Complex Algorithms

(I) Recursion

It is possible, and oftentimes useful, for a function to call itself.

The jargon is: $\boxed{\text{recursion,}}$ or recursive function callings.

(Note: to recur ⇒ to occur again (and again!))

Concept 1: Having a function call itself is easy. Having an endpoint, other than infinity, can be tricky!!

Example:

```
def count_down(count):
    count_down(count-1)


# main program
count_down(3)
```

Main

count_down(3)

count_down

count = 3

↓

count_down(2)

↓

count = 2

↓

count_down(1)

↓

count = 1

↓

count_down(0)

↓

cont = ∅

↓

count-down (-1)

↓

∞ !! ☹

We can fix this, and get the desired behavior, by creating an "end condition" in the function.

Idea: We don't <u>always</u> recall the function → only under certain conditions.

```
def count_down(count):
    if (count == 0):
        print("Go!")
    else:
        print(count)
        count_down(count-1)
```

Count_down(3)

→ 3
2
1
Go!

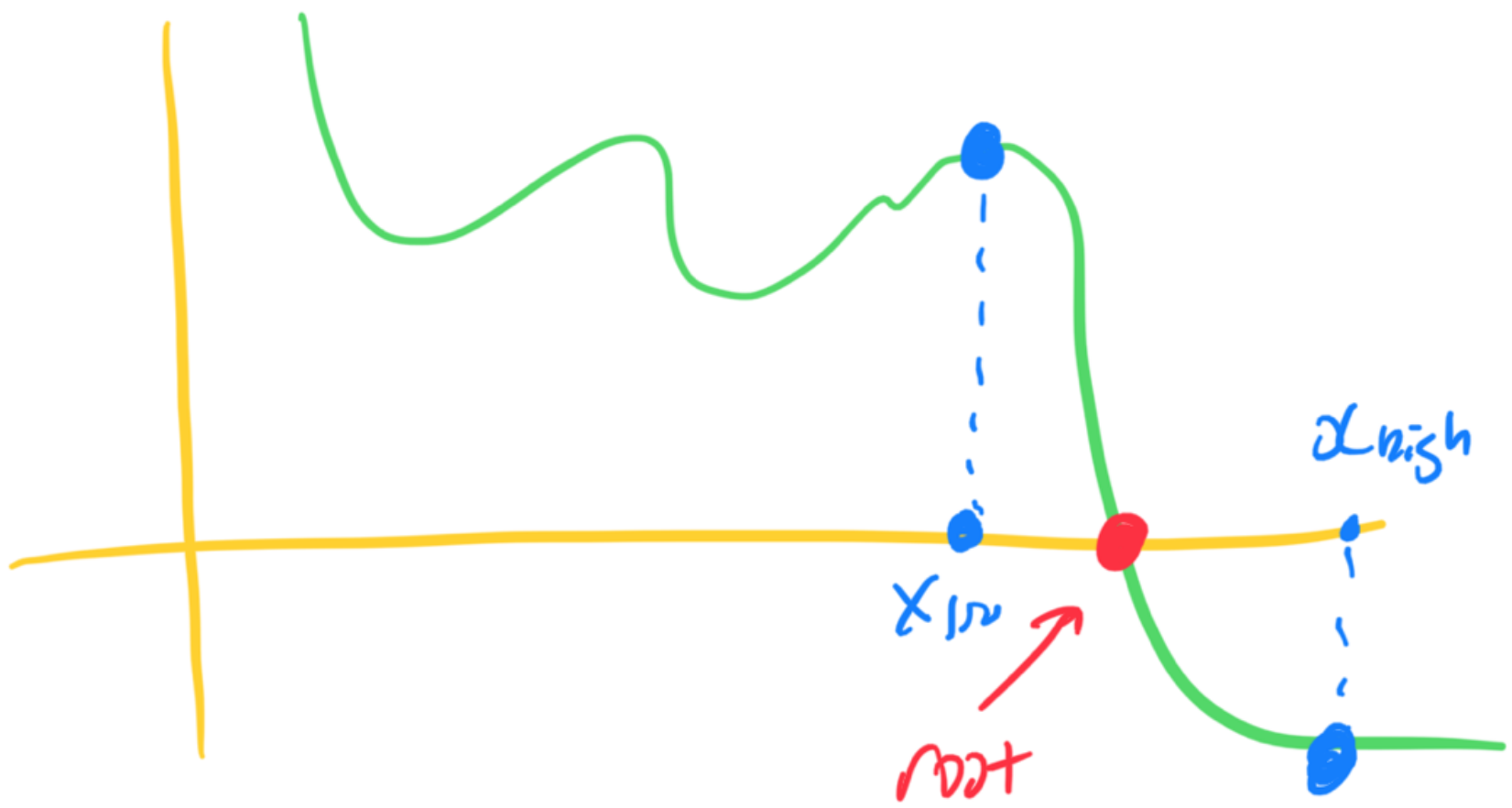
Binary Search Algorithm

The above function was cool, but sort of useless → we could have solved this easily and more intuitively without doing a recursive function. In fact, we usually can!! I'm not a huge fan of recursive functions. They usually are (a) cryptic to figure out and (b) <u>not</u> more efficient than a non-recursive algorithm. An exception, I think

BINARY SEARCH TREE.

