## Chapter 19 Practice Exercises

**Disclaimer:** These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. A complete graph has 10 vertices. How many edges does it have?
   **A)** 10
   **B)** 30
   **C)** 45
   **D)** 55
   **E)** 90

2. True or false? For a weighted, undirected graph, the cost of going from points A to B will always be the same regardless of how you get there.
   **A)** True
   **B)** False

3. For which of the following types of graphs would an adjacency matrix be preferred over an adjacency list?
   **A)** Dense
   **B)** Sparse
   **C)** Directed
   **D)** Undirected
   **E)** Weighted

4. Consider two social media platforms, Facebook and Twitter. On Facebook, if you are friends with another user, that user must also be friends with you. On Twitter, if you follow another user, that user does not need to follow you back. Knowing this, friends on Facebook can be best represented as a _____________ graph, while followers on Twitter can be best represented as a _____________ graph.
   **A)** Directed, directed
   **B)** Directed, undirected
   **C)** Undirected, directed
   **D)** Undirected, undirected
   **E)** None of the above

5. Which one of the following graphs is directed?
   **A)** A graph where the vertices represent people at an event, and the edges represent handshakes that have occurred between two people
   **B)** A graph where the vertices represent all intersections in New York City, and the edges represent all two-way streets that pass through these intersections
   **C)** A graph where the vertices represent all students at the University of Michigan, and the edges represent whether a student shares a class with another student
   **D)** A graph where the vertices represent all students and staff in EECS 281, and the edges represent whether someone has gotten help from someone else during office hours
   **E)** A graph where the vertices represent all EECS students, and the edges represent whether a student has ever collaborated with another student on an EECS assignment

6. Suppose you have a graph with 100 edges and 100 vertices. Is the graph sparse or dense, and should you represent this graph as an adjacency list or an adjacency matrix?
   **A)** This graph is sparse, and an adjacency list should be used
   **B)** This graph is sparse, and an adjacency matrix should be used
   **C)** This graph is dense, and an adjacency list should be used
   **D)** This graph is dense, and an adjacency matrix should be used
   **E)** None of the above

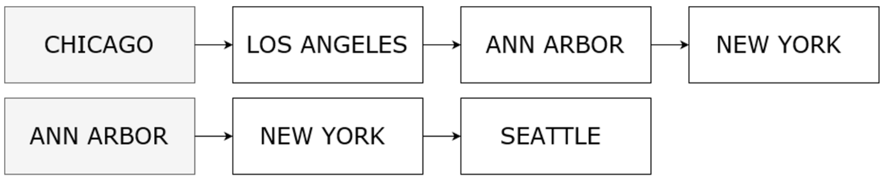
7. Suppose you have a binary tree of height 281, where the leaf nodes have height 1, and you want to search for an element $k$. You know that $k$ exists as a distinct element in this tree, and that it is a leaf node. Which of the following statements is **FALSE**?
   **A)** If you conduct a depth-first search, you'll never have to store more than 281 nodes in memory
   **B)** Conducting a breadth-first search would likely require much more memory than a depth-first search
   **C)** Using a stack to implement this search is preferable to using a queue
   **D)** The path from the root to element $k$ returned by a BFS is shorter than the path returned by a DFS
   **E)** None of the above

*Consider questions 8 and 9 independently.*

8. You are performing a *depth*-first search on a binary tree with depth 281. You are using a deque to conduct this search, and you want to find an element $k$. You do not know where this element $k$ is, but you do know that $k$ exists as a unique value in this tree. Once $k$ is first encountered, the search terminates immediately (without pushing it into the deque). You start the search by pushing the root into the deque. After running the DFS for a while, you notice that an element at a depth of 121 was pushed into your deque. Knowing this information, which of the following statements is *definitely* **FALSE**?
   **A)** The element $k$ can be found at a depth of 120
   **B)** The element $k$ can be found at a depth of 121
   **C)** The element $k$ can be found at a depth of 122
   **D)** More than one of A, B, or C
   **E)** None of the above

9. You are performing a *breadth*-first search on a binary tree with depth 281. You are using a deque to conduct this search, and you want to find an element $k$. You do not know where this element $k$ is, but you do know that $k$ exists as a unique value in this tree. Once $k$ is first encountered, the search terminates immediately (without pushing it into the deque). You start the search by pushing the root into the deque. After running the BFS for a while, you notice that an element at a depth of 121 was pushed into your deque. Knowing this information, which of the following statements is *definitely* **FALSE**?
   **A)** The element $k$ can be found at a depth of 120
   **B)** The element $k$ can be found at a depth of 121
   **C)** The element $k$ can be found at a depth of 122
   **D)** More than one of A, B, or C
   **E)** None of the above

