//GLASS FALLING

Given: n floors, m glass sheets
Find: What's the minimum amount of trials?

(a) Describe the optimal substructure/recurrence that would lead to a recursive solution

We need to find the floor that yields the minimum number of trials in the worst case.
Because we assume the worst case,
for each floor,
we need to find the max of the two cases:
1) If the glass shatters,
we can eliminate all floors above it and consider floors-1 with one less sheet.
2) If the glass does not shatter,
we can consider all the floors above it and eliminate lower floors: floors-currentFloor.

We traverse the floors starting at the first floor.
For each floor, we take the worst case scenario by recursively calculating the max number of trials needed.
This is done by comparing the max number of trials for the two cases above.
Once we have calculated the max for the given floor, we compare that to the current minimum number of trials needed.
If the current max is less than the current min, we set min equal to it.

(b) Draw recurrence tree for given (floors = 4, sheets = 2)

Using recurrence:

```
y = Math.max (recurse(x-1, sheets-1), //check lower floors, assume sheet broke
              recurse(floors-x, sheets)); //check upper floors, assume sheet okay
```

F=4, S=2
recurse[4,2]

Floors |x|

// floor |1| optimal solution = 4

```
recurse[0,1]                              recurse[3,2] // return 3
  |                                         |
  0                          recurse[2,1]   recurse[2,2] // return 2
                               |              |
                               2   recurse[1,1]  recurse[1,2] // return 1
                                     |             |
                                     1             0
```

//floor |2| optimal solution = 3

```
recurse[1,1] // return 1                recurse[2,2] // return 2
    |                                       |
    1              recurse[1,1]  recurse[1,2]  // return 1
                       |              |
                       1              1
```

//floor |3| optimal solution = 3

```
recurse[2,1] // return 2                recurse[1,2] // return 1
    |                                       |
    2                                       1
```

//floor |4| optimal solution = 4

```
recurse[3,1] // return 3                recurse[0,2] // return 0
    |                                       |
    3                                       0
```

(c) Code your recursive solution under GlassFallingRecur(int n numFloors, int m numGlass)
// see code

(d) How many distinct subproblems do you end up with given 4 floors and 2 sheets?
// You end up with 8 distinct subproblems
0,1
3,2
2,2
1,1
1,2
3,1
2,1
0,2

(e) How many distinct subproblems for n floors and m sheets? [n = floors, m = sheets]
Let s = the number of distinct subproblems
If m = 1, s = n.
If m = 2, s = 2n
For m > 2, s = 2n + # remaining subproblems for [n-1, m-1]
For m > 2, s = 2n + (n-1) + (n-2) + … + (n-k) where k = m – 2
We can write this as 2n + (Rieman‘s sum of [n-k] from 1 to m-2).

