1. Explain the differences between:

(a) **class vs object**
    A class defines methods and fields—it can be viewed as a template. An object is an instance of a class. Think of classes as molds and objects as individual things created by those molds.

(b) **constant vs non-constant field (variable). Declare a constant.**
    A constant field cannot be changed during run time (it may be set once).
    ```java
    public final int NUM_PEOPLE_WHO_LIKE_JAVA = 1;
    ```
    Note that marking mutable types `final` only prevents them from being reassigned; they can still call methods that mutate themselves. For example, you can declare a `final ArrayList`, but this does not prevent you from modifying the contents of it, such as through the use of the `add(T toInsert)` method.

(c) **final vs non-final method.**
    A final method can’t be overridden by a subclass.

(d) **class vs instance variable. Declare variables of both types.**
    Class (static) variables are associated with the class, not to instances of that class. All instances of a class access the same static variable. However, instance (non-static) variables are unique to each instance.
    ```java
class Types {
    public static int im_a_class_var = 1;
    public int im_an_instance_var = 2;
}
``` 

2. Briefly describe the difference (for objects) between `a.equals(b)`, `a==b`, `a.compareTo(b)`, and `Comparator.compare(a,b)`.

- **a.equals(b)** Compares objects for equality. Class `Object` provides a default implementation (to be precise, it is `==` by default) that can be overridden for behavior necessary for a certain class. Returns a `boolean`.
- **a == b** Checks *memory locations* (if the two objects are the SAME object, as defined by whether or not a and b point to the same spot in memory). Can also be used to check whether a is `null`. Also returns a `boolean`.
- **a.compareTo(b)** Returns an `int` indicating whether a is less than (-1), equal to (0), or greater than (+1) b, according to their natural ordering. Specified by the `Comparable` interface.
- **compare(a,b)** Returns a negative a < b, 0 if a = b, a positive if a > b.
    With two `Comparator` objects, `comp1` and `comp2`, `comp1.equals(comp2)` implies that `sgn(comp1.compare(o1,o2))==sgn(comp2.compare(o1, o2))` for every object reference `o1` and `o2`.
3. Convert the following code to use generics.

```java
interface StringCondition {
    boolean checkString(String s);
}

interface IntegerCondition {
    boolean checkInteger(Integer i);
}

class StringContainer {
    ArrayList<String> values;

    // Add and other methods are defined correctly here...

    String getFirstWhereHolds(StringCondition condition) {
        for (String s : values) {
            if (condition.checkString(s))
                return s;
        }
        return null;
    }
}

class IntegerContainer {
    ArrayList<Integer> values;

    // Add and other methods are defined correctly here...

    Integer getFirstWhereHolds(IntegerCondition condition) {
        for (Integer i : values) {
            if (condition.checkInteger(i))
                return i;
        }
        return null;
    }
}

interface Condition<E> {
    boolean check(E value);
}

public class Container<E> {
    ArrayList<E> values;

    E getFirstWhereHolds(Condition<E> condition) {
        for (E v : values) {
            if (condition.check(v))
                return v;
        }
        return null;
    }
}
4. Reimplement the following function using an **Iterator** instead of a for-each loop.

```java
public static Integer sum ( ArrayList<Integer> lst ) {
    Integer total = 0;
    for ( Integer elem : lst ) {
        total = total + elem;
    }
    return total;
}
```

```java
public static Integer sum (ArrayList<Integer> lst) {
    Integer total = 0;
    for (Iterator<Integer> iter = lst.iterator(); iter.hasNext();) {
        Integer cur_val = iter.next();
        total = total + cur_val;
    }
    return total;
}
```

5. Assume the following line of code is given:

```
Collection<Integer> t = new ArrayList<>();
```

What, then, is wrong with the following? Correct any errors:

(Hint: there are no syntax errors.)

```java
for ( int i = 0; i < 20; ++i )
    t.add(i);
for ( int i=0; i < t.size(); ++i )
    System.out.println(t.get(i));
```