10. Consider the following adjacency list:



   Which of the following statements is **FALSE**?
   **A)** This adjacency list represents a directed graph
   **B)** Ann Arbor has a direct connection with New York
   **C)** New York has a direct connection with Seattle
   **D)** Chicago has a direct connection with New York
   **E)** More than one of the above

11. The partially-filled table below represents the distances between six locations. The graph associated with this table is simple and undirected.

|  | **A** | **B** | **C** | **D** | **E** | **F** |
|---|---|---|---|---|---|---|
| **A** |  | 258 |  |  | 447 | −x− |
| **B** |  |  |  | 611 |  |  |
| **C** | 114 |  |  |  |  | 745 |
| **D** |  | −y− | 331 |  |  |  |
| **E** |  |  |  |  |  | 583 |
| **F** | 252 |  |  |  |  |  |

   What is the value of $x + y$?
   **A)** 583
   **B)** 778
   **C)** 863
   **D)** 997
   **E)** Not enough information is given to answer this question

*For questions 12-13, consider the following scenario.*

You currently have a graph that represents several airports (vertices) and the flights that connect each of them (edges), weighted by the distance of each flight. You decide to implement this graph using an adjacency list. The number of vertices is represented by $V$, and the number of edges is represented by $E$.

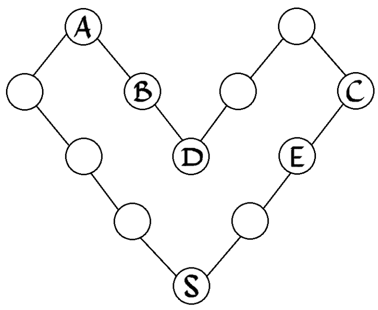
12. Given an airport $X$, what is the *average-case* time complexity of finding the closest airport to airport $X$?
    - **A)** $\Theta(1)$
    - **B)** $\Theta(E)$
    - **C)** $\Theta(V)$
    - **D)** $\Theta(1+E/V)$
    - **E)** $\Theta(V^2)$

13. Given an airport $X$, what is the *worst-case* time complexity of finding if *any* flights depart from airport $X$?
    - **A)** $\Theta(1)$
    - **B)** $\Theta(E)$
    - **C)** $\Theta(V)$
    - **D)** $\Theta(1+E/V)$
    - **E)** $\Theta(V^2)$

14. You are given a graph where vertices represent current students at the University of Michigan and edges represent whether two students have shared a class together. If you want to model this graph in a computer algorithm, which type of graph representation should you prefer, and why?
    - **A)** Adjacency list, because this graph is sparse
    - **B)** Adjacency list, because this graph is dense
    - **C)** Adjacency matrix, because this graph is sparse
    - **D)** Adjacency matrix, because this graph is dense
    - **E)** Both the adjacency list and adjacency matrix are equally preferable

15. You are given a graph that represents a localized network of servers, where each server has a direct connection with every other server in the network. If you want to model this graph in a computer algorithm, which type of graph representation should you prefer, and why?
    - **A)** Adjacency list, because this graph is sparse
    - **B)** Adjacency list, because this graph is dense
    - **C)** Adjacency matrix, because this graph is sparse
    - **D)** Adjacency matrix, because this graph is dense
    - **E)** Both the adjacency list and adjacency matrix are equally preferable

16. You are conducting a breadth-first search on the graph below. The graph is unweighted. If you start your search at vertex $S$, which one of the following vertices would you visit last?



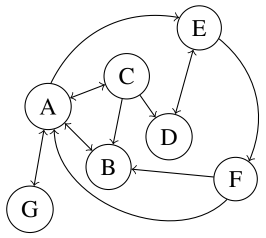
    - **A)** Vertex $A$
    - **B)** Vertex $B$
    - **C)** Vertex $C$
    - **D)** Vertex $D$
    - **E)** Vertex $E$

17. For which of the following graphs would a breadth-first search always find the lowest weighted path between two vertices in the graph?
    - **I.** A tree with weighted edges
    - **II.** A weighted graph where all edges have the same weight
    - **III.** A weighted graph where all paths from the source to destination vertex have edge weights that are in strictly increasing order as the distance from the source vertex increases

    - **A)** I only
    - **B)** II only
    - **C)** I and II only
    - **D)** II and III only
    - **E)** I, II, and III

18. Which of the following statements is **FALSE** regarding a breadth-first search?
    **A)** A breadth-first search will always find the shortest weighted path between two vertices in a weighted graph, but only if that path consists of the fewest possible number of edges
    **B)** A breadth-first search will always find the shortest path between two vertices in an unweighted graph, if a path exists
    **C)** The time complexity of a breadth-first search is $\Theta(|V|+|E|)$ on an adjacency list and $\Theta(|V|^2)$ on an adjacency matrix, where $|V|$ is the number of vertices and $|E|$ is the number of edges
    **D)** The time complexity of a breadth-first search is the same on both a directed graph and an undirected graph, provided that they share the same underlying graph representation
    **E)** None of the above

