# Lect 6:

## Time analysis

A int list is either

