferences — most likely there are many more participants than sessions so there is the potential for the graph to be dense, and it depends on whether there

are a few popular combinations of sessions that everyone wants or if preferences are more varied

- *Is the graph cyclic or acyclic?*
  depends on the preferences, but there's no reason it wouldn't be cyclic

(c) *Algorithm.*
This is a 2-coloring task — the colors are the session times (morning and afternoon). There's a feasible schedule if and only if a 2-coloring can be found.

4. For each of the following scenarios, address the tradeoffs with using an adjacency list vs an adjacency matrix representation for the graph. Is there a clear winner? Does it not matter? Consider both time and space requirements, and give a brief explanation for your answer.

   (a) The graph has 10 vertices and 20 edges, and you need to frequently use both `areAdjacent` and `adjacentVertices`.

   Time: `areAdjacent` is $O(\min(\deg(u, v))$ for adjacency list and $O(1)$ for adjacency matrix. `adjacentVertices` is $O(\deg(u))$ for adjacency list and $O(n)$ for adjacency matrix.

   Space: 20 edges and 10 vertices means an average degree of 2, which is a lot less than the maximum degree of 9 (for a simple graph) — the graph is sparse. Adjacency list is preferable for sparse graphs.

   Overall: An edge to adjacency list — the only place where adjacency matrix wins in big-Oh is `areAdjacent`, but for a sparse graph $\deg(v) = O(1)$. However, this is a small graph so either could work, depending on other factors — time and space difference matter less for small graphs.

   (b) The graph has 1,000 vertices and 100,000 edges, and you need to frequently use `areAdjacent`.

   Time: `areAdjacent` is $O(\min(\deg(u, v))$ for adjacency list and $O(1)$ for adjacency matrix.

   Space: 100,000 edges and 1,000 vertices means an average degree of 100, which is a lot less than the maximum degree of 999 (for a simple graph) — the graph is sparse. Adjacency list is preferable for sparse graphs.

   Overall: Adjacency matrix, if there is sufficient space — while $O(\min(\deg(u, v)) = O(1)$ for sparse graphs, the impact of constant factors is magnified for large graphs and 100 vs 1 is significant. If not, then adjacency list takes less space. With a large graph, this can be important.

(c) The graph has 1,000 vertices and 800,000 edges, and you need to frequently use `adjacentVertices`.

Time: `adjacentVertices` is $O(\deg(u))$ for adjacency list and $O(n)$ for adjacency matrix.

Space: 800,000 edges and 1,000 vertices means an average degree of 800, which is closer to the maximum degree of 999 (for a simple graph) — the graph is dense. Adjacency matrix is preferable for dense graphs.

Overall: It depends on whether time or space is more important because it comes down to constant factors. Adjacency matrix has an edge in space due to constant factors — adjacency list requires $O(n^2)$ space to store $O(n^2)$ edges but has the extra overhead of list structures. Adjacency list has an edge in time due to constant factors — for dense graphs $O(\deg(u)) = O(n)$ but for simple graphs $\deg(u) < n$. With a large graph constant factors are more significant.