19. Consider the following graph:



Which of the following set of terms best describes this graph?
    **A)** Weighted, undirected, acyclic
    **B)** Unweighted, undirected, acyclic
    **C)** Unweighted, directed, acyclic
    **D)** Weighted, directed, cyclic
    **E)** Unweighted, directed, cyclic

20. Consider the following adjacency matrix:

| | **A** | **B** | **C** | **D** | **E** | **F** |
|---|---|---|---|---|---|---|
| **A** | 0 | 0 | 1 | 0 | 0 | 1 |
| **B** | 0 | 0 | 0 | 0 | 1 | 0 |
| **C** | 0 | 0 | 0 | 1 | 0 | 0 |
| **D** | 0 | 1 | 1 | 0 | 1 | 0 |
| **E** | 0 | 0 | 0 | 1 | 0 | 1 |
| **F** | 1 | 0 | 0 | 1 | 1 | 0 |

Which of the following search (i.e., discovery) orders are **possible** with a depth-first search?
    **I.** A, C, B, E, D, F
    **II.** A, F, D, E, B, C
    **III.** A, F, E, D, B, C

    **A)** I only
    **B)** II only
    **C)** III only
    **D)** II and III only
    **E)** I, II, and III

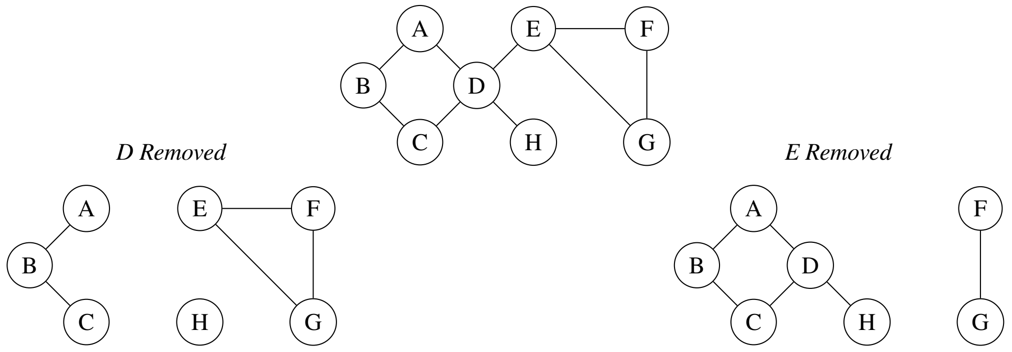
21. Which of the following correctly matches the graph searching algorithm with its corresponding data structure?
    **A)** BFS: Queue DFS: Stack
    **B)** BFS: Stack DFS: Queue
    **C)** BFS: Queue DFS: Queue
    **D)** BFS: Stack DFS: Stack
    **E)** None of the above

22. True or false? Given the same graph, a recursive depth-first search will always visit vertices in the same order as an iterative depth-first search.
    **A)** True
    **B)** False

23. Which of the following techniques can be used to detect whether a cycle exists in an undirected graph?
    **I.** Breadth-first search
    **II.** Depth-first search
    **III.** Union-find

    **A)** I only
    **B)** II only
    **C)** I and II only
    **D)** I and III only
    **E)** I, II, and III

24. An *articulation point* in a graph is a vertex whose removal would increase the number of connected components in the graph. For instance, vertices $D$ and $E$ are articulation points in the following graph, since removing either of these vertices would disconnect the graph.



Consider a brute force approach for finding all articulation points of a graph that individually removes each node and then checks if the number of connected components has increased or not using a depth-first search. If the graph is represented as an adjacency list, and the number of nodes and edges are $|V|$ and $|E|$, respectively, what is the time complexity of this brute force approach?

**A)** $\Theta(|V|)$
**B)** $\Theta(|V|+|E|)$
**C)** $\Theta(|V|^2)$
**D)** $\Theta(|V|^2+|V||E|)$
**E)** $\Theta(|V|^3)$

25. You are given a matrix of 0's and 1's. Implement a function that finds the distance of the nearest 0 for each cell of the matrix. The distance between two adjacent cells is 1.

**Example:** Given the following matrix:

```
matrix = [ [0, 0, 0],
           [0, 1, 0],
           [1, 1, 1] ]
```

you should return the following:

```
result = [ [0, 0, 0],
           [0, 1, 0],
           [1, 2, 1] ]
```

You may assume that there is at least one 0 in the given matrix. Cells are only adjacent in four directions: up, down, left, and right. You solution should run in $O(MN)$ time and $O(MN)$ auxiliary space, where $M$ and $N$ are the dimensions of the matrix.

```
std::vector<std::vector<int32_t>> distance_to_zero(const std::vector<std::vector<int32_t>>& matrix);
```

26. You are given an integer $n$ representing the number of vertices in a directed graph where the vertices are numbered from 0 to $n-1$. Each edge in this graph is colored either red or blue, and there could be self-edges and parallel edges. Given two vectors `red_edges` and `blue_edges`, where `red_edges[i] = [a, b]` indicates that there is a directed red edge from from vertex `a` to vertex `b` (and similar for `blue_edges`), return a vector `result` of length $n$ such that `result[x]` is the length of the shortest path from vertex 0 to vertex `x` such that the edge colors alternate between red and blue along the path, or `-1` if such a path does not exist.