**Collection** does not support **get(i)**. The better solution is:

```java
for ( int i = 0; i < 20; ++i )
    t.add(i);
for ( Integer i : t )
    System.out.println(i);
```
6. 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).
7. Use the following code to answer the questions listed on the next page.

```java
import java.util.ArrayList;
import java.util.Collection;

public class UFO {
    private Collection<Probable> toProbe;
    private int aggression;

    public UFO(int agg) {
        toProbe = new ArrayList<Probable>();
        aggression = agg;
    }

    public void probeEverything() {
        for (Probable p : toProbe) {
            p.probe();
        }
    }

    public void sendHome() {
        for (Probable p : toProbe) {
            p.returnHome();
        }
    }

    public void abduct(Collection<Probable> potentialAbductees) {
        for (Probable p : potentialAbductees) {
            if (p.getMentalResolve() < aggression) {
                toProbe.add(p);
            }
        }
    }

    public static void main(String[] args) {
        UFO ufo = new UFO(1199999999);
        ArrayList<Probable> field = new ArrayList<Probable>();

        field.add(new Cow("Bessie");
        // 10 == mentalResolve
        Human cleatus = new RedNeck("CLEATUS", 10);
        // 50 == mentalResolve and 100 == academicRespect
        Human brown = new Professor("Brown", 50, 100);

        field.add(cleatus);
        field.add(brown);

        ufo.abduct(field);
        ufo.probeEverything();
        ufo.sendHome();
    }
}
```
(a) Why should Probable be an interface, rather than a class or an abstract class?

Probable needs to define that certain actions can be performed on Probable objects, but does not need to define what those actions should do.

(b) Write the Probable interface.

```java
public interface Probable{
    public void probe();
    public int getMentalResolve();
    public void returnHome();
}
```

(c) Should the Redneck and Professor classes implement Probable directly?

No! Since Redneck and Professor objects are stored in variables of type Human, they must extend the Human class. Human must be a class rather than an interface because it has state information that must be inherited. In addition, since the Human variables are able to be added into a collection of Probable objects, the Human class must implement Probable, which will carry down into the Redneck and Professor classes.

8. Name the design pattern used in the following snippet of code.

```java
public class Car {
    private String make;
    private String model;
    private int mileage;
    private Car(String make, String model, int mileage) {
        this.make = make;
        this.model = model;
        this.mileage = mileage;
    }
    public static Car makeCar(String str) {
        String arr[] = str.split(" ");
        return new Car(arr[0], arr[1], Integer.parseInt(arr[2]));
    }
    public static void main(String[] args) {
        Car myCar = Car.makeCar("Toyota Camry 200000");
    }
}
```

The Factory design pattern is used.
9. NullPointerExceptions

(a) Briefly describe what a NullPointerException is.
A NullPointerException occurs (is thrown) when an application attempts to make use of null where an object instance was required. In other words, the exception will occur if you make use of a reference as if it were a object, but it is actually null; the error occurs because objects have functionality that null does not.

(b) Provide a short code example that will throw a NullPointerException without explicitly writing “null” in your snippet, explain why the exception will occur, and finally, explain how you would fix the problem.

```java
import java.util.ArrayList;

public class Example {
    ArrayList<Integer> myList;
    public Example() {
        for (int i = 0; i < 20; i++) {
            myList.add(i);
        }
    }
    public static void main(String[] args) {
        Example foo = new Example();
    }
}
```

Above, we declare `myList`, but we don’t actually give it a value. This is not a compile-time error; Java will recognize ‘myList’ as a valid variable for use in the rest of your code, but it will set its value to null, since we didn’t initialize the variable. Since `myList` is still null by the time we reach the inside of the for loop, the line that is executed would effectively read “null.add(0)”, which is clearly invalid. Note that not initializing local variables declared inside methods will produce an error at compile time. However, instance variables and class (‘static’) variables are initialized to a default value, which is ‘null’ for object types. We may remedy this situation by initializing our list, by adding the following line at the beginning of our constructor:

```
    myList = new ArrayList<>(); // java7 diamond operator ftw!
```

(c) What is the most common mistake programmers make that lead to NullPointerExceptions?
The most common mistake that programmers make which causes NullPointerExceptions is the one we have exemplified above - forgetting to initialize your variables before you use them.
10. Suppose you have the classes DecodeStream and EncodeStream which decodes and encodes text in a particular format. Assume that both classes buffer I/O and have readLine() which returns a string and writeLine( String s ) respectively to read and write a line at a time.

