Underground Cables
When reading the problem, it looks as if a minimum spanning tree is requested except for one issue: edges that physically intersect/cross are not allowed to be selected. In the typical algorithm it doesn’t seem as if there’s a method to prevent these edges from being chosen. The key realization is that when the usual algorithm is run on edges with distances generated between vertices on the x-y plane, crossing edges will never be chosen. Consider a picture with crossing edges of a potential minimum spanning tree (Figure 1) that is shown below. In all cases, we can redraw the picture without crossing edges where the alternate edges added are of shorter total length (Figure 2):

Thus, it follows that we can just create all possible edges between all pairs of vertices (points) and then run either Kruskal’s or Prim’s. The key point to note is that since the edge weights are doubles, we must take care with our compareTo method. A common bug in a compareTo method would be something of this nature:

```java
public int compareTo(edge other) {
    return (int)(this.w - other.w);
}
```

The problem with this implementation is that edges with weights that are within 1 of each other are compared as equal with this method. If two edges are equal, there’s no guarantee in which order they will get sorted. Thus, either Prim’s or Kruskal’s may end up considering an edge with length 2.5 before an edge with length 2.2, for example. In doing so, the algorithm may choose the 2.5 weighted edge OVER the 2.2 weighted edge (if the latter ends up causing a cycle with the former).

Thus, the key is just to compare the edge weights as usual with <, >, etc. For this problem, no tolerance checking is necessary, but it’s usually better to give some tolerance when comparing doubles.
Dueling Philosophers
This is a slight twist on the regular topological sort problem. Instead of just answering whether or not there is a valid topological sort, in the case that a topological sort exists, we just determine if there are more than one possible topological sort. Take the usual algorithm and run it to completion. This will distinguish between cases with and without a topological sort. Now, edit as follows: as the algorithm is running, keep track of if there were ever two possible vertices that could have been placed in a particular slot (ie. two vertices with a current in-degree of 0) in the topological sort. This can just be a boolean flag that is initially set to false and toggles to true any time two 0s are noted for in-degrees when the algorithm looks to choose the next vertex of minimum in-degree. At the very end, in the case that a topological sort has been found, we distinguish between outputs of 1 and 2 by simply looking at the value of the boolean flag. If it's false, we return 1, if it's true we return 2.

Relatives
The maximum degree of separation in a connected graph is the longest distance between any two pairs of vertices. This is also known as the diameter of a graph. Since this graph is so small, an algorithm with a run-time of $O(V^3)$ runs plenty fast, since $V \leq 50$. Thus, Floyd-Warshall's can be executed in time to find the shortest distance between all pairs of vertices. Then, we can scan all of these shortest distances to find the largest one. We can handle the disconnected case by either running a single BFS/DFS or simply creating an edge of length 100 between all pairs of vertices that aren't connected. Any shortest distance of 100 represents no path (or a disconnected graph).

Triangular Sums
This is the easiest problem in the set. A definition for $W(n)$ is given to you and you have to calculate $W(n)$ for any input value from 1 to 300. One key observation is that $W(n+1) = W(n) + (n+1)T(n+2)$. Thus, we can simply build up the results, one by one. The triangular numbers themselves have a formula, $T(n) = n(n+1)/2$. (Even if one didn't know that formula, one can create an extra variable to store the sum of the first $i$ integers and add to it.)

Substituting, we have $W(n+1) = W(n) + (n+1)(n+2)(n+3)/2$ and $W(0) = 0$. From there we can just run a single for loop building up these values and storing them in an array. Then, process the input answering each query immediately.