**Example:** Given the following inputs:

```
red_edges  = [[0, 1], [1, 2]]
blue_edges = [[3, 1], [2, 3]]
```

you would return the output `[0, 1, -1, 2]`. This is the because the shortest path with alternating colors has length 0 from vertex 0 to 0 (trivially), length 1 from vertex 0 to 1 ($0 \to 1$), impossible from vertex 0 to 2, and length 2 from vertex 0 to 3 ($0 \to 1 \to 3$).

```
std::vector<int32_t> shortest_color_alternating_path(int32_t n,
    const std::vector<std::vector<int32_t>>& red_edges,
    const std::vector<std::vector<int32_t>>& blue_edges);
```

27. You have a lock in front of you with four circular wheels. Each wheel has ten slots: `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`. The wheels can rotate freely and wrap around (e.g., `'9'` can become `'0'` in a single turn, and vice versa). Each move consists of a turning one wheel one slot. The lock initially starts at `"0000"`, a string representing the state of the four wheels.

You are given a vector of `deadends`, and if the lock displays any of these dead ends, the wheels of the lock will stop turning and you will be unable to open it. Given a `target` representing the value that will unlock the lock, return the minimum number of turns needed to open the lock, or `-1` if it is impossible.

**Example:** Given `deadends = ["0201", "0101", "0102", "1212", "2002"]` and `target = "0202"`, the function would return `6`, since six moves are needed to get to `"0202"` without hitting any of the dead ends: `"0000"` → `"1000"` → `"1100"` → `"1200"` → `"1201"` → `"1202"` → `"0202"`.

You are given the following function, which returns all sequences that can be attained in a single turn from the original sequence `orig_seq`:

```
std::vector<std::string> single_turn_seqs(const std::string& orig_seq) {
  std::vector<std::string> result;
  for (int32_t i = 0; i < 4; ++i) {
    std::string temp = orig_seq;
    temp[i] = (orig_seq[i] - '0' + 1) % 10 + '0';
    result.push_back(temp);
    temp[i] = (orig_seq[i] - '0' - 1 + 10) % 10 + '0';
    result.push_back(temp);
  } // for i
  return result;
} // single_turn_seqs()
```

You may use the `single_turn_seqs()` function in your solution.

```
int32_t min_turns_to_open_lock(const std::vector<std::string>& deadends, const std::string& target);
```

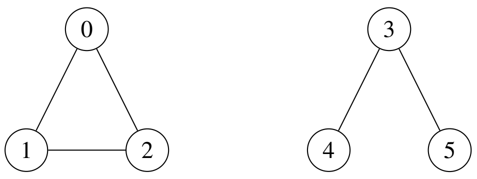
28. You are given a 2D matrix `matrix` of size $m \times n$ in the form of a vector of vectors, where each cell can store a character from `'a'` to `'z'`. Implement a function that returns whether there exists any cycle consisting of the same value in `matrix`. A cycle is a path of length 4 or more in the matrix of the same value that starts and ends at the same cell. From any given cell, you can move to any one of the cells adjacent to it in the four directions of up, down, left, and right. However, you cannot move to the cell that you visited in your last move (e.g., do not consider $(1,1) \to (1,2) \to (1,1)$ as a cycle). For example, you would return `true` if given this matrix, due to the cycle of b's:

| | | | | |
|---|---|---|---|---|
| a | **b** | **b** | **b** | **b** |
| a | **b** | b | a | **b** |
| a | **b** | a | **b** | **b** |
| a | **b** | **b** | **b** | a |
| a | b | a | a | b |

```
bool contains_cycle(const std::vector<std::vector<char>>& matrix);
```

29. You are given an integer $n$ representing the number of vertices in an undirected graph, where each vertex is numbered from 0 to $n-1$. You are also given a vector `edges` where `edges[i] = [a, b]` indicates the existence of an undirected edge from vertex `a` to `b`. Implement a function that returns the number of *complete* connected components in the graph. A connected component is a subgraph within a graph in which there exists a path between any two vertices, and no vertex in the subgraph shares an edge with a vertex outside the subgraph. A complete connected component is one where there exists an edge between every pair of vertices.

