//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.
2) If the glass does not shatter,
we can consider all the floors above it and eliminate lower floors, so 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.

Excerpt of code:

```
if (floors == 1 || floors == 0)
   return floors;

// cannot have any trials if we have no sheets
if (sheets == 0) return 0;

// if we have only 1 sheet would need to try all the floors in the worst case
if (sheets == 1)
   return floors;

int min = Integer.MAX_VALUE; // set min to a high number, to tell if it didn't change
int x, y;

// as we go up the floors,
for (x = 1; x <= floors; x++) {
   y = Math.max(glassFallingRecur(x-1, sheets-1), //check lower floors, assume sheet broke
            glassFallingRecur(floors-x, sheets)); //check upper floors, assume sheet okay
   if (y < min)
      min = y; // set min to value above
   System.out.println("This x is " + x + "with a min of " + min);
}
return min + 1; // we add 1 to show we made it through another trial
```

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

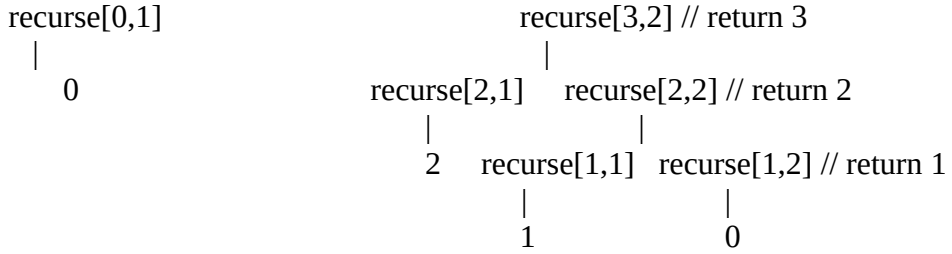
Using recurrence:

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

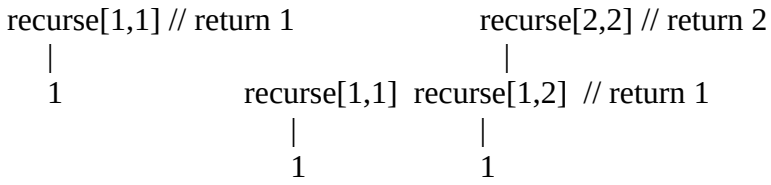
F=4, S=2
recurse[4,2]

Floors |x|

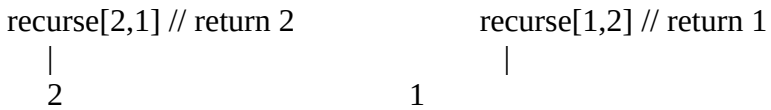
// floor |1| optimal solution = 4



//floor |2| optimal solution = 3



//floor |3| optimal solution = 3



//floor |4| optimal solution = 4

(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]
The number of distinct subproblems is: 2n + # remaining subproblems for [n-1, m-1]
This turns out to be 2n + (n-1) + (n-2) + ... + (n-x)
where x = floors - 2.

We can write this as 2n + the Rieman's Sum from x=n-2 to n.

(f) Describe how you would memoize GlassFallingRecur

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

Turn 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

rodCut(p,5), q = -∞ before loop begins
for i=1 to 5
i =1 q = max(q,p[i] + **rodCut(p,4)**)
for i=1 to 4
i =1 q = max(q, p[i] + **rodCut(p,3)**)
for i=1 to 3
i =1 q = max(q, p[i] + **rodCut(p,2)**)
for i=1 to 2
i =1 q = max(q, p[1] + **rodCut(p,1)**)
for i=1 to 1
i =1 q = max(q, p[1] + **rodCut(p,0)**)
Return q = p[i] |
Return 0

i=2 q = max(q, p[2] + rodCut(p,0))

i=2 q = max(q, p[2] + **rodCut(p,1)**)
i=3 q = max(q, p[3] + **rodCut(p,0)**)

i=2 q = max(q, p[2] + **rodCut(p,2)**)
i=3 q = max(q, p[3] + **rodCut(p,1)**)
i=4 q = max(q, p[4] + **rodCut(p,0)**)

i=2 q = max(q, p[2] + **rodCut(p,3)**)
i=3 q = max(q, p[3] + **rodCut(p,2)**)
i=2 q = max(q, p[4] + **rodCut(p,1)**)
i=3 q = max(q, p[5] + **rodCut(p,0)**)