1. (i) Draw the BFS tree for the Petersen graph: Start with A and assume that the vertices are ordered: A, B, C, D, E, F, G, H, I, J. Classify all the 15 edges in the BFS-tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six cross edges (curved and blue). An arrow from vertex X to vertex Y for some X,Y ∈ {A, B, . . . , J} indicates that the edge (X, Y) was explored from X to Y. The children of a vertex are ordered from left to right by their discovery times.

(ii) Draw another BFS tree for the Petersen graph: Start with J and assume that the vertices are ordered: J, I, H, G, F, E, D, C, B, A. Classify all the 15 edges in the BFS-tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six cross edges (curved and blue). An arrow from vertex X to vertex Y for some X,Y ∈ {A, B, . . . , J} indicates that the edge (X, Y) was explored from X to Y. The children of a vertex are ordered from left to right by their discovery times.

(iii) The Petersen graph is very symmetric. Support this claim with your trees from the previous two parts.

**Answer:** The structure of any BFS tree is exactly the same regardless of the starting vertex and the order among the other vertices. The root has three children each having two children. The six grandchildren of the root are one of the cycles in the Petersen graph in which each vertex is connected to two of its four non-siblings cousins in the traversal tree. Note that the above two trees would be perfectly identical if the two children of F in the top tree, H and I, would exchange their positions in the illustration.
2. (i) Draw the DFS tree for the Petersen graph: Start with \( A \) and assume that the vertices are ordered: \( A, B, C, D, E, F, G, H, I, J \). Classify all the 15 edges in the DFS-tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six cross edges (curved and blue). An arrow from vertex \( X \) to vertex \( Y \) for some \( X, Y \in \{A, B, \ldots, J\} \) indicates that the edge \((X, Y)\) was explored from \( X \) to \( Y \).

(iii) The Petersen graph is very symmetric. Support this claim with your trees from the previous two parts.

**Answer:** Regardless of the starting vertex and the order among the other vertices, the DFS tree has a path with nine vertices. The tenth vertex is either the child of the ninth and last vertex in the path (creating a Hamiltonian path with ten vertices), or the child of the eights vertex in the path, or the child of the seventh vertex in the path. However, unlike Part (i) and Part (ii), for each structure, there are few different ways to add the six back edges. See below the three possible structures without the back edges.
3. Consider the following weighted Petersen graph:

(a) Below is one of the minimum weighted spanning trees whose total weight is 22. The tree is the output of the Kruskal algorithm that considers the edge \((D, I)\) before the edge \((F, H)\) (both have weight 4).

(b) Below is one of the maximum weighted spanning trees whose total weight is 42. The tree is the output of the modified Kruskal algorithm (from problem 4(a)) that considers the edges \((A, B)\) and \((C, H)\) before the edge \((F, H)\) (all three have weight 4).
4. Let $G$ be a weighted connected graph with $n$ vertices and $m$ edges.

   (a) Design an efficient algorithm that finds a **maximum weighted spanning tree** in $G$. What is the complexity of your algorithm as a function of $n$ and $m$?

   **Answer 1:** Construct a graph $G'$ that is $G$ with different weights. The weight of any edge $(u, v)$ in $G'$ is the negation of its weight in $G$:

   $$w'(u, v) = -w(u, v)$$

   Run any minimum spanning tree algorithm on $G'$. The output is a maximum spanning tree in $G$.

   The complexity of generating $G'$ is $\Theta(m)$ if $G$ is represented by adjacency lists. Therefore, the complexity of both Kruskal and Prim algorithms remain the same.

   **Answer 2:** Update the Kruskal algorithm by sorting the edges from the heaviest to the lightest and update the Prim algorithm by maintaining the priority queue in a way that gives higher weights more priority. Run the modified versions on $G$ to output a maximum weighted spanning tree.

   The complexity of both modified versions of the Kruskal and Prim algorithms remain the same.

   **Remark:** The outputs of the Kruskal algorithm and the Prim algorithm on $G'$ are the same as the outputs of their modified versions on $G$.