**Example:** Given $n = 6$ and `edges = [[0, 1], [0, 2], [1, 2], [3, 4], [3, 5]]`, you would return `1`. This is because there is only one *complete* connected component, consisting of elements 0, 1, and 2. The component containing 3, 4, and 5 is not complete, since there is no connection between 4 and 5, and thus is not counted in our solution.



```
int32_t num_complete_components(int32_t n, const std::vector<std::vector<int32_t>>& edges);
```

## Chapter 19 Exercise Solutions

1. **The correct answer is (C).** The number of edges can be calculated using the following equation:
$$E = \frac{V(V-1)}{2} = \frac{10\times 9}{2} = 45$$

2. **The correct answer is (B).** This statement is not always true: parallel edges may exist that have different weights.The distance required to go to a certain vertex may be different depending on the path you take.

3. **The correct answer is (A).** Adjacency matrices are good for dense graphs, as they provide constant time edge lookup without having to worry about additional overhead that could be incurred while exploring edges that do not exist (which is what an adjacency list is designed to handle, for sparse graphs).

4. **The correct answer is (C).** Friends on Facebook are bidirectional: being friends with someone on Facebook means that the other person must also be friends with you. Thus, a graph representing Facebookd friends is undirected. On the other hand, Twitter followers have a "direction": you can follow someone without forcing them to follow you back. Because instances exist where one person may be connected to another person but not the other way around, this graph is directed.

5. **The correct answer is (D).** If person A got help from person B during office hours, it does not mean that person B also got help from person A. Thus, a graph representing help during office hours would be directed. All the other examples are undirected, where a connection from A to B also implies a connection from B to A.

6. **The correct answer is (A).** The number of vertices is on the order of the number of edges ($E \approx V$), so the graph is sparse, and an adjacency list should be used.

7. **The correct answer is (D).** For a binary tree, there is only one way to get to each node. Because of this, the path returned by a breadth-first search would be the same as the path returned by a depth-first search. The other statements are all true. For option (A), a DFS explores vertices in the graph one branch at a time, so the most nodes that would need to be stored in memory at any point in time is bounded by the length of the longest possible path between two vertices (in this case, 281). For option (B), a BFS visits the tree one level at a time, so all the nodes at a single level may need to be stored in the underlying queue at the same time (which is a lot greater than 281 for the final level, since $k$ is a leaf node). For option (C), a stack is preferable because a DFS is preferable to a BFS, for the reasons discussed above.

8. **The correct answer is (E).** Since a depth-first search looks down each path individually (and does not look at every element in a level before moving down), you cannot conclude anything about where the element $k$ may be.

9. **The correct answer is (A).** Since a breadth-first search explores every element at a level before moving down to the next level, and the search terminates immediately when the target element is discovered (without pushing it into the deque), if you reached a depth of 121, you know that $k$ must not have been found at a depth of 120: otherwise, the search would have terminated before reaching depth 121.

10. **The correct answer is (C).** The adjacency list tells us that (1) Chicago has a direct connection with Los Angeles, Ann Arbor, and New York, and (2) Ann Arbor has a direct connection with New York and Seattle. Because Chicago has a direct connection to Ann Arbor, but Ann Arbor does not have a connection back to Chicago, we know that this graph is directed and that choice (A) is true. The only false statement is choice (C): since New York does not have an adjacency list entry, there are no connections from New York to anywhere in this graph, even though there are direct connections from Chicago and Ann Arbor to New York.

11. **The correct answer is (C).** Because the graph is simple and undirected, you know that the weight from A to B is equal to the weight from B to A. $x$ is equal to the weight from A to F, which is the same as the weight from F to A (252). $y$ is equal to the weight from D to B, which is equal to the weight from $B$ to $D$ (611). Thus, $x + y = 252 + 611 = 863$.

12. **The correct answer is (D).** The average length of a linked list in an adjacency list is $E/V$, where $V$ is the number of airports and $E$ is the total number of connections that exist among these airports. In the average case, the total time complexity is the time required to access the vertex list (a $\Theta(1)$ operation) and all the elements in the list (a $\Theta(E/V)$ operation), which comes out to $\Theta(1 + E/V)$.

13. **The correct answer is (A).** To see if any flight departs from an airport, simply check if there is an element in the list corresponding to that airport. This only takes $\Theta(1)$ time.

14. **The correct answer is (A).** The number of students that any one student has shared a class with at the university is a relatively small constant in comparison with the total number of students at the university (e.g., it's infeasible for a single student to have shared a class with every student at the university), so the graph is sparse, and an adjacency list should be used.

15. **The correct answer is (D).** Since every server has a direct connection with every other server in the network, the graph is complete and therefore dense, and an adjacency matrix should be used.

