1. Suppose we are talking about the depth-first search (DFS) algorithm. Nodes are added to the data structures in alphabetical order.

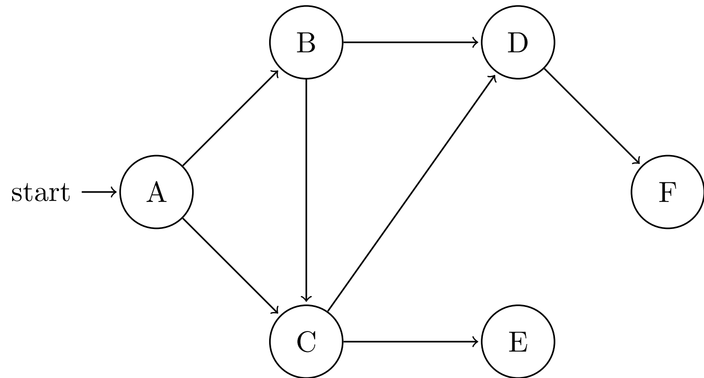
   (a) What underlying data structure does this algorithm use?

   A stack.

   (b) Given the following graph, state the DFS traversal order and show the data structure at each step. Node `A` is the start node, and `F` is the destination node.

   ← bottom of stack
   |A
   |B C
   |B D E
   |B D
   |B F
   |B

   The traversal order is ACEDF.



   (c) What path from `A` to `F` does the DFS algorithm return?

   ACDF

2. Now consider a BFS algorithm, again populating data structures in alphabetical order.

   (a) What changes would need to be made to a DFS implementation to turn it into a breadth-first search (BFS)?

   Use a queue data structure (instead of a stack).

   (b) Using the graph as described in Question 1, what is the BFS traversal order? Show the data structure at each step.

   ← front of queue
   A
   B C
   C D
   D E
   E F
   F

   The traversal order is ABCDEF.

   (c) What path from `A` to `F` results from the BFS algorithm?

   ABDF

3. **Searching a Graph**

   (a) Write a recursive algorithm that (given a graph, start vertex, and goal vertex), determines whether or not there is a path to the goal vertex.

   Assume you are provided with a `Graph` class with a `getNeighbors( int vertex )` method, which returns a `Set<Integer>` representing the numbers corresponding to neighboring vertices. Assume `visited` is a `Set` keeping track of all visited vertices.
   (Note: Your algorithm should return a Boolean value, not an actual path!)

```
boolean hasPathToRec(Graph g, int start, int goal, Set<Integer> visited) {

    if( start == goal ){
        return true;
    } else {
        for( int n : g.getNeighbors(start) ){
            if( ! visited.contains(n) ){
                visited.add(n);
                if ( hasPathToRec(g, n, goal , visited) )
                    return true;
            }
        }
        return false;
    }

}
```

   (b) Rewrite your algorithm to be iterative instead.
   (Hint: What data structure do you need to use if you no longer have recursion?)

```
boolean hasPathToIter(Graph g, int start, int goal, Set<Integer> visited) {

    Stack<Integer> theStack = new Stack<Integer>();
    theStack.push(start);
    visited.add(start);
    while( ! theStack.empty() ){
        int curr = theStack.pop();
        if( curr == goal ){
            return true;
        }
        for( int n : g.getNeighbors(curr) ){
            if( ! visited.contains(n) ) {
                visited.add(n);
                theStack.push(n);
            }
        }
    }
    return false;

}
```

4. When is a vertex's sum weight finalized in Dijkstra's algorithm?

   A vertex's sum weight is final after it has updated the tentative distances of all of its neighbors. Formally, a vertex is considered "finalized" when it is removed from the priority queue.