B the BINARY SEA

Example: Find the root(s) of a function, $f(x)$



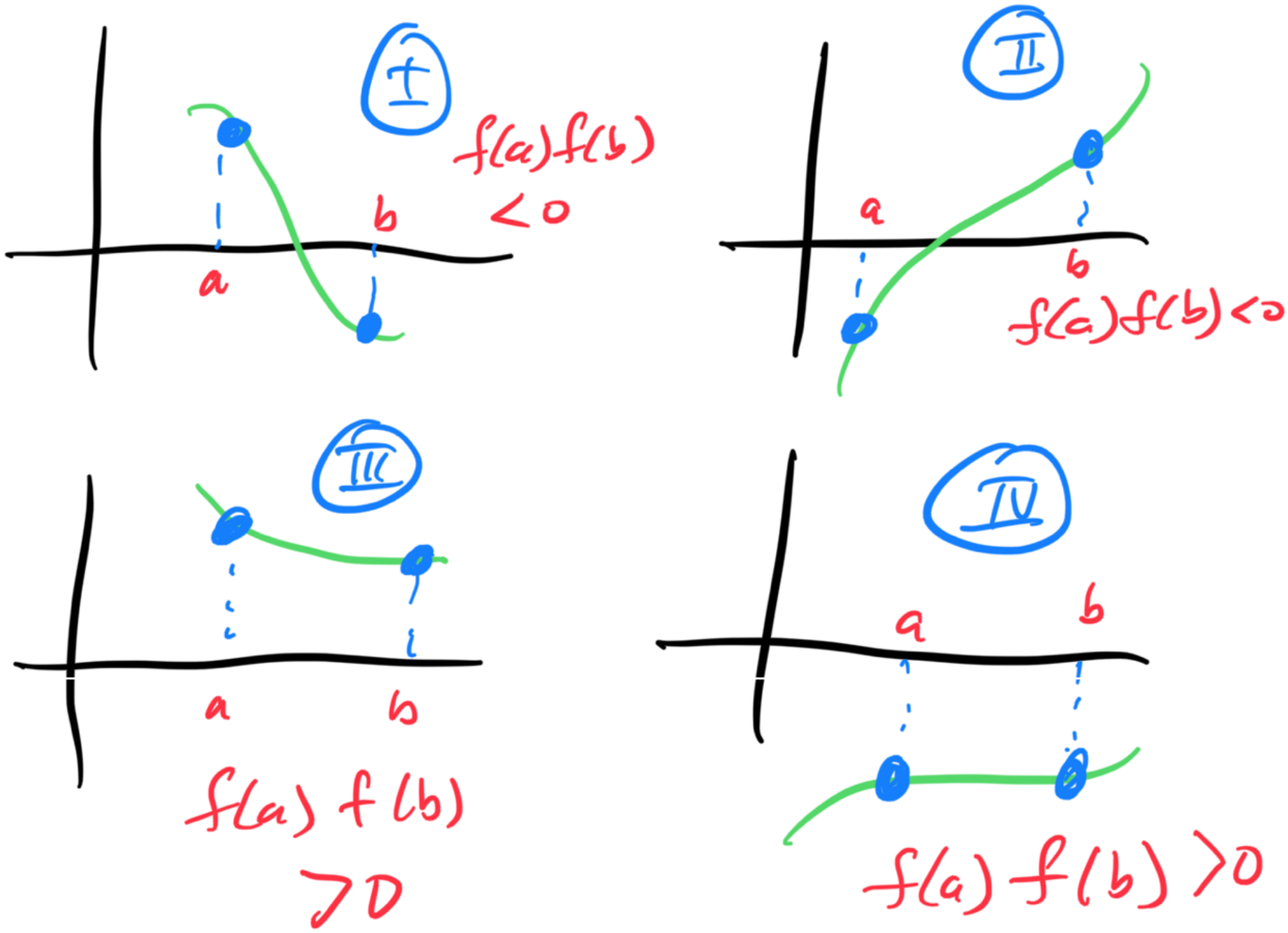
$$f(x) = 0$$

Suppose that we know that the root