   (b) Design an efficient algorithm that finds a **minimum** weighted set of edges that intersect all the cycles of $G$. That is, for any cycle of $G$, at least one of its edges belongs to this set.

   What is the complexity of your algorithm as a function of $n$ and $m$?

   **Answer:** Let $S$ be a set of edges that intersects all the cycles of $G$. Remove these edges from $G$. By definition, the remaining graph has no cycles. Furthermore, the remaining graph is connected because $G$ is connected and there is no reason to add to $S$ an edge that disconnects $G$. Therefore, the remaining graph is a spanning tree of $G$.

   The above arguments imply that finding a minimum weighted set $S$ of edges that intersects all the cycles of $G$ is equivalent to first finding a maximum spanning tree $T$ in $G$ and then define $S$ as the set of all the edges in $G$ that are not in $T$.

   If the graph is represented by adjacency lists, then during the run of the maximum weighted spanning tree algorithm (Kruskal or Prim), finding all the edges that are not in the spanning tree can be done with an extra complexity of $\Theta(m)$ time. Therefore, the overall complexity of this algorithm is the same as the complexity of either the Kruskal algorithm or the Prim algorithm.
**Example:** Consider the following weighted Petersen graph.

Below is the maximum weighted spanning tree from problem 3(b) whose total weight is 42.

Below is a set $S$ of edges that intersect all the cycles of the above weighted Petersen graph whose weight is 12 which is the optimal weight for such a set of edges.
5. The cycle graph $C_n$ is a graph that has one cycle containing all $n$ vertices. Describe an efficient algorithm that finds a minimum weighted spanning tree in $C_n$. The algorithm should be more efficient than the algorithms for general graphs. What is the complexity of your algorithm as a function of $n$?

**Algorithm:** Find the heaviest edge in the cycle and omit it. What remains is a path (which is a tree) with $n$ vertices that spans $C_n$.

**Correctness:** This algorithm is another way to implement the Kruskal algorithm. In the Kruskal algorithm, the edges are considered from the lightest to the heaviest one at a time. An edge is added to the spanning tree as long as it does not close a cycle with previously selected edges. The only cycle in $C_n$ is the whole graph. Therefore, only the last considered edge would close a cycle and would not be part of the spanning tree. This last edge is the heaviest edge.

**Complexity:** When $C_n$ is represented by adjacency lists, the length of each list is exactly 2. Hence, collecting the $n$ edges can be done with $\Theta(n)$-complexity. Next, the complexity of finding the heaviest edge is also $\Theta(n)$. Finally, omitting this edge from the cycle has complexity $\Theta(1)$. As a result, the overall complexity of the algorithm is $\Theta(n)$.

When the graph is represented by an adjacency matrix, the algorithm must examine $\Omega(n^2)$ entries of the matrix to collect the $n$ edges of the cycle and determine the identity of the heaviest edge. Since the algorithm examines at most $O(n^2)$ entries in the matrix to collect the $n$ edges, it follows that the overall complexity of the algorithm when the cycle is represented by an adjacency matrix is $\Theta(n^2)$.
6. Let $G$ be a directed graph that is represented by the adjacency matrix $A$.

- In directed graphs: $A(u, v) = 1$ if the edge $(u \to v)$ exists and $A(u, v) = 0$ otherwise.
- A tournament is a directed graph such that for any pair of vertices $u \neq v$, exactly one of the edges $(u \to v)$ or $(v \to u)$ exists.
- An acyclic tournament is a tournament with no directed cycles.
- A source is a vertex whose in-degree is zero. A destination is a vertex whose out-degree is zero.

(a) Describe an algorithm that checks if a directed graph $G$ is a tournament. What is the complexity of your algorithm?

**Algorithm:** For any pair of indices $1 \leq i < j \leq n$, check if $A(i, j) + A(j, i) = 1$.

Conclude that $G$ is a tournament if and only if the answer is “TRUE” for all pairs of vertices.

**Correctness:** Since both $A(i, j)$ and $A(j, i)$ could be either 1 or 0, it follows that their sum is either 0, or 1, or 2. If the sum is 1 then exactly one of them is 1, if the sum is 0 then both are 0, and if the sum is 2 then both are 1. By definition, in a tournament all of these sums must be 1.

