```
# Class that represents a directed graph.
class Graph:
  # Representation of the graph:
  # * The graph is represented by 3 dictionaries,
  #   one that stores the inbound edges for each vertex, one that stores
  #   the outbound edges for each vertex, and one that stores the cost
  #   associated to each edge.
  # * The existence of a vertex can be checked by simply checking if it
  #   exists as a key in either the `__inbound` or `__outbound` dictionaries.
  # * The existence of an edge can be checked by simply checking if it exists
  #   as a key in the `__cost` dictionary.
  def __init__(self) -> None:
    self.__inbound: Dict[int, List[int]] = {}
    self.__outbound: Dict[int, List[int]] = {}
    self.__cost: Dict[Tuple[int, int], int] = {}

  # Returns the number of vertices in the graph.
  def vertexCount(self) -> int: pass

  # Checks whether the graph contains the specified vertex.
  # Returns: `True` if the vertex exists in the graph, otherwise `False`.
  def isVertex(self, vertex: int) -> bool: pass

  # Returns a list containing all vertices in the graph, in ascending order.
  def vertices(self) -> List[int]: pass

  # Returns a list containing all the inbound edges of the specified vertex.
  # Raises: Exception if the specified vertex is not in the graph.
  # Law: forall v. forall v2 in inbound(v). existsEdge(v2, v)
  def inbound(self, vertex: int) -> List[int]: pass

  # Returns a list containing all the outbound edges of the specified vertex.
  # Raises: Exception if the specified vertex is not in the graph.
  # Law: forall v. forall v2 in outbound(v). existsEdge(v, v2)
  def outbound(self, vertex: int) -> List[int]: pass

  # Checks whether there exists an edge between the 2 vertices.
  # Returns: `True` if there exists an edge between the 2 vertices,
  # otherwise `False`.
  # Raises: Exception if either of the vertices is not in the graph.
  def existsEdge(self, vertex1: int, vertex2: int) -> bool: pass

  # Returns the in degree of the specified vertex.
  # Raises: Exception if the specified vertex is not in the graph.
  def inDegree(self, vertex: int) -> int: pass
```

```
# Returns the out degree of the specified vertex.
# Raises: Exception if the specified vertex is not in the graph.
def outDegree(self, vertex: int) -> int: pass

# Returns the cost associated to the edge between `vertex1` and `vertex2`.
# Raises:
#   * Exception if either vertex is not in the graph.
#   * Exception if there is no edge between vertex1 and vertex2.
def getCost(self, vertex1: int, vertex2: int) -> int: pass

# Sets the cost associated to the edge between `vertex1` and `vertex2` to
# be equal to `cost`.
# Raises:
#   * Exception if either vertex is not in the graph.
#   * Exception if there is no edge between vertex1 and vertex2.
def setCost(self, vertex1: int, vertex2: int, cost: int) -> None: pass

# Adds a new edge between `vertex1` and `vertex2`, with the cost equal
# to `cost`.
# Raises:
#   * Exception if either vertex is not in the graph.
#   * Exception if there already exists an edge between
#     vertex1 and vertex2.
def addEdge(self, vertex1: int, vertex2: int, cost: int) -> None: pass

# Removes the edge between `vertex1` and `vertex2`.
# Raises:
#   * Exception if either vertex is not in the graph.
#   * Exception if there is no edge between vertex1 and vertex2.
def removeEdge(self, vertex1: int, vertex2: int) -> None: pass

# Adds the specified vertex to the graph.
# Raises: Exception if the vertex already exists.
def addVertex(self, vertex: int) -> None: pass

# Removes the specified vertex from the graph.
# Raises: Exception if the vertex is not in the graph.
def removeVertex(self, vertex: int) -> None: pass

# Returns a string representation of the graph.
# Law: forall g. fromString(str(g)) == g
def __str__(self) -> str: pass

# Returns an independent copy of the graph.
def copy(self) -> 'Graph': pass
```

```python
# Constructs a graph from the given string.
@staticmethod
def fromString(s: str) -> 'Graph': pass

# Constructs a graph from the given string, which is expected
# to be in the "old" format (that assumes the graph contains all
# vertices from 0 to n - 1).
# Raises: Exception if edge count > vertex count ^ 2.
@staticmethod
def fromStringOld(s: str) -> 'Graph': pass

# Creates a graph with `vertexCount` vertices numbered from 0
# to `vertexCount` - 1, and `edgeCount` randomly generated edges.
# Raises: Exception if `edgeCount` > `vertexCount` ** 2
@staticmethod
def randomGraph(vertexCount: int, edgeCount: int) -> 'Graph': pass
```