Write code to read the file "secret.txt", decode it using the DecodeStream, and print the contents.

```java
String line;
DecodeStream in = new DecodeStream( new FileReader( "secret.txt" ) );
while( ( line = in.readLine() ) != null ) {
    System.out.println( line );
}
```

Write code to encode and write a string called "message" to a file.

```java
EncodeStream out = new EncodeStream( new FileWriter( "message.txt" ) );
out.writeLine( message );
```

11. 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.

   The traversal order is ACEDF.

   (c) What path from A to F does the DFS algorithm return?
       ACDF
12. 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

13. 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.

14. 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.

15. 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.
16. Consider the following graph.

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

<table>
<thead>
<tr>
<th>Finalized</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
<td>(0, None)</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
</tr>
<tr>
<td>C</td>
<td>(4, C)</td>
<td>(∞, None)</td>
<td>(0, None)</td>
<td>(∞, None)</td>
<td>(∞, None)</td>
<td>(6, C)</td>
</tr>
<tr>
<td>A</td>
<td>(4, C)</td>
<td>(6, A)</td>
<td>(0, None)</td>
<td>(11, A)</td>
<td>(∞, None)</td>
<td>(6, C)</td>
</tr>
<tr>
<td>F</td>
<td>(4, C)</td>
<td>(6, A)</td>
<td>(0, None)</td>
<td>(7, F)</td>
<td>(12, F)</td>
<td>(6, C)</td>
</tr>
<tr>
<td>B</td>
<td>(4, C)</td>
<td>(6, A)</td>
<td>(0, None)</td>
<td>(7, F)</td>
<td>(11, B)</td>
<td>(6, C)</td>
</tr>
<tr>
<td>D</td>
<td>(4, C)</td>
<td>(6, A)</td>
<td>(0, None)</td>
<td>(7, F)</td>
<td>(11, B)</td>
<td>(6, C)</td>
</tr>
<tr>
<td>E</td>
<td>(4, C)</td>
<td>(6, A)</td>
<td>(0, None)</td>
<td>(7, F)</td>
<td>(11, B)</td>
<td>(6, C)</td>
</tr>
</tbody>
</table>

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.

17. Given a node in an array-based binary heap at index i, where are the indices of both its children? What is the index of its parent?

The children are at 2i + 1 and 2i + 2. The parent is at ⌊i−1/2⌋.

18. For a binary heap containing n elements, what is the maximum number of swaps occurring after an insert operation?

log₂ (n + 1), rounded down.
19. Chris made a mistake in his hash table implementation!

```java
import java.util.ArrayList;

public class WonkyHashTable {
    private int size;
    private ArrayList<String> table;

    public WonkyHashTable(int s) {
        size = s;
        table = new ArrayList<String>(size);
        for(int i = 0; i < s; i++) {
            table.add("");
        }
    }

    public void add(String element) {
        int hash = bad_hash(element);
        table.set(hash, element);
        System.out.println(table.toString());
    }

    private boolean contains(String element) {
        for (String t : table) {
            if(element.equals(t)) {
                return true;
            }
        }
        return false;
    }

    private int bad_hash(String element) {
        return element.length() % table.size();
    }

    public static void main(String args[]) {
        WonkyHashTable htable = new WonkyHashTable(4);
        for(String s : "I wrestled a bear once".split(" ")) {
            htable.add(s);
        }
    }
}
```

(a) Show what the hash table looks like after the for loop in main completes.

```
[once, a, ' ', ' ']
```

(b) What is wrong with the code? What can we do to make the function behave as Chris expects it to behave?

Whenever `bad_hash()` dictates that an element should be placed in an occupied bucket, that bucket’s contents get overwritten! We need to change the table to be a list of lists.

```java
private ArrayList<ArrayList<String>> table;
```
We have to initialize the sublists in the constructor as well as add each element like so:

```
table.get(hash).add(element);
```

(c) Draw the table of the properly behaving hash function.

```
[ [ 'wrestled', 'bear', 'once' ], [ 'I', 'a' ], [], [] ]
```

(d) Assuming that this hash table will only be used on strings, is the hashing function being used a good one? Why or why not?

No: It ignores the fact that most English words are the roughly the same length. The number of collisions is expected to be massive. We should take advantage of the characters in the input strings, not the number of characters.

