```
function moony(n) {
    return n === 1
           ? circle
           : stack_frac(1/n,
                beside_frac(1/n, circle, blank),
                beside_frac(1/n, square, moony(n - 1)));

}

show(moony(5));
```

The time and space for evaluating moony has order of growth of **O(n)**
Assumption: stack_frac and beside_frac runs in O(1) time

Hand-wavy explanation (understanding this is enough for this week)
Conditional expression (? :), stack_frac, and beside_frac all evaluates in constant time so these can be ignored. Then the sequence of function calls therefore looks like this:
moony(n) -> moony(n-1) -> moony(n-2) -> … -> moony(2) -> moony(1)
You have to call moony n times so the time and space required has order of growth O(n)

Mathematical explanation (good to know, useful for more difficult questions)
Key understanding (using time as an example but it's the same idea for space):

> time needed to solve a recursive problem =
> time needed in the current step (everything that's not the recursive call) +
> time needed to solve the smaller subproblem(s)

We can let **T(n)** be the time needed to solve a recursive problem of size n

For moony, since the conditional expression (? :), stack_frac, and beside_frac all evaluates in O(1) time, we can write:

T(n) = 4*O(1) + T(n-1)
T(n) = O(1) + T(n-1)
T(n) = O(1) + O(1) + T(n-2)
…
T(n) = O(1) + O(1) + … + O(1)
T(n) = O(n)

It's not important to know how to solve recurrence relations (because you can just put it in your cheat sheet). But it is very important to know how to write derive the recurrence relation for a given recursive function.