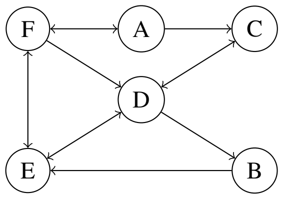
16. **The correct answer is (D).** A breadth-first search encounters vertices in order of distance from the source node. In this case, the path length from $S$ to $D$ is longer than the path length from $S$ to any other vertex, so vertex $D$ would be discovered last during the search.

17. **The correct answer is (C).** A breadth-first search will find a path from source to destination that traverses the fewest number of edges, so if that path also has the lowest weight, then the breadth-first search would have also found the lowest weighted path. This is true for I, since there is only one path between any two vertices in a tree, and also for II, since any path with the fewest edges must also be the lowest weighted path (because all edges have the same weight, more edges = more total weight). However, for III, it is still possible for a shorter path to have a larger weight, which would cause BFS to fail to find the lowest weighted path (e.g., a path with three edges of increasing weight 1, 2, and 3, vs. a path of two edges of increasing weight 4 and 5).

18. **The correct answer is (A).** There can be multiple paths to the target vertex that consist of the same number of edges. BFS will always find one of these paths, but there is no guarantee that this path will be the shortest weighted path (e.g., if there are two paths with the fewest possible number of edges, but with different total weights, you cannot guarantee which path BFS will discover).

19. **The correct answer is (E).** This graph is unweighted since the edges have no weight associated with them, directed since some edges are one-directional (e.g., F goes to B, but B does not go back to F), and cyclic, since there are cycles in the graph (e.g., $A \to E \to F \to A$).

20. **The correct answer is (D).** This is what the graph looks like:



The search order in I is not possible, since there is no way for the search to discover $B$ if only $A$ and $C$ were discovered previously. The search order in II is possible if the DFS first explores the path $A \to F \to D \to E$, and then the path $A \to F \to D \to B$, and then the path $A \to C$. The search order in III is possible if the DFS first explores the path $A \to F \to E \to D \to B$, and then the path $A \to C$.

21. **The correct answer is (A).** A breadth-first processes vertices in first-in first-out (FIFO) order, which uses a queue. A depth-first search processes vertices in last-in first-out (LIFO) order, which uses a stack (or recursion, which uses the call stack).

22. **The correct answer is (B).** There is no guarantee that a recursive and iterative DFS would visit all potential branches in the same order, since that depends on the order in which neighboring vertices are visited. See section 19.3.1 and 19.3.2 for an example where the branches are visited in different orders.

23. **The correct answer is (E).** All three strategies can be used to detect cycles in an undirected graph. With a BFS and DFS, we know a cycle exists if there are multiple ways to access a node from the starting vertex, which we can identify by checking if each vertex encountered during the search had been discovered before (see section 19.5). We can also use a similar strategy with union-find: in such an approach, we would start by having every vertex as its own disjoint set. Then, for each edge from two vertices $i$ and $j$ in the graph, we check if $i$ and $j$ are part of the same disjoint set. If there are not, we union them together. Otherwise, there must be a cycle, since $i$ and $j$ already being in the same disjoint set implies that there must exist some other way to get from $i$ to $j$ that was discovered previously, and that the new edge under consideration would add another path from $i$ to $j$. Note that this union-find approach would only work if the graph were undirected, since we cannot represent a directed graph using this data structure.

24. **The correct answer is (D).** This brute force approach individually removes each of the $n$ nodes and performs a DFS after each removal. Since an adjacency list is used, the time complexity of each DFS is $\Theta(|V|+|E|)$, which is run $|V|$ times. Multiplying these two terms together gives us the time complexity in option (D).

25. Since we want to find the closest distance to the nearest 0 for each cell of the matrix, this can be treated as a shortest-path problem, and we can try using a breadth-first search. One solution is to iterate over the entire matrix, and whenever we encounter a 0, we push it into a queue. Then, we begin a BFS using the queue, and every unvisited cell we discover in the matrix must have a shortest distance that is one larger than that of the neighboring cell we encountered directly before it during our search. An implementation of this is shown below:

```cpp
std::vector<std::vector<int32_t>> distance_to_zero(
    const std::vector<std::vector<int32_t>>& matrix) {
  // we will iterate over this to easily access neighboring cells (right, up, left, down)
  std::vector<std::pair<int32_t, int32_t>> dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
  // queue of coordinates to use for BFS
  std::queue<std::pair<int32_t, int32_t>> bfs;
  // push all existing zeros into the queue, and then start a BFS using these
  // coordinates to discover the closest distance to all other cells
  int32_t m = matrix.size(), n = matrix[0].size();
  std::vector<std::vector<int32_t>> dist(m, std::vector<int32_t>(n, -1));
  for (int32_t i = 0; i < m; ++i) {
    for (int32_t j = 0; j < n; ++j) {
      if (matrix[i][j] == 0) {
        bfs.emplace(i, j);
        dist[i][j] = 0;
      } // if
    } // for j
  } // for i
  while (!bfs.empty()) {
    auto [curr_row, curr_col] = bfs.front();
    bfs.pop();
    for (auto [row_offset, col_offset] : dirs) {
      int32_t next_row = curr_row + row_offset;
      int32_t next_col = curr_col + col_offset;
      if (!(next_row == m || next_col == n || next_row < 0 || next_col < 0) &&
          dist[next_row][next_col] == -1) {
        bfs.emplace(next_row, next_col);
        dist[next_row][next_col] = dist[curr_row][curr_col] + 1;
      } // if
    } // for offset
  } // while
  return dist;
} // distance_to_zero()
```