5. In Dijkstra's algorithm, what role does the priority queue play in finding the shortest path? When do we use it?

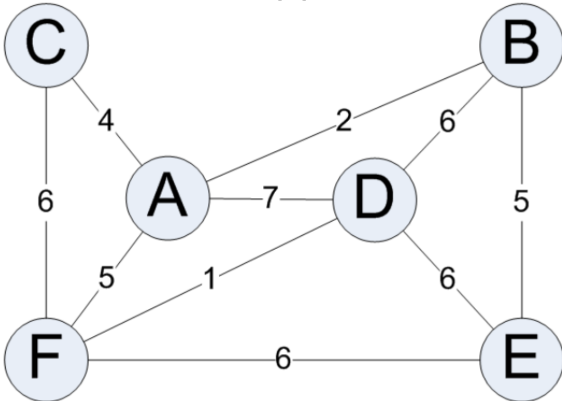
   The priority queue is used to select the next vertex to visit. We want to visit the node which currently has the lowest tentative distance from the start, so we use a priority queue that always returns the lowest element.

6. Why does Dijkstra's algorithm not work correctly on graphs with negative edge weights?

Dijkstra's algorithm doesn't work correctly on graphs with negative edge weights due to one of its *greedy* behaviors.

When selecting the next node to visit, Dijkstra's algorithm chooses the node with the current lowest total value, then *finalizes* that value forever. If there were another route to that node, which had not yet been explored (and contained a net negative weight) Dijkstra's algorithm would return a suboptimal path.

7. Consider the following graph.



(a) Perform Dijkstra's algorithm to find the shortest path between C and E.

Finalized	A	B	C	D	E	F
–	(∞, None)	(∞, None)	**(0, None)**	(∞, None)	(∞, None)	(∞, None)
C	**(4, C)**	(∞, None)	(0, None)	(∞, None)	(∞, None)	**(6, C)**
A	(4, C)	**(6, A)**	(0, None)	**(11, A)**	(∞, None)	(6, C)
F	(4, C)	(6, A)	(0, None)	**(7, F)**	**(12, F)**	(6, C)
B	(4, C)	(6, A)	(0, None)	(7, F)	**(11, B)**	(6, C)
D	(4, C)	(6, A)	(0, None)	(7, F)	(11, B)	(6, C)
E	(4, C)	(6, A)	(0, None)	(7, F)	(11, B)	(6, C)

The shortest path is CABE with a total cost of 11. To reconstruct this solution, we start with the destination node, then move on to its recorded optimal predecessor. We repeat the process until we reach the starting node. (Note that this isn't the only possible table.)

(b) In general, when using Dijkstra's algorithm to find the shortest path between nodes, do you need to use every row of the table? Why or why not?

No. The algorithm is finished as soon as the destination node has been finalized; Dijkstra's is a greedy algorithm so it will never change decisions once they are made.

8. Explain the Model-View-Controller design pattern.
MVC is a way of dividing responsibility within a project's code to prevent it from growing intertwined to the point of unmaintainability. The model is the portion of the code that manages storing, representing, and manipulating the data. The controller is the module that accepts user input and prepares it for use by the model. The view describes the appearance of the user interface.

9. Describe each of the following layout managers:

   (a) `FlowPane` The `FlowPane` class puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, the `FlowPane` class uses multiple rows. If the container is wider than necessary for a row of components, elements in the row will be flushed left by default.

   (b) `BorderPane` `BorderPanes` allow you to specify the positional location to place new elements into the pane, including top, center, bottom, left, and right. You can specify where to put elements by `paneName.set[Position](Node value)`. Elements that are placed in the scene will retain their size if there is enough space.

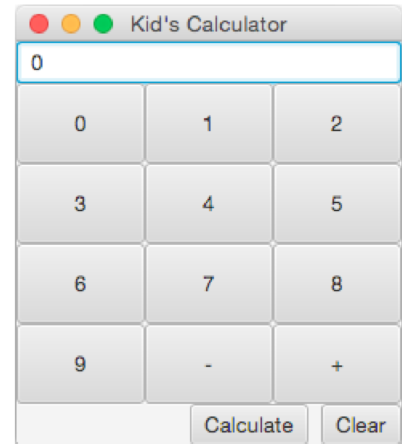
   (c) `GridPane` A `GridPane` places components in a grid of cells. The largest component in any given row or column dictates the size of that row or column, meaning if all your components are the same size, all the grid cells in the pane will be the same size.