(f) Describe how you would memoize GlassFallingRecur

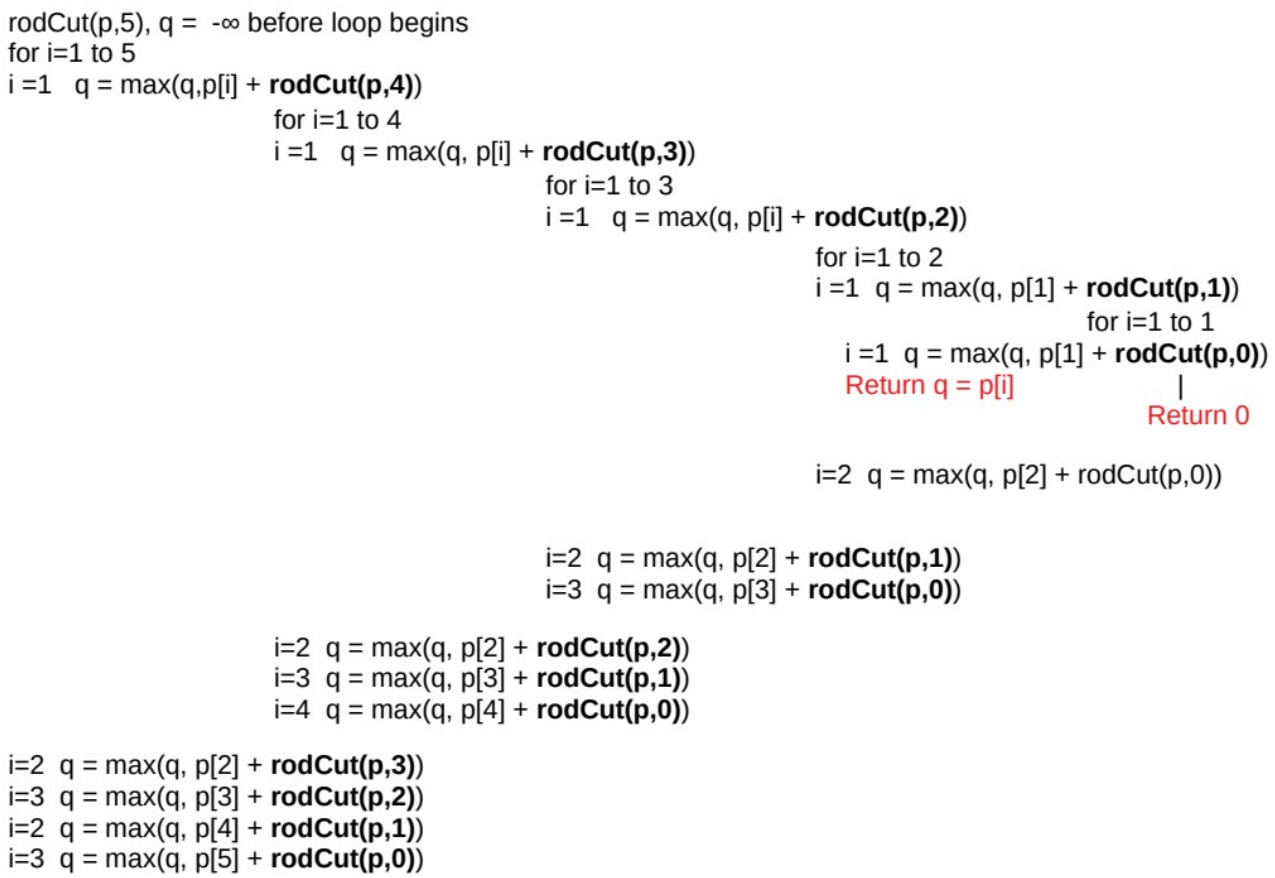
You save the data in a 2-d array and return the answer as an entry in the array. You check, before doing recursion, whether the answer for a given floors and sheets combination is already stored in the array.

(g) Code a bottom-up solution GlassFallingBottomUp(int n numFloors, int m numGlass)
// see code

Turned in: A pdf write-up of parts: a, b, d, e, f with clear and careful explanations! Coding parts c, g in the file GlassFalling.java

// ROD CUTTING

(a) Draw the recursion tree for a rod of length 5



(b) On page 370: answer 15.1-2 by coming up with a counterexample, meaning come up with a situation / some input that shows we can only try all the options via dynamic programming instead of using a greedy choice.

The greedy choice will only work if the local optimal solution is the global optimal solution. However, we can find a counterexample to show that the local optimal solution will not always be globally optimal. Therefore, we must try all the options.

Take a rod of length 4 where we have the following length and value pairs, and their corresponding densities:

Length	Value	Density
1	5	5
2	35	17.5
3	60	20
4	10	2.25

In the greedy algorithm, we choose the rod with length 3, the length with the greatest density, then we can only choose a rod of length 1 for the next choice.

Greedy solution: $60 + 5 = 65$

Optimal solution: $35 + 35 = 70$

However, the optimal solution is to choose two rods of length 2.

Thus we have a counterexample to a claim that the greedy solution is the optimal solution for this rodcutting scenario.