is somewhere between x_{low} and x_{high}. How would we know, mathematically, that this is the case?

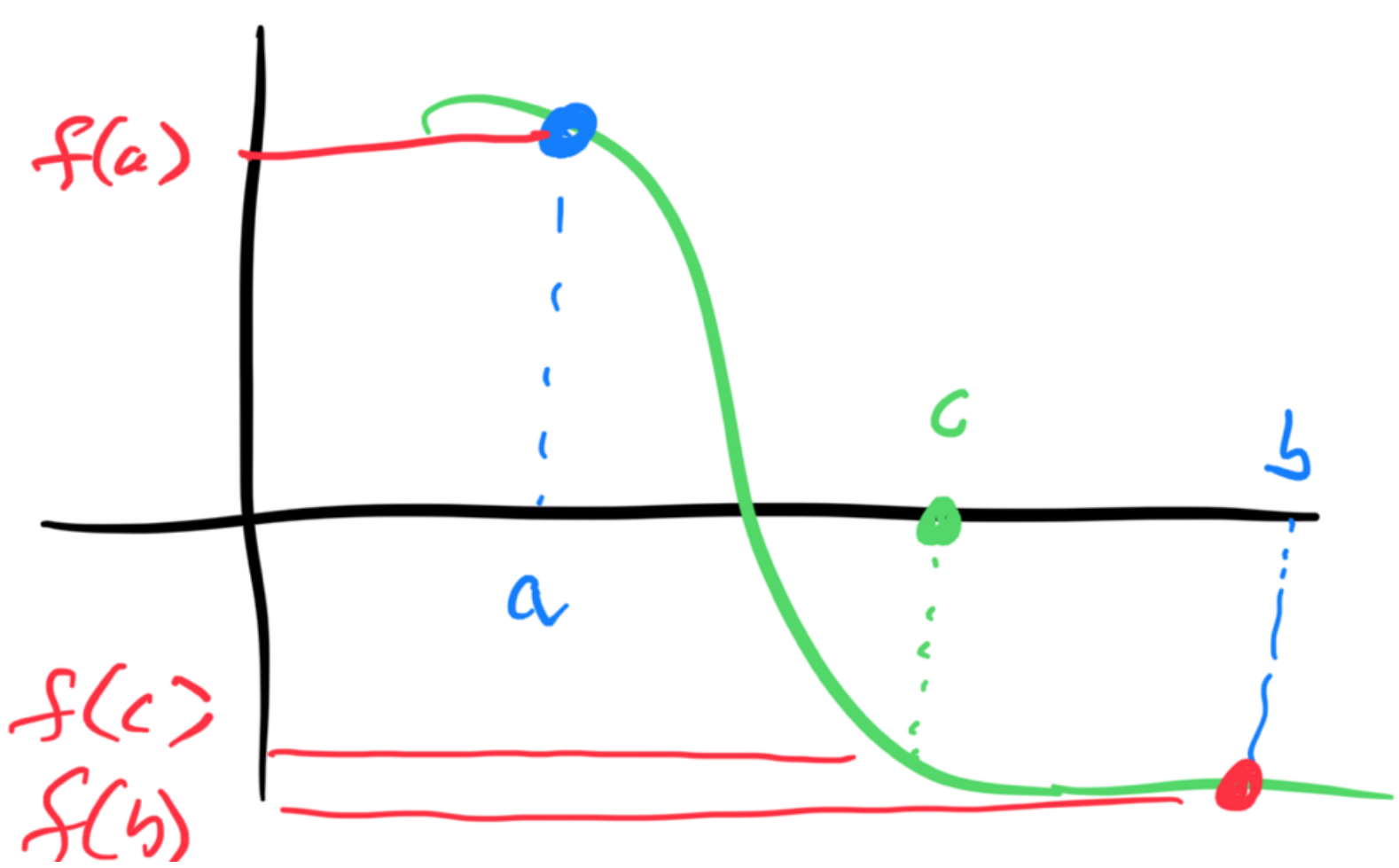
Condition:

$$\boxed{f(x_{low}) \cdot f(x_{high}) < 0}$$

< 0



OK, let's get back to finding the <u>root</u>.



Step 1: Choose the <u>midpoint</u> →

$$C = \left(\frac{a+b}{2}\right)$$

Step 2: If $(f(a) \cdot f(c) < 0)$: (root is in $a \to c$)

$b = c$

Else:

$a = c$

→ In other words, redefine the interval to be the half where the root is!

Step 3: Rinse. Repeat.

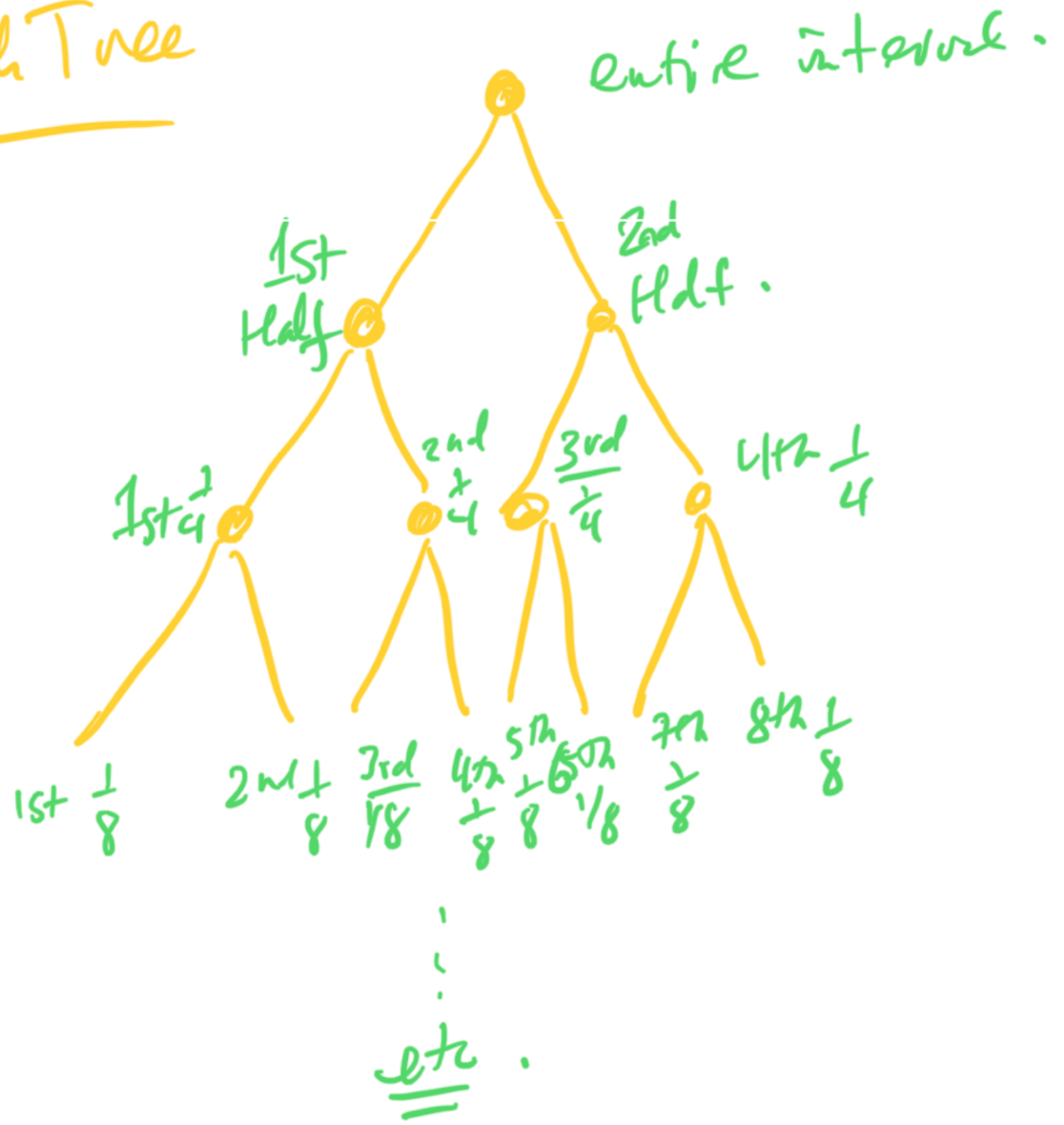
End Condition:

$abs(b-a) <$ some smallish number.

→ i.e. we have narrowed down that a and

things down exponential.
b are really close together.

Binary Search Tree



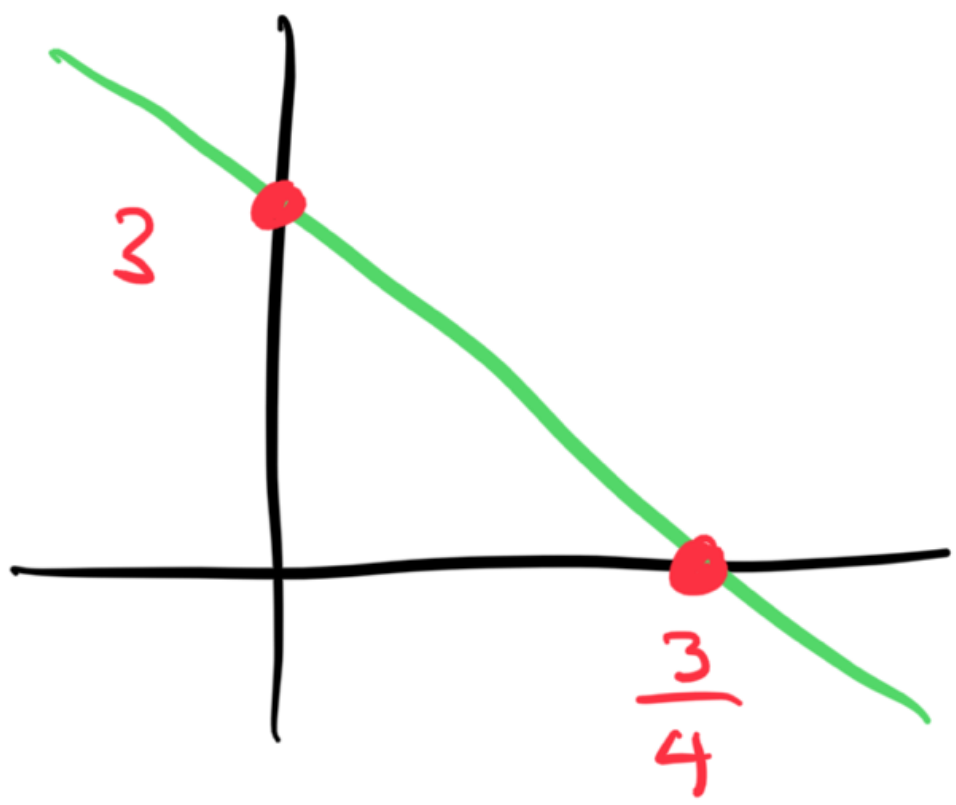
Let's Try to Implement this in Python!

