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.

\[ \begin{align*}
& \text{ACEDF} \\
& \text{The traversal order is ACEDF.}
\end{align*} \]

(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.
\[ \begin{align*}
& \text{ABCD} \\
& \text{The traversal order is ABCDEF.}
\end{align*} \]

(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!)

```java
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?)

```java
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. 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 `IOException`, `VirtualMachineError` and `ThreadDeath` (see Java's `Error` class).

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

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

```java
public class HeySteve{
    public int bananza(int in) throws NewException{
        if ( in == 7 ){
            throw new NewException("HeySteve, cut that out!");
        }
        return in;
    }
}

class NewException extends Exception{
    public NewException (String message)
    {
        super(message);
    }
}

class WakaWaka{
    public String BeachBash(Object a, Object b) throws NewException{
        if ( a.equals(b) ){
            throw new NewException("It's a Beach-bash! WakaWaka!");
        }
        return "Da-nanananan";
    }
}

class LookAtDatMagic{
    public void magic() throws NewException{
        int maraca = 5;
        try{
            HeySteve steve = new HeySteve();
            maraca = steve.bananza(7);
            } catch ( NewException e){
                System.out.println(e.getMessage());
            } finally{
                WakaWaka waka = new WakaWaka();
                System.out.println(waka.BeachBash(maraca, 5));
            }
        }
    public static void main(String[] args){
        try{
            LookAtDatMagic ladm = new LookAtDatMagic();
            ladm.magic();
        } catch ( NewException e){
            System.out.println(e.getMessage());
        }
    }
}
```

HeySteve, cut that out!

It's a Beach-bash! WakaWaka!
7. 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):
(\textit{Hint: make up a function name for each of the parts above, if necessary})

```java
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;
}
```