20. Rick owns a positively popular pizza place conveniently located just off of campus. Originally, he made all the pizzas himself, but rising campus food prices are making demand skyrocket. Luckily, the college students are as desperate for money as they are for food, a situation from which Rick, being a pragmatic individual, finds he can benefit. Drawing from the exploitable labor pool, Rick turns his already-hot kitchen into a sweat shop, ordering his workers like so:

```
for ( PizzaSlave student : laborPool ) {
    new Thread(student).run();
}
```

Sensing an early retirement, Rick promotes his first hire from slave to manager, rewarding him with slightly higher—but still illegal—pay. (Despite these perks and the envy of his peers, the manager is just like everyone else.) Alas, when Rick returns a few days later, he is so displeased with what he sees that he fires the manager on the spot. What made Rick so angry, and what should he have done differently to prevent its happening?

In his haste to make a profit, Rick mistakenly called the `Thread` class’s `run()` method, which caused his manager to run synchronously and make pizzas while everyone else stood around waiting. Rick should instead have called `start()`, which would have whipped all of his workers into shape at roughly the same time.

21. The Life of Nick: Nick, an aspiring entrepreneur, trained for 7 grueling years in the jungles of Zimbabwe. Nick’s preparation was overseen by a group of trainers. His stages of learning were fueled by the following process:

```
public class Nick {
    private int experience = 0;
    public void train() {
        this.experience += 1;
    }
}
```

Imagine instructors are threads in Java. What are some problems we may encounter if Nick is having multiple people train him at the same time? How might we remedy these issues?

The train function, and specifically its incrementation, is non-atomic, meaning the value of Nick’s `experience` variable will be undefined after multiple threads attempt to call the function concurrently. One solution would be to synchronize on `Nick` to
ensure only one thread at a time executes within the critical section.

22. **Nick’s Heavy Threads:** Nick now operates a store in Marketview Mall which has poor lighting, blasts black metal and sells jeans. Only one pair of jeans is available to purchase at a time, though there are more stored in the back. If a size is out that you don’t want, you must wait for someone else to purchase the jeans. Nick’s only employee, Hank, sits in a chair and stares at people angrily until someone makes a purchase, at which point he replaces the jeans with the same model of a random size. In order to prevent customers’ waiting infinitely for an unavailable size, Hank will switch the jeans for a different size pair if no one has bought them after a period of three seconds.

```java
public class NicksHeavyThreads {
    // jeans' size [1-5], or 0 when none on display
    private static int awesomeJeans = 0;

    // keep this updated as customers arrive and leave
    private static int customers = 0;

    private static MeanWorker hank = new MeanWorker();

    public static void main(String[] args) {
        for (int i = 0; i < 10; ++i) {
            (new LameCustomer()).start();
        }

        // wait one second before introducing Hank
        try { Thread.sleep(1000); } catch (InterruptedException pleaseDont) {} hank.start();
    }

    private static class LameCustomer extends Thread {
        // (implementation omitted)
    }

    private static class MeanWorker extends Thread {
        // (implementation omitted)
    }
}
```

(Questions may be found on the next page. You may answer them in the space allotted here, or on the following page.)
(a) Complete the implementation of the `LameCustomer` class: Each instance must choose a jeans size and wait for it to be available, update the jeans to indicate that they have been taken, print the message “Customer: I got my size \textit{size} jeans!” and inform all threads that the jeans selection has changed.

\textit{(Hint: Remember to keep an accurate count of how many customers are in the shop.)}

```java
static class LameCustomer extends Thread {
    public synchronized void run() {
        ++customers;
        // pick size
        int desiredSize = (int) (Math.random()*(5) ) + 1;
        // wait for pair
        while( awesomeJeans != desiredSize ) {
            try { hank.wait(); }
            catch( InterruptedException pleaseDont ) {} 
        }
        System.out.println( "Customer: I got my size \\
                             + awesomeJeans + " jeans!"
        hank.notifyAll(); // take the jeans
        --customers; // inform everyone they are gone
    }
}
```