   (d) `HBox and VBox` An `HBox` places its components horizonally left-to-right. A `VBox` is like an `HBox` except that it adds components vertically top-to-bottom.

10. Write a function or routine which creates a `Button` with the text "Click me!" that prints out a message of your choosing when it is clicked, using `System.out.println()`. When you've reached a solution, think about how you might be able to reorganize the code which tells the button what do when it is pressed in a different way.

```
 1 // 1: anonymous class (can be done either in-line or in a method, like below)
 2 private EventHandler<ActionEvent> getEventHandler(){
 3     EventHandler<ActionEvent> myEventHandler = new EventHandler<>(){
 4         @Override
 5         public void handle(ActionEvent e){
 6             System.out.println("#ButtonPressed2015");
 7         }
 8     }
 9     return myEventHandler;
10 }
11
12 // 2: create-your-own event handler class
13 class MyEventHandler implements EventHandler<ActionEvent> {
14     @Override
15     public void handle(ActionEvent event) {
16         System.out.println("I'm not anonymous!");
17     }
18 }
19
20 // ...
21
22 Button myButton = new Button("Click me!");
23 myButton.setOnAction(getEventHandler()); // uses #1
24 myButton.setOnAction(new MyEventHandler()); // uses #2
```

11. Create a class that constructs and displays the following GUI. If you can't remember exactly how to implement a certain part of the GUI in code, explain what the component is and how it would fit in with the rest of the calculator. (*Hint: draw out the GUI's* ***component hierarchy***.)



The buttons within the GUI do not need to be functional. You may or may not need the following: `Scene`, `BorderPane`, `FlowPane`, `HBox`, `TextField`, `Button`. The window should fit to all of the components and have a title.

```
1  public class KidsCalc extends Application {
2      private final static int BUTTON_WIDTH = 80;
3      private final static int BUTTON_HEIGHT = 50;
4      private final static int MIN_WIDTH = BUTTON_WIDTH*3;
5
6      public void start(Stage primaryStage) throws Exception {
7          Scene scene = new Scene(this.makeMainPane());
8          primaryStage.setTitle("Kid's Calculator");
9          primaryStage.setScene(scene);
10         primaryStage.show();
11     }
12
13     private Parent makeMainPane() {
14         BorderPane bp = new BorderPane();    //BORDER PANE LAYOUT
15         bp.setPrefWidth(MIN_WIDTH);
16         bp.setTop(makeTopArea());          //TOP
17         bp.setCenter(makeCenterArea());  //CENTER
18         bp.setBottom(makeBottomArea());  //BOTTOM
19         return bp;
20     }
21
22     private Node makeTopArea(){
23         TextField t = new TextField("0");
24         t.setEditable(false);
25         return t;
26     }
27
28     private Node makeCenterArea(){
29         FlowPane flow = new FlowPane();
30         flow.setMinWidth(MIN_WIDTH);
31
32         for(int i = 0; i<10; i++){  // buttons 0 - 9
33             Button b = new Button(String.valueOf(i));
34             b.setPrefSize(BUTTON_WIDTH,BUTTON_HEIGHT);
35             flow.getChildren().add(b);
36         }
37
38         Button minus = new Button("-");
39         minus.setPrefSize(BUTTON_WIDTH,BUTTON_HEIGHT);
40         flow.getChildren().add(minus);
41         Button plus = new Button("+");
42         plus.setPrefSize(BUTTON_WIDTH, BUTTON_HEIGHT);
43         flow.getChildren().add(plus);
44         return flow;
45     }
46
47     private Node makeBottomArea(){
48         HBox bottom = new HBox(8);                //HBox for bottom row
49         bottom.setAlignment(Pos.CENTER_RIGHT);
50         Button calculate = new Button("Calculate"); //calculate button
51         bottom.getChildren().add(calculate);
52         Button clear = new Button("Clear");          //clear button
53         bottom.getChildren().add(clear);
54         return bottom;
55     }
56 }
```

