Orders of Growth

For each function below:

- Break it up into blocks, as we did in the chapter.
- Identify the order of growth for each block, or draw the recursive tree if applicable.
- Identify the order of growth for the whole function.

```
def min(lst):
    if not lst:
        return None
    min = lst[0]
    for elem in lst:
        if elem < min:
            min = elem
    return min
```

Θ(1)

Θ(1)

Θ(n)

Θ(1)

Θ(n)

```
def captains_log(n):
    while n > 0:
        j = 1
        while j < n:
            j *= 2
            print("Captain's log")
        n -= 1
```

Θ(1)

Θ(1)

Θ(log n)

Θ(1)

Θ(n log n)

```
def turbo_count(n):
    for i in range(42000):
        print(i)
```

Θ(1)

Θ(1)

```
def blink(n):
    i = 1
    while i < n ** 2:
        for j in range(i):
            print()
        i *= 2
```

Θ(1)

Θ(1)

Θ(i)

Θ(1)

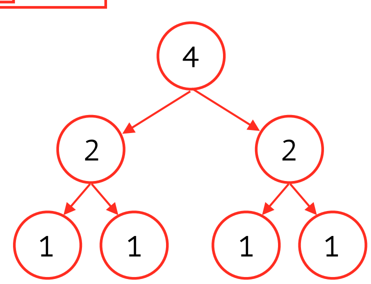
Θ(n^2)

Θ(n^2)

This function represents the sum 1 + 2 + 4 + 8 + ... + n^2. More formally, it's the sum over i=[0 to log n^2] of 2^i, which equals n^2. Note: This is one of the hardest analysis problems you'll see in 61A. It is worth knowing this mathematical identity, in particular.

```
def layer_cake(n):
    if n > 1:
        layer_cake(n / 2)
        layer_cake(n / 2)
```

Θ(n)



In each call, n is cut in half so each node is Θ(log n). However, there are two branches for each call, so there are 2^n nodes. The runtime is Θ($2^{\log n}$) = Θ(n).

```
def fib_iterative(n):
    curr, next = 0, 1                            Θ(1)
    while n > 0:
        curr, next = next, curr + next      Θ(1)  Θ(n)  Θ(n)
        n -= 1                              Θ(1)
    return curr                                  Θ(1)
```

```
def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)    Θ(ϕⁿ)
```

$\Theta(2^n)$ is also fine here, but technically it is not as exact as $\Theta(\phi^n)$, where ϕ is the golden ratio.

Code Writing

Write a program called `factors` that takes in a number n, and returns a set containing all the factors of n. It should run in $\Theta(\sqrt{n})$ at most.

```
from math import sqrt
def factors(n):
    factors, i = set(), 1
    while i <= sqrt(n):
        if n % i == 0:
            factors.update([i] if i * i == n else [i, n // i])
        i += 1
    return factors
```