**Complexity:** For a graph with $n$ vertices there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices. Therefore, the complexity of the algorithm is $\Theta(n^2)$.

(b) Prove that in an acyclic tournament there is exactly one source and one destination.

**Proof:** First prove that in any tournament there is at most one source and at most one destination. Then prove that in an acyclic tournament there is at least one source and at least one source. The proof is implied by both claims.

(i) Consider first any tournament that is not necessarily acyclic. Assume to the contrary that there are at least two sources. Let $u$ and $v$ be two of these sources. If the edge $u \to v$ exists then $v$ is not a source and if the edge $v \to u$ exists then $u$ is not a source. Since in a tournament only one of these edges exists, it follows that it cannot be the case that both $u$ and $v$ are sources. A contradiction. Similar arguments show that there cannot be more than one destination.

(ii) Consider now an acyclic tournament. Assume to the contrary that there are no sources in the tournament. Construct the following path that includes all $n$ vertices starting with an arbitrary vertex $v_0$.

Since $v_0$ is not a source, there exists another vertex, denoted by $v_1$, such that $(v_1 \to v_0)$ is an edge. Let $P_1 = (v_1 \to v_0)$ be a path with two vertices.

Assume by induction that for $0 \leq i \leq n - 2$, the path $P_i = (v_i \to v_{i-1} \to \cdots \to v_0)$ that includes $i + 1$ distinct vertices has been constructed after $i$ rounds. Show how to construct in round $i + 1$ the path $P_{i+1}$ that includes $i + 2$ distinct vertices. Since $v_i$ is not a source, there exists a vertex $v$ such that the edge $(v \to v_i)$ exists. Note that $v$ cannot be $v_{i-1}$ because the graph is a tournament. Moreover, $v$ cannot be $v_j$ for some $0 \leq j < i - 1$, since this would form the cycle $(v_j \to v_i \to \cdots v_{j+1} \to v_j)$ which cannot exist in an acyclic tournament. Therefore, $v_{i+1} \notin P_i$. Define $v_{i+1} = v$ and let $P_{i+1} = (v_{i+1} \to v_i \to \cdots \to v_0)$.

After $n - 1$ rounds, this procedure constructs the path $P_{n-1} = (v_{n-1} \to \cdots v_1 \to v_0)$ that includes all the $n$ vertices.

Finally, since $v_{n-1}$ is not a source, it follows that there exists an edge $(v_j \to v_{n-1})$ for some $0 \leq j < n - 1$. This edge would form the cycle $(v_j \to v_{n-1} \to \cdots v_{j+1} \to v_j)$. This is a contradiction because the tournament is acyclic.

Similar arguments show that an acyclic tournament has at least one destination.
(c) Describe an efficient algorithm \((O(n)\)-time) to find the source vertex of an acyclic tournament.

**Algorithm:** If the tournament has only one vertex, then trivially this vertex is a source (and also a destination). Assume that the tournament has \(n \geq 2\) vertices. Denote the \(n\) vertices by \(v_0, v_1, \ldots, v_{n-1}\).

- Let \(s_0 = v_0\) be the first vertex to be the candidate source.
- If \((v_1 \rightarrow v_0)\) is an edge, then \(s_1 = v_1\) is the new candidate source and \(v_0\) cannot be the source. Otherwise, \(s_1 = s_0 = v_0\) remains the candidate source and \(v_1\) cannot be the source.
- Assume by induction that for \(1 \leq i < n - 1\) and \(0 \leq j \leq i\), after \(i\) rounds, \(s_i = v_j\) is the candidate source and all the other vertices among \(v_0, v_1, \ldots, v_i\) cannot be sources. Show how to define the candidate source \(s_{i+1}\) among the vertices \(v_0, v_1, \ldots, v_{i+1}\).
- If \((v_{i+1} \rightarrow s_i)\) is an edge, then \(s_{i+1} = v_{i+1}\) is the new candidate source and \(s_i\) cannot be the source. Otherwise, \(s_{i+1} = s_i\) remains the candidate source and \(v_{i+1}\) cannot be the source.
- After \(n - 1\) rounds, \(s_{n-1}\) is the candidate source while the other \(n - 1\) vertices cannot be sources.
- Since an acyclic tournament must have a source, it follows that \(s_{n-1}\) is the source of the tournament.