(b) Now implement the `MeanWorker` class, which should choose a size and stock a pair of jeans of that size, print the message “Hank: I grumpily restocked with size \textit{size},” and inform all threads that the selection has changed. It should then wait until someone has taken the jeans or until three seconds have elapsed, whichever comes first. These steps should be repeated until all customers have left the store.

```java
static class MeanWorker extends Thread {
    public synchronized void run() {
        do {
            // new size
            awesomeJeans = (int) (Math.random()*(5) ) + 1;
            System.out.println( "Hank: I grumpily restocked \\
                                + with jeans of size " + awesomeJeans );
            notifyAll(); // inform customers of the restocking
            try { wait( 3000 ); } // let people shop
            catch( InterruptedException pleaseDont ) {} 
        }
        while( customers > 0 );
    }
}
```
23. 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.

```java
public class KidsCalc extends Application {
    private final static int BUTTON_WIDTH = 80;
    private final static int BUTTON_HEIGHT = 50;
    private final static int MIN_WIDTH = BUTTON_WIDTH * 3;

    public void start(Stage primaryStage) throws Exception {
        Scene scene = new Scene(this.makeMainPane());
        primaryStage.setTitle("Kid’s Calculator");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private Parent makeMainPane() {
        BorderPane bp = new BorderPane(); // BORDER PANE LAYOUT
        bp.setPrefWidth(MIN_WIDTH);
        bp.setTop(makeTopArea()); // TOP
        bp.setCenter(makeCenterArea()); // CENTER
        bp.setBottom(makeBottomArea()); // BOTTOM
        return bp;
    }

    private Node makeTopArea() {
        TextField t = new TextField("0");
        t.setEditable(false);
        return t;
    }

    private Node makeCenterArea() {
        FlowPane flow = new FlowPane();
        flow.setMinWidth(MIN_WIDTH);
        for (int i = 0; i < 10; i++) { // buttons 0 - 9
            Button b = new Button(String.valueOf(i));
            b.setPrefSize(BUTTON_WIDTH, BUTTON_HEIGHT);
            flow.getChildren().add(b);
        }
        Button minus = new Button("-");
        minus.setPrefSize(BUTTON_WIDTH, BUTTON_HEIGHT);
        flow.getChildren().add(minus);
        Button plus = new Button("+");
        plus.setPrefSize(BUTTON_WIDTH, BUTTON_HEIGHT);
        flow.getChildren().add(plus);
        return flow;
    }

    private Node makeBottomArea() {
        // Code for making the bottom area
    }
}
```
24. Networking

(a) What does TCP stand for? Where and why do we use TCP?
TCP stands for Transmission Control Protocol. We use TCP in Telephone Connection because TCP guarantees packet delivery and thus can be considered "lossless and reliable".

(b) What does UDP stand for? When and where do we use UDP?
User Datagram Protocol. We use UDP when we are managing a tremendous amount of state (ex: weather data, video transmission).

(c) Which one does a stream socket use for data transmission? TCP (or) UDP?
TCP.

(d) Which one does a datagram socket use for data transmission? TCP (or) UDP?
UDP.

(e) What is a datagram?
A datagram is an independent, self-contained message sent over the network with no guarantees.

(f) What is a socket?
A socket refers to the endpoints of logical connections between two hosts, which can be used to send and receive data.

25. Find at least 3 (total) errors in the following code:

Server: echoes one line of data sent to it

ServerSocket pubServer = new ServerSocket(0);
System.out.println(pubServer.getLocalPort());
Socket client;
BufferedReader reader = null;
try {
    reader = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
    } catch (IOException e) {
        System.out.println("IOException : "+ e.getMessage());
    }
String response = null;
try {
    response = reader.readLine();
} catch (IOException e) {
    System.out.println("IOException : " + e.getMessage());
}
System.out.println(response);
pubServer.close();
Client: sends a line of text to a server: server address, port, text

```java
InetAddress server = null;
try {
    server = InetAddress.getByName(args[0]);
} catch (UnknownHostException e) {
    System.out.println("Unknown host");
}
int port = Integer.parseInt(args[1]);
Socket conn = null;
try {
    conn = new Socket(server, port);
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
try {
    System.out.println(args[2]);
} catch (IOException e) {
    System.out.println("IOException: " + e.getMessage());
}
conn.close();
```

(a) Server: reader is not always instantiated!
(b) Server: client never initialized, use pubServer.accept().
(c) Server: Missing client.close().
(d) Server and Client: *.close() should be in a try-catch block.
(e) Client: Need a PrintWriter writer; writer = new PrintWriter(conn.getOutputStream(),
true); writer.println(args[2]); instead of System.out.println(args[2]);

(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)

```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;
}
```