## Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.LinkedList` for a Linked List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered non-efficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

## Graph Algorithms

For this assignment, you will be coding 5 different graph algorithms. WARNING: This homework has quite a few files in it, so you should make sure to read ALL of the documentation given to you, including this pdf as well as all of the javadocs before asking a question.

### Graph Representations

For this assignment, you will be using two different representations of graphs, the adjacency list and adjacency matrix. The adjacency list will be used for BFS, Dijkstra's, Prim's and Kruskal's. The adjacency matrix will be used for DFS.

Generally, an adjacency matrix is better for when the graph is dense (many edges), and an adjacency list is better for when the graph is sparse (fewer edges). Also, it's worth noting from here on out, the time complexities will have two inputs, $|V|$ to mean the number of vertices, and $|E|$ to mean the number of edges. Although it is true that a simple graph can have at most $|E| \leq \frac{|V|(|V|-1)}{2} = O(|V|^2)$ edges, other than this, there isn't really any relationship between $|V|$ and $|E|$.

Keep in mind that the way that the graph is stored/represented will affect the time complexity of the algorithm. For example, for Dijkstra's algorithm, if an adjacency matrix is used, the time complexity is $O(|V|^2)$ while if an adjacency list with a priority queue is used, it is instead $O(|E| + |V| \log |V|)$.

### Search Algorithms

The first two algorithms are breadth-first search (BFS) and depth-first search (DFS). These algorithms are search algorithms that start at the given vertex and traverse the entire graph in a particular order (the order in which the vertices are visited depend on whether you are doing breadth-first search or depth-first search, the edges that are in the graph, and the order the adjacent vertices are listed).

Breadth-First Search is a search algorithm that visits the vertices closest to the starting vertex before visiting further away ones. It relies on a Queue based data structure (think of it as a more general version of level order traversal from BSTs). You will be using an adjacency list for your implementation of BFS.

Depth-First Search is a search algorithm that visits all the vertices in a certain branch before moving onto another branch. DFS depends on a Stack based data structure to work. Your implementation of DFS **MUST** be recursive to receive credit. You will be using an adjacency matrix for your implementation of DFS.

One important property of both BFS and DFS is that you can figure out if the graph is connected or not by running either of the algorithms. This can be done by seeing if all of the vertices have been visited on termination of the algorithm. For your implementation, if the graph is disconnected, the list of vertices will not contain all of the vertices in the graph.

You will be modifying the passed in list, adding the vertices in the order that you visit them, and you will return whether the graph is connected or not. The DFS **MUST** be recursive to receive credit.

## Single-Source Shortest Path (Dijkstra's Algorithm)

The next algorithm is Dijkstra's algorithm. This algorithm finds the shortest path from one vertex to all of the other vertices in the graph. This algorithm only works for non-negative edge weights, so you will have to account for this in your algorithm in accordance with the javadocs. This algorithm will be implemented using an adjacency list along with a priority queue.

## Minimal Spanning Trees (MST)

The last two algorithms are Prim's and Kruskal's algorithms to find the minimal spanning tree (MST) of the graph. An MST is a tree that uses only edges from the original graph that connects all of the vertices in the graph.

Prim's Algorithm begins at an arbitrary vertex and visitsan adjacent vertex using the cheapest edge from that vertex. This algorithm will naturally partition the vertices into those that have been visited and those that have been not, and the algorithm will continually add the edge that adds a new vertex with the cheapest cost to the visited portion.

Kruskal's Algorithm takes in all of the edges of the graph and continually adds the cheapest to the MST as long as that edge does not form a cycle. This means that as the algorithm progresses, there will be multiple disconnected components that will eventually come together if the original graph is connected. To handle cycle detection and these disconnected components, you will be using a disjoint-set/union-find data structure that we have provided for you.

If the original graph is disconnected, these algorithms should instead return a minimal spanning forest (MSF), described in the javadocs. You may assume that the passed in graph has a **unique** MST/MSF and that the graph is undirected, so the starting vertex for Prim's algorithm does not matter. You will be implementing both of these algorithms using an adjacency list.

## Clarifications

For all algorithms, you may assume that you are allowed to go from a vertex to itself, and that the distance of that is 0. In addition, the graph will be either directed or undirected; it will not be a mix of the two, though this should not have too much of an effect, if at all, on your implementation if done correctly. Some more restrictions are described in the javadocs. More information on these algorithms can be found in Chapter 14 of the class textbook.

## A note on JUnits

We have provided a **very basic** set of tests for your code, in `GraphAlgsStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor does it guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza (when it comes up).

If you need help on running JUnits, there is a guide, available on T-Square under Resources, to help you run JUnits on the command line or in IntelliJ.

## Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located in T-Square, under Resources, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Joonho Kim (jkim844@gatech.edu) with the subject header of "CheckStyle XML".

### Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs (remember to keep helper methods private). If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs.

### Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. "Error", "BAD THING HAPPENED", and "fail" are not good messages. The name of the exception itself is not a good message.

For example:

```
throw new PDFReadException("Did not read PDF, will lose points.");
```

```
throw new IllegalArgumentException("Cannot insert null data into data structure.");
```

### Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedList<Integer>()` instead of `new LinkedList()`. Using the raw type of the class will result in a penalty.

## Forbidden Statements

You may not use these in your code at any time in CS 1332. If you use these, we will take off points.

- `break` may only be used in switch-case statements
- `continue`
- `package`
- `System.arraycopy()`
- `clone()`

- `assert()`
- `Arrays` class
- `Array` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes

Debug print statements are fine, but nothing should be printed when we run them. We expect clean runs - printing to the console when we're grading will result in a penalty.

## Provided

The following file(s) have been provided to you. There are several, but you will only edit one of them.

1. `GraphAlgs.java`

   This is the class in which you will implement the different graph algorithms. Feel free to add private static helper methods but **do not add any new public methods, new classes, instance variables, or static variables**.

2. `GraphAlgsStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `GraphAlgs` class. It is not intended to be exhaustive and does not guarantee any type of grade. Write your own tests to ensure you cover all edge cases.

3. `GraphAdjList.java`

   This class represents a graph represented using an adjacency list. It contains the adjacency list, a set of edges, and information about whether the graph is directed or not. It will be used for BFS, Dijkstra's, Prim's, and Kruskal's algorithms. **Do not alter this file.**

4. `Vertex.java`

   This class represents a vertex in the graph. It contains the data in this vertex. It is used in conjunction with `GraphAdjList`. **Do not modify this file**.

5. `Edge.java`

   This class represents an edge in the graph. It contains the vertices connected to this edge, its weight, and whether or not it is directed. It is used in conjunction with `GraphAdjList`. **Do not modify this file**.

6. `VertexDistancePair.java`

   This class holds together a vertex and a distance. It is used in conjunction with `GraphAdjList`. **Do not modify this file**.

7. `DisjointSet.java`

   This class represents a union-find data structure to be used for Kruskal's algorithm, consisting of the operations find and union. **Do not modify this file**.

8. `DisjointSetNode.java`

   This class represents a node for `DisjointSet.java`. **Do not modify this file**.

9. `GraphAdjMatrix.java`

   This class represents a graph represented using an adjacency matrix. Unlike `GraphAdjList`, this class is a standalone class, and it will be used for DFS. **Do not alter this file.**

## Deliverables

You must submit all of the following file(s). Please make sure the filename matches the filename(s) below. Be sure you receive the confirmation email from T-Square, and then download your uploaded files to a new folder, copy over the interfaces, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `GraphAlgs.java`