**Remark:** A similar algorithm can find the destination of the tournament.

**Complexity:** The complexity of each round is \(\Theta(1)\) and therefore the complexity of the algorithm is \(\Theta(n)\).

**Arbitrary tournaments:** Tournaments with cycles can still have a source but some do not. For example, tournaments with a directed Hamiltonian cycle do not have a source. For arbitrary tournaments, after defining \(s_{n-1}\), the algorithm checks if \((s_{n-1} \rightarrow v_i)\) is an edge for all \(0 \leq i \leq n - 1\) such that \(s_{n-1} \neq v_i\). If the answer is always “TRUE,” then \(s_{n-1}\) is the source of the tournament. Otherwise, since the algorithm has already verified that no other vertex could be a source, this arbitrary tournament does not have a source.

**Challenge:** Any tournament has a Hamiltonian path. Prove this by designing an efficient algorithm that outputs a Hamiltonian path for a given tournament.
7. Let $G$ be a graph with $n$ vertices and $m$ edges. Describe a greedy algorithm that finds a set $S$ of vertices with the following two properties:

- $S$ is independent: There is no edge between any pair of vertices of $S$.
- $S$ is dominating: Every vertex that is not in $S$ has at least one neighbor in $S$.

What is the complexity of your algorithm if the graph is represented by adjacency lists and what is the complexity of your algorithm if the graph is represented by an adjacency matrix?

**Algorithm:**

- Initially $S$ is an empty set. Repeat until $G$ has no vertices:
  - Add an arbitrary vertex $v$ from $G$ to the set $S$.
  - Omit from $G$ the vertex $v$, all of its neighbors, and all the edges that include the omitted vertices.

**Correctness:**

- $S$ is an independent set: During the run of the algorithm, when a vertex $v$ joins the set $S$, it must be the case that it is not a neighbor of a previously added vertex to $S$ because these neighbors were omitted from $G$ before $v$ is added to $S$. Moreover, none of $v$’s neighbors that are in $G$ at the time $v$ joins $S$ will be added to $S$ later because they are omitted from $G$ immediately after $v$ joins $S$. Therefore, $v$ has no neighbors in the output set $S$. Since this is true for all the vertices in $S$, it follows that $S$ is an independent set.
- $S$ is a dominating set: Every vertex that is not in $S$ is omitted from $G$ because it is a neighbor of at least one of the vertices in $S$. Otherwise, such a vertex would have been selected to be added to $S$. Therefore, $S$ is a dominating set in $G$.

**Implementation details:** Denote the $n$ vertices by $v_0, v_1, \ldots, v_{n-1}$.

- Maintain an array $B$ of size $n$. For $0 \leq i \leq n-1$, during the run of the algorithm, if $B[i] = S$ then $v_i$ belongs to $S$, if $B[i] = R$ then $v_i$ was omitted from $G$, and if $B[i] = V$ then $v_i$ is in $G$.
- Initially, $B[i] = V$ for all $0 \leq i \leq n-1$ and let $p = -1$.
- Repeat the following procedure until $B[i] = R$ for all $p < r \leq n-1$:
  - Find the first index $i > p$ in $B$ for which $B[i] = V$.
  - Set $B[i] = S$ and $p = i$.
  - For each neighbor $v_j$ of $v_i$, set $B[j] = R$. Note that it could be the case that already $B[j] = R$, but it cannot be the case that $B[j] = S$.

**Complexity:** When $G$ is represented by adjacency lists, scanning all the neighbors can be done with complexity $\Theta(n + m)$ and when $G$ is represented by an adjacency matrix, scanning all the neighbors can be done with complexity $\Theta(n^2)$. The rest of the complexity for both data structures is $\Theta(n)$.