Step 1: Write a "generating function.

```
def f(x):
```

return $-4 * x + 3$

$y = -4x + 3$



So, we know that this function has <u>one</u> root, at $x = 0.75$.

<u>Step 2</u>: Define the starting interval.

$a = 0.0$

$b = 5.0$

$fa = f(a)$

```
fb = f(b)
if (fa * fb > 0):
    print("No root on this interval!!")
elif (fa * fb == 0):
    if (fa == 0):
        print("Found root at x = ", a)
    else:
        print("Found root at x = ", b"
else:
```

→ There's a root and we have to do work now.

Step 3: Write a function to carry out the binary search algorithm.

```
def find_root(a, b):
```

```
def find_root(
    epsilon = 1.0E-5
    fa = f(a)
    fb = f(b)
    if abs(b-a) < epsilon :
        print("found root at x= ",a)
    else:
        c = (a+b)/2.0
        fc = f(c)
        if fc==0.0 :
            print("found root at x=",c)
        else:
            if fa*fc < 0 :
                find_root(a,c)
            else:
                find_root(c,b)
```

Notes:

① Multiple ending conditions !!

Type 1 (i) abs(b-a) < epsilon

Type 2 (ii) fc == 0.0

Type 2

Type 3 a/b (iii) fa or fb = 0.0 at the start.

The binary search tree algorithm is used extensively!! Another example would be searching a list of names (in alphabetical order) for a match.

a → Alice
Charlie
Dave
Frieda
Kallie ← c (midpoint
Moe
Paul
Sarah
b → Xavier

Alice ← a
Charlie
Dave ← c
Frieda
Kallie ← b

Dave is in the.

Found it!!

Search for Dave

Algorithm Efficiency

"Big-O notation"

$O(n)$

$O(n^2)$

$O(\log(n))$

Tells us how the time required scales with the size of the input data set (n)

Example:

Search a list for an element

→ Start at the start and check each element, one by one.

