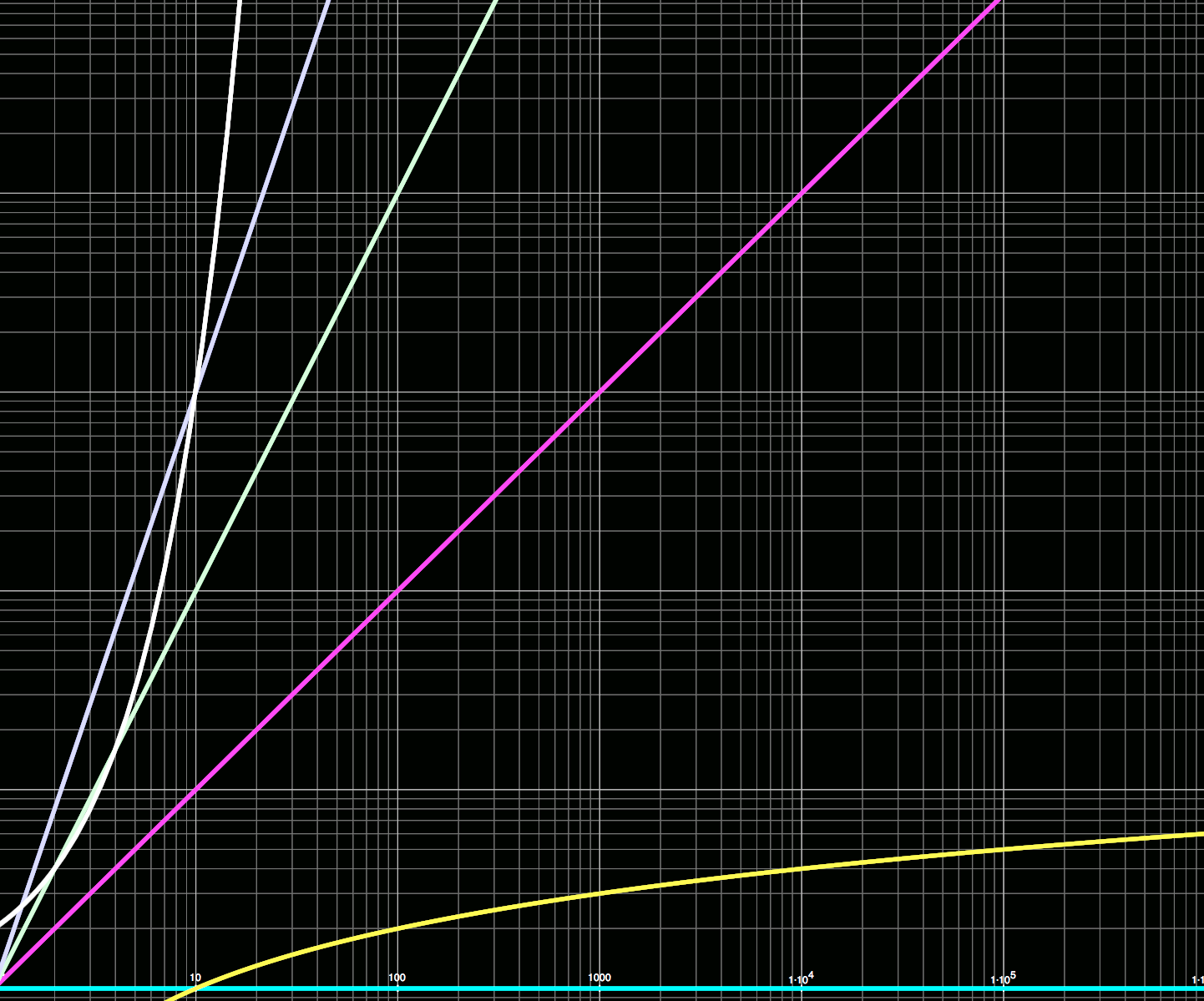
**Orders of Growth:** running time as input grows

Run time

Types of Orders of Growth from smallest to largest

1. Constant
2. 46
3. 1
4. Logarithmic
5. log(x)
6. log()
7. Linear
8. 3x
9. 14x -78
10. Quadratic
11. 3– 4
12. + log(x)
13. Cubic
14. 6
15. 4–xlog(x)
16. Exponential
17. 4+ 5
18. – 14log(x)



Input size

Big Theta (θ):

-Any function that has a similar growth pattern to the subject function.

Ex: Find Big Theta of:

1. 100 ans: ∈ θ (1)

2. 2xlog(x) ans: ∈ θ (xlogx )

3. 3^n -1 ans; ∈ θ (3^n)

4. 3sqrt(x) + logx ans: ∈ θ (x^(1/2))

a. Whenever you have a polynomial and a log together, ignore the log because the polynomial’s growth rate will always be greater

5. log(x) + log() ans: ∈ θ (log(x))

a. You can rewrite log() as 2log(x)

def mystery(L, val, x, y):

if x < y:

return False

z = (x + y) // 2

if L(z) > val:

return mystery(L, val, x, (z – 1))

if L(z) < val:

return mystery(L, val, (z + 1), y

return True

ans: ∈ θ(n)

-This problem splits up a list(L) in half until it finds a particular value(val) using the beginning and ending items (x and y). So if the length of L was 100, the first call would split it to 50, then 25. 12.5 … etc.

**Scheme:**

> (+ (\* 3 5) (- 10 6) )

19

(if <predicate> <consequent> <alternative>)

Difference between “cond” and “if”:

***“if” takes in one predicate and has one consequent and one alternative***

(if (= 3 4) 3 6)

6

***“cond” can have multiple predicates and consequences, but needs an “else” statement to have an alternative.***

(cond ((= 8 9) 4)

((= 9 10) 5)

(else (+ 4 5)))

9

***#t means True and #f means False in scheme***

> (<= 2 1)

#f

(define (<name> <formal parameters>) <body>)

(lambda (<formal-parameters>) <body>)

“If you did project and understand the scheme part of it, you will be fine on the final”

-GSI that taught this section

**Scheme Lists and Parsing:**

> (cons 1

(cons 2

(cons 3

(cons 4 nil))))

(1 2 3 4)

> (list 1 2 3 4)

(1 2 3 4)

> (define x (cons 1 2))

> x

(1 . 2)

> (car x)

1

> (cdr x)

2

***Parsing: Taking scheme list (1 2 3) and making it into P(1, P(2, P(3, nil)))***

.

>’(1. (2. ((4. (3. (2. ()))). ((2. (4. 2)). (3. ())))))

(1 2 (4 3 2)(2 4. 2) 3)

>’(1 2 3. ((4. 5). (5 6. 7)))

(1 2 3 (4. 5) 5 6. 7)

;;deep-map ‘(1 2 (3 4) 5)

(lambda (x) (\* x x)))

(1 4 (9 16) 25)

(define (deep-map x f)

(cond ((null? x) ())

((list? (car x))(cons(deep-map (car x) f)(deep-map(cdr x) f)))

(else (cons (f (car x))

(deep-map(cdr x) f)))

Count how many eval and applies there are

(define (f x) (\* x x))

(f, (+, 1 (if (= 1 1) 2 1)))

eval:16

apply: 4

**Tail Calls:**

Tail recursion is just placing your recursive call in the return statement.

Not a tail call:

def f(x):

if x == 0:

return 1

f(x-1)

return 2

Tail call:

π= (4/1) – (4/3) + (4/5) – (4/7)…

(define (calc-pi n)

define (help i sum s)

(if (= i n) sum

(help (+ i 1) (+ sum (\* s (/ 4 (+ (\* i 2) 1)))) (\* s -1))

help (0 0 1))

**Map Reduce and Parallelism:**

Parallelism:

What are the possible outputs of the following threads run in parallel?

x=0

Thread 1:

1. Acquire x
2. x = x + 1
3. Write x

Thread 2:

1. Acquire x

1. x = x + 2
2. Write x

Answer: Possible outputs are x= 1, 2, and 3

You can mix and match steps 1 and 2 for both threads, but only 1 write can be computed at a time. For x=1 just run through thread 1 and terminate, x=2 run through thread 2 and terminate, x=3 run through steps 1 and 2 on thread 1 then switch to thread 2 and do steps 1-3 and terminate.

Locks:

Parallel for loop 0<i<10:

acquire\_lock(x)

x + = 1 (this line is critical , only 1 thread can do this at a time)

release\_lock(x)

Map Reduce: (k,v)= (key,value)

Sort

Map

Reduce

k,v k,v k,v

k,v

k,v

k,v k,v k,v

(emit)

k,v k,v k,v

k,v

***The map part of this has a special characteristic called “emit”, where it can choose whether or not to map a function onto the new list. In this way Map Reduce can also be used as a filter.***