1. Counting Operations: Image Loops

Images are stored as grids of pixels. The function below paints every pixel in an even-numbered column black.

```
def paint_even_cols(img, height, width):
    for y in range(height):
        print("outer loop")
        for x in range(width):
            print("inner loop")
            if x % 2 == 0:  # even columns only
                img.set_rgb(x, y, 0, 0, 0)
```

a. Suppose `height = 3` and `width = 4`. List every `(x, y)` pair for which `img.set_rgb` gets called.

y	x **values where** `set_rgb` **is called**
0	**0, 2**
1	**0, 2**
2	**0, 2**

b. For a general image with `height` $= h$ rows and `width` $= w$ columns, how many times does each `print` run?

`print("outer loop")` runs **h** times.

`print("inner loop")` runs $h \times w$ times.

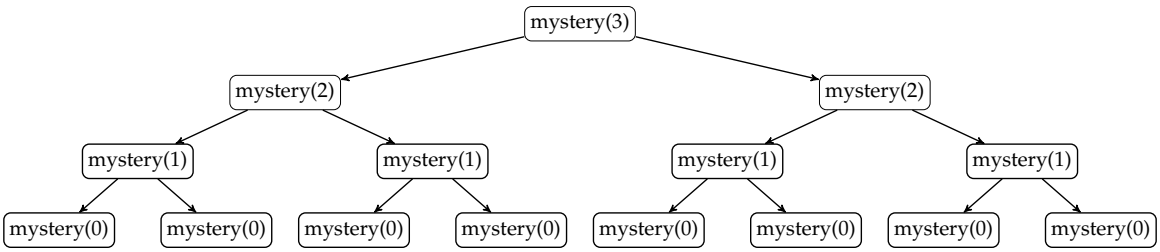
c. Now suppose the image is *square* (`height` $=$ `width` $= n$). What is the Big-O runtime?

$$O(\ \mathbf{n^2}\ )$$

2. Recursive Calls

Here is a recursive algorithm. The call tree below shows all the calls made when we run `mystery(3)`.

```
def mystery(n):
    if n == 0:
        return 1
    return mystery(n - 1) + mystery(n - 1)
```



a. Every call to `mystery(n)` (for $n \geq 1$) makes exactly **2** recursive calls.

b. What does `mystery` return for small values of n?

`mystery(0)` = **1** `mystery(1)` = **2** `mystery(2)` = **4** `mystery(3)` = **8**

In general, `mystery(n)` returns 2^n

c. Count how many times each call appears in the tree above.

Call	Times called
`mystery(3)`	**1**
`mystery(2)`	**2**
`mystery(1)`	**4**
`mystery(0)`	**8**
Total calls	**15**

d. Complete the table below for larger values of n.

n	**Total calls to** `mystery`
0	1
1	3
2	7
3	**15**
4	**31**
5	**63**

e. Each time n increases by 1, the total number of calls (circle one):

stays the same **doubles** **triples** **grows by** n

f. Based on your answer to part **e**, what is the Big-O runtime of `mystery(n)`?

$O($ **2^n** $)$