**Observations about the output set $S$ for some families of graphs:**

- Complete graphs: $S$ contains only one vertex that dominates the rest of the vertices.
- Null graph: $S$ is an independent set that contains all the vertices.
- Star graphs: If $v_0$ is the root of the star, then $S$ contains only $v_0$ that dominates its $n-1$ neighbors. Otherwise, $S$ is the independent set that contains all the $n-1$ leaves each dominates the root.
- Path graphs: Depending on the initial order of the vertices, the size of $S$ can get any value in the range $[n/3], \ldots, [n/2]$.
- Cycle graphs: Depending on the initial order of the vertices, the size of $S$ can get any value in the range $[n/3] \ldots [n/2]$. 

The Petersen graph: Assume the initial the order among the vertices is the alphabetical order \( v_0 = A, v_1 = B, \ldots, v_9 = J \).

In the first round of the algorithm, \( A \) is added to \( S \) and the vertices \( A, B, E, F \) are omitted from the graph. The remaining graph is a cycle with six vertices. Due to symmetry, for any choice of \( v_0 \), the remaining graph is always a cycle with six vertices.

In the second round of the algorithm, \( C \) is added to \( S \) and the vertices \( C, D, H \) are omitted from the graph. The remaining graph is always a path with three vertices. Due to symmetry, regardless of the initial ordr among the vertices, the remaining graph is always a path with three vertices.

In the third and last round of the algorithm \( G \) is added to \( S \) and the vertices \( G, I, J \) are omitted from the graph. After this round, \( G \) has no more vertices and the algorithm terminates with the set \( S = \{ A, C, G \} \).

However, if instead of \( G \) the algorithm adds \( I \) (respectively, \( J \)) to \( S \), then in the fourth and last round \( J \) (respectively, \( I \)) is added to \( S \) and the algorithm terminates with the set \( S = \{ A, C, I, J \} \).

The unlabeled Petersen graph: The algorithm can output only two types of sets,

- A set containing the three neighbors of a particular vertex.
- A set containing the four neighbors of a particular edge.

In the above example, \( S = \{ A, C, G \} \) contains all the neighbors of \( B \) and \( S = \{ A, C, I, J \} \) contains all the neighbors of either the edge \((B, G)\) or the edge \((D, E)\) or the edge \((F, H)\).

The labeled Petersen graph: There are ten possible outputs for the first option in which \( S \) contains three vertices since the Petersen graph has ten vertices. Thee are five possible outputs for the second option in which \( S \) contains four vertices since the Petersen graph has fifteen edges but the four vertices of a particular set are neighbors of three different edges.
8. A leaf in a graph $G$ (not necessarily a tree) is a vertex that has exactly one neighbor. Let $G$ be a graph with $n$ vertices and $m$ edges. Describe an efficient algorithm that counts the number of leaves in $G$.

What is the complexity of your algorithm if the graph is represented by adjacency lists and what is the complexity of your algorithm if the graph is represented by an adjacency matrix?

**Algorithm:** For each vertex check if its degree is exactly 1. Count the number of such vertices.

**Complexity:**

- $G$ is represented by adjacency lists: For each vertex, it is enough to examine at most two of its neighbors to determine if its degree is 1. Therefore, it takes $\Theta(1)$ time to verify if a vertex is a leaf. Hence, the complexity of this algorithm is $\Theta(n)$.

- $G$ is represented by an adjacency matrix: For each vertex, in the worst case, its entire row in the matrix must be examined to see if it contains only one entry that equals 1. Therefore, the complexity for each vertex is $\Theta(n)$ and for all the vertices the complexity is $\Theta(n^2)$.

**Remark:** To show that any algorithm must have complexity $\Omega(n^2)$ when the graph is represented by an adjacency matrix consider the following instance. Assume that it is given that in the graph each vertex is either a leaf or an isolated vertex with no neighbors. As a result, in the worst case, $n(n-1)/2 = \Omega(n^2)$ entries in the matrix must be examined to distinguish between the cases of a null graph and a graph whose single edge is between the last two examined vertices.