26. This is a standard shortest-path BFS problem, with the added twist that the path discovered must have alternating colors. This can be handled by performing a normal BFS, but when exploring the unvisited neighbors of a vertex, we only push them into the queue if the edge color differs from the previous edge we traversed to reach the current vertex. One solution is shown below; here, we build an adjacency list that also keeps track of the color of each edge for each vertex. Then, when performing our BFS, we only push a vertex into the queue if its edge color is different from the preceding edge color we encountered.

```cpp
enum class EdgeColor { None, Red, Blue };

std::vector<int32_t> shortest_color_alternating_path(int32_t n,
    const std::vector<std::vector<int32_t>>& red_edges,
    const std::vector<std::vector<int32_t>>& blue_edges) {
  std::vector<int32_t> solution(n, -1);
  std::vector<std::vector<std::pair<int32_t, EdgeColor>>> adj_list(n);
  std::queue<std::pair<int32_t, EdgeColor>> bfs{{{0, EdgeColor::None}}};
  // can use std::unordered_set as well, but will need custom hash function
  std::set<std::pair<int32_t, EdgeColor>> visited;

  // populate adjacency list
  for (const std::vector<int32_t>& edge : red_edges) {
    adj_list[edge[0]].emplace_back(edge[1], EdgeColor::Red);
  } // for edge

  for (const std::vector<int32_t>& edge : blue_edges) {
    adj_list[edge[0]].emplace_back(edge[1], EdgeColor::Blue);
  } // for edge

  int32_t len = 0;
  while (!bfs.empty()) {
    size_t init_queue_size = bfs.size();
    for (size_t i = 0; i < init_queue_size; ++i) {
      auto [curr_vertex, prev_color] = bfs.front();
      bfs.pop();
      if (solution[curr_vertex] == -1) {
        solution[curr_vertex] = len;
      } // if
      for (auto [neighbor, next_color] : adj_list[curr_vertex]) {
        if (visited.find({neighbor, next_color}) != visited.end() || prev_color == next_color) {
          continue;
        } // if
        bfs.emplace(neighbor, next_color);
        visited.emplace(neighbor, next_color);
      } // for neighbor
    } // for i
    ++len;
  } // while

  return solution;
} // shortest_color_alternating_path()
```

27. While it may not seem like it initially, this problem can be converted into a graph problem if we view each state of the lock as a vertex and each possible turn as an edge (e.g., there is an edge from the state "0000" to "0001"). By doing this, the problem essentially becomes "what is the shortest path from "0000" to the target combination?", which can be solved using a breadth-first search. The method for determining which combinations we can attain from any single source combination is already given to use in the `single_turn_seqs()` function, so we can use that to determine the neighbors of any vertex of our graph. One implementation of this solution is shown below:

```cpp
int32_t min_turns_to_open_lock(const std::vector<std::string>& deadends,
                               const std::string& target) {
  std::unordered_set<std::string> deadends_set(deadends.begin(), deadends.end());
  std::unordered_set<std::string> visited;
  std::queue<std::string> bfs;
  std::string init = "0000";
  if (deadends_set.find(init) != deadends_set.end()) {
    return -1;
  } // if
  visited.insert(init);
  bfs.push(init);
  int32_t result = 0;
  while (!bfs.empty()) {
    // same strategy as level order traversal, we keep track of size of the queue at start of loop
    // to identify all vertices that are the same distance away from the source vertex
    size_t curr_layer_size = bfs.size();
    for (size_t i = 0; i < curr_layer_size; ++i) {
      std::string next = bfs.front();
      bfs.pop();
      if (next == target) {
        return result;
      } // if
      std::vector<std::string> possible_seqs = single_turn_seqs(next);
      for (const std::string& seq : possible_seqs) {
        if (visited.find(seq) == visited.end() && deadends_set.find(seq) == deadends_set.end()) {
          bfs.push(seq);
          visited.insert(seq);
        } // if
      } // for seq
    } // for i
    ++result;
  } // while
  return -1;
} // min_turns_to_open_lock()
```

28. This is just a variation of the cycle detection problem, except that the graph is represented in the form of a 2D matrix. We can perform DFS or BFS on the matrix from each cell, and if we encounter a cell we have already visited, we know there must be a cycle. A DFS solution is shown below (note that BFS and union-find are also acceptable, see details in the cycle detection section of the chapter):

```cpp
bool contains_cycle(const std::vector<std::vector<char>>& matrix) {
  int32_t m = matrix.size(), n = matrix[0].size();
  std::vector<std::vector<bool>> visited(m, std::vector<bool>(n, false));
  for (int32_t i = 0; i < m; ++i) {
    for (int32_t j = 0; j < n; ++j) {
      if (!visited[i][j] && is_cyclic(matrix, visited, i, j, -1, -1)) {
        return true;
      } // if
    } // for j
  } // for i
  return false;
} // contains_cycle()

bool is_cyclic(const std::vector<std::vector<char>>& matrix, std::vector<std::vector<bool>>& visited,
               int32_t curr_row, int32_t curr_col, int32_t prev_row, int32_t prev_col) {
  // we will iterate over this to easily access neighboring cells (right, up, left, down)
  static const std::vector<std::pair<int32_t, int32_t>> dirs = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
  visited[curr_row][curr_col] = true;
  for (auto [row_offset, col_offset] : dirs) {
    int32_t next_row = curr_row + row_offset;
    int32_t next_col = curr_col + col_offset;
    bool is_next_valid = next_row != matrix.size() && next_col != matrix[0].size() &&
                         next_row >= 0 && next_col >= 0;
    if (is_next_valid && matrix[curr_row][curr_col] == matrix[next_row][next_col] &&
        !(next_row == prev_row && next_col == prev_col)) {
      if (visited[next_row][next_col] ||
          is_cyclic(matrix, visited, next_row, next_col, curr_row, curr_col)) {
        return true;
      } // if
    } // if
  } // for
  return false;
} // is_cyclic()
```

29. To solve this problem, we can use a DFS (or BFS) to identify the connected components in the graph (similar to example 19.13). While identifying each components, we can count the number of nodes and edges and use the equation $\frac{n(n-1)}{2}$ to determine if the component is complete (see section 19.1 for more details, but a complete graph with $n$ vertices has $\frac{n(n-1)}{2}$ edges). A DFS solution is provided below:

```
int32_t num_complete_components(int32_t n, const std::vector<std::vector<int32_t>>& edges) {
  // convert to adjacency list
  std::vector<std::vector<int32_t>> adj_list(n);
  for (const auto& edge : edges) {
    adj_list[edge[0]].push_back(edge[1]);
    adj_list[edge[1]].push_back(edge[0]);
  } // for edge
  std::vector<bool> visited(n);
  int32_t count = 0;
  for (int32_t i = 0; i < n; ++i) {
    int32_t num_vertices = 0, num_edges = 0;
    if (!visited[i]) {
      dfs(adj_list, i, visited, num_vertices, num_edges);
      if (num_vertices * (num_vertices - 1) == num_edges) {
        ++count;
      } // if
    } // if
  } // for
  return count;
} // num_complete_components()

void dfs(const std::vector<std::vector<int32_t>>& adj_list, int32_t curr_vertex,
         std::vector<bool>& visited, int32_t& num_vertices, int32_t& num_edges) {
  visited[curr_vertex] = true;
  num_vertices += 1;
  num_edges += adj_list[curr_vertex].size();
  for (int32_t neighbor : adj_list[curr_vertex]) {
    if (!visited[neighbor]) {
      dfs(adj_list, neighbor, visited, num_vertices, num_edges);
    } // if
  } // for
} // dfs()
```

Here is an iterative DFS solution (a BFS is similar, just with a queue instead of a stack):

```
int32_t num_complete_components(int32_t n, const std::vector<std::vector<int32_t>>& edges) {
  // convert to adjacency list
  std::vector<std::vector<int32_t>> adj_list(n);
  for (const auto& edge : edges) {
    adj_list[edge[0]].push_back(edge[1]);
    adj_list[edge[1]].push_back(edge[0]);
  } // for edge
  std::vector<bool> visited(n);
  std::stack<int32_t> dfs;
  int32_t count = 0;
  for (int32_t i = 0; i < n; ++i) {
    if (!visited[i]) {
      int32_t num_vertices = 0, num_edges = 0;
      visited[i] = true;
      dfs.push(i);
      while (!dfs.empty()) {
        int32_t curr = dfs.top();
        dfs.pop();
        num_vertices += 1;
        num_edges += adj_list[curr].size();
        for (int32_t neighbor : adj_list[curr]) {
          if (!visited[neighbor]) {
            visited[neighbor] = true;
            dfs.push(neighbor);
          } // if
        } // for neighbor
      } // while
      if (num_vertices * (num_vertices - 1) == num_edges) {
        ++count;
      } // if
    } // if
  } // for
  return count;
} // num_complete_components()
```