12. Briefly explain the differences between the three kinds of exceptions: checked exceptions, runtime exceptions, and errors.

    checked exceptions - Exceptions that a method signature must specify it throws. If a method may throw a checked exception, all calls to that method must be within a `try-catch` block. Checked exceptions should be used exclusively for foreseeable runtime mistakes, and any reasonably robust system should be able to recover from one. Classic example is `IOException`.

    runtime exception - Not declared in a method signature and not anticipated to be thrown. Usually arise due to software bugs and often cause the program to crash. Classic examples are `NullPointerException` and `ArrayIndexOutOfBoundsException`.

    errors - Represent a serious issue outside of the control of the programmer (hard drive failure, not enough memory, device issue). Examples are `IOError`, `VirtualMachineError` and `ThreadDeath` (see Java's `Error` class).

13. Is there anything wrong with the following exception handler as written? Will this code work as intended?

```
try {
    this.epicFail();
} catch (Exception e) {
    ...
} catch (ArithmeticException a) {
    ...
}
```

    `Exception` is more broad than `ArithmeticException`, so the second `catch` statement is unreachable. The `catch` statements should filter possible exception types from *most* specific to *least* specific.

14. What is the output when `LookAtDatMagic`'s main is executed?

```
1  public class HeySteve{
2      public int bananza(int in) throws NewException{
3          if ( in == 7 ){
4              throw new NewException("HeySteve, cut that out!");
5          }
6          return in;
7      }
8  }
9
10 public class NewException extends Exception{
11     public NewException(String message){
12         super(message);
13     }
14
15 }
16
17 public class WakaWaka{
18     public String BeachBash(Object a, Object b) throws NewException{
19         if ( a.equals(b) ){
20             throw new NewException("It's a Beach-bash! WakaWaka!");
21         }
22         return "Da-nanananan";
23     }
24 }
25
26 public class LookAtDatMagic{
27     public void magic() throws NewException{
28         int maraca = 5;
29         try{
30             HeySteve steve = new HeySteve();
31             maraca = steve.bananza(7);
32         }catch(NewException e){
33             System.out.println(e.getMessage());
34         }finally{
35             WakaWaka waka = new WakaWaka();
36             System.out.println(waka.BeachBash(maraca, 5));
37         }
38     }
39
40     public static void main(String[] args){
41         try{
42             LookAtDatMagic ladm = new LookAtDatMagic();
43             ladm.magic();
44         }catch(NewException e){
45             System.out.println(e.getMessage());
46         }
47     }
48 }
```

HeySteve, cut that out!

It's a Beach-bash! WakaWaka!

15. Consider a simple backtracking algorithm.

(a) What are the four core components of any backtracking `solve` function?

i. Checking if the current state is the solution - `isGoal()`,

ii. Getting the next possible state - `getSuccessors()`,

iii. Checking if the next state is valid - `isValid()`,

iv. Calling solve on the new valid state - `solve()`

(b) Write a generic `solve()` function for a given configuration, which returns either the solution configuration or `null` (if no solution exists):
(*Hint: make up a function name for each of the parts above, if necessary*)

```
public Configuration solve(Configuration config) {

    if(config.isGoal()){
        return config;
    }else{
        for(Configuration child : config.getSuccessors()){
            if(child.isValid()){
                Configuration ans = solve(child);
                if(ans != null)
                    return ans;
            }
        }
        // implicit backtracking happens here
    }
    return null;

}
```