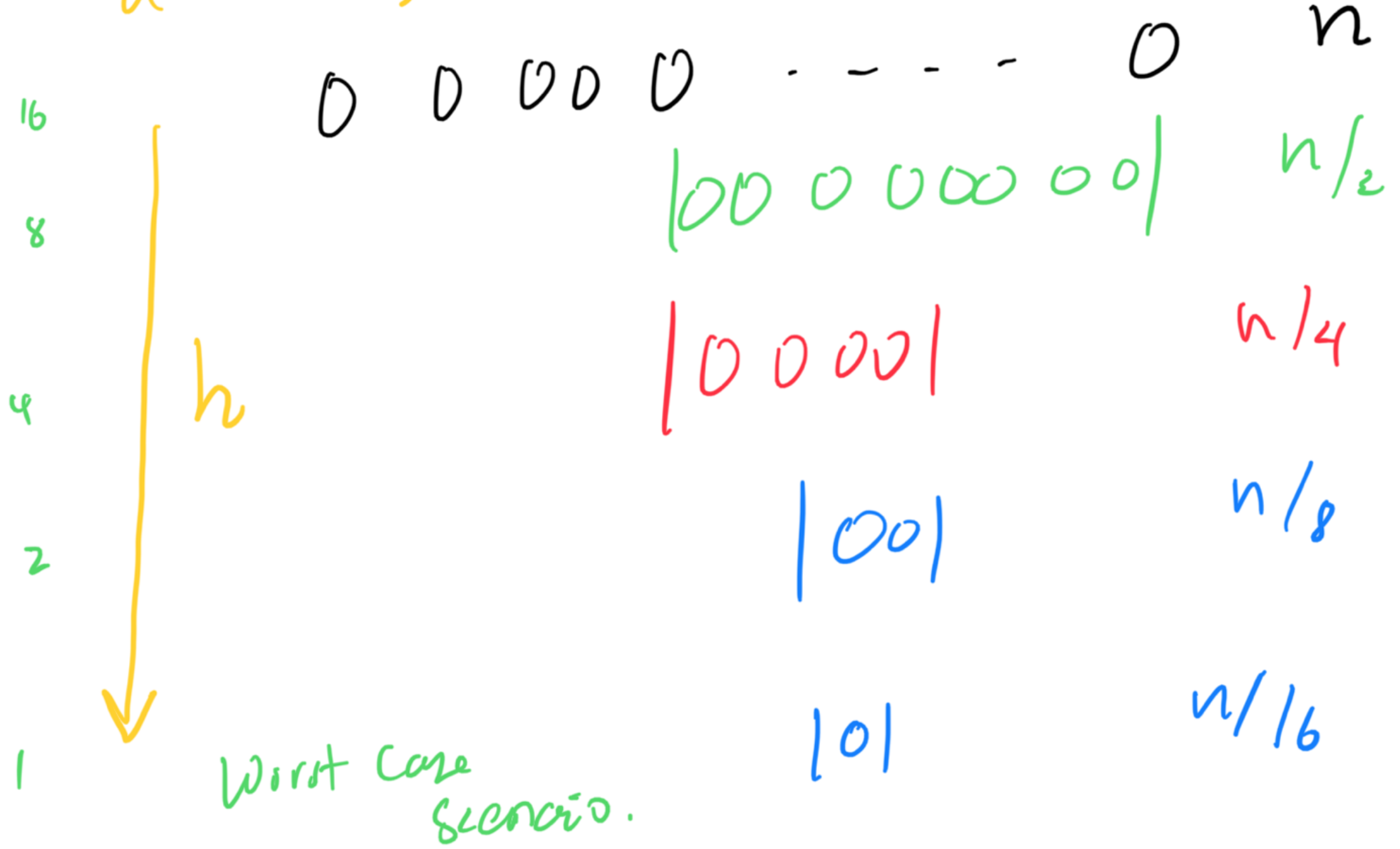
→ On average, this will require $\frac{n}{2}$ checks.

$\therefore$ Complexity is $O(n)$

Because the

→ as n gets bigger the time will scale $\propto n$.

However, let's imagine we use a binary search tree algorithm....



h is known as the height of the binary search tree.

$$n = 2^h$$

$$\log(n) = \log(2^h)$$

$$= h \log(2)$$

$$\boxed{h = \frac{\log(n)}{\log(2)}}$$

$\therefore$ Complexity of BST $\Rightarrow O(\log(n))$

much better !!!!!!!!!

Example: Let's suppose for

$n = 100$ $\quad t = 0.001\,s$

image now $n = 100{,}000{,}000$

$O(n) \rightarrow \therefore t = 1000\,s \approx 17$ minutes

i.e. kinda slow

However, if the algorithm has complexity $O(\log(n))$, then we have.

$\log(100) = 2 \rightarrow t = .001\,s$

$\log(100{,}000{,}000) = 8 \rightarrow \boxed{t = 0.004\,s}$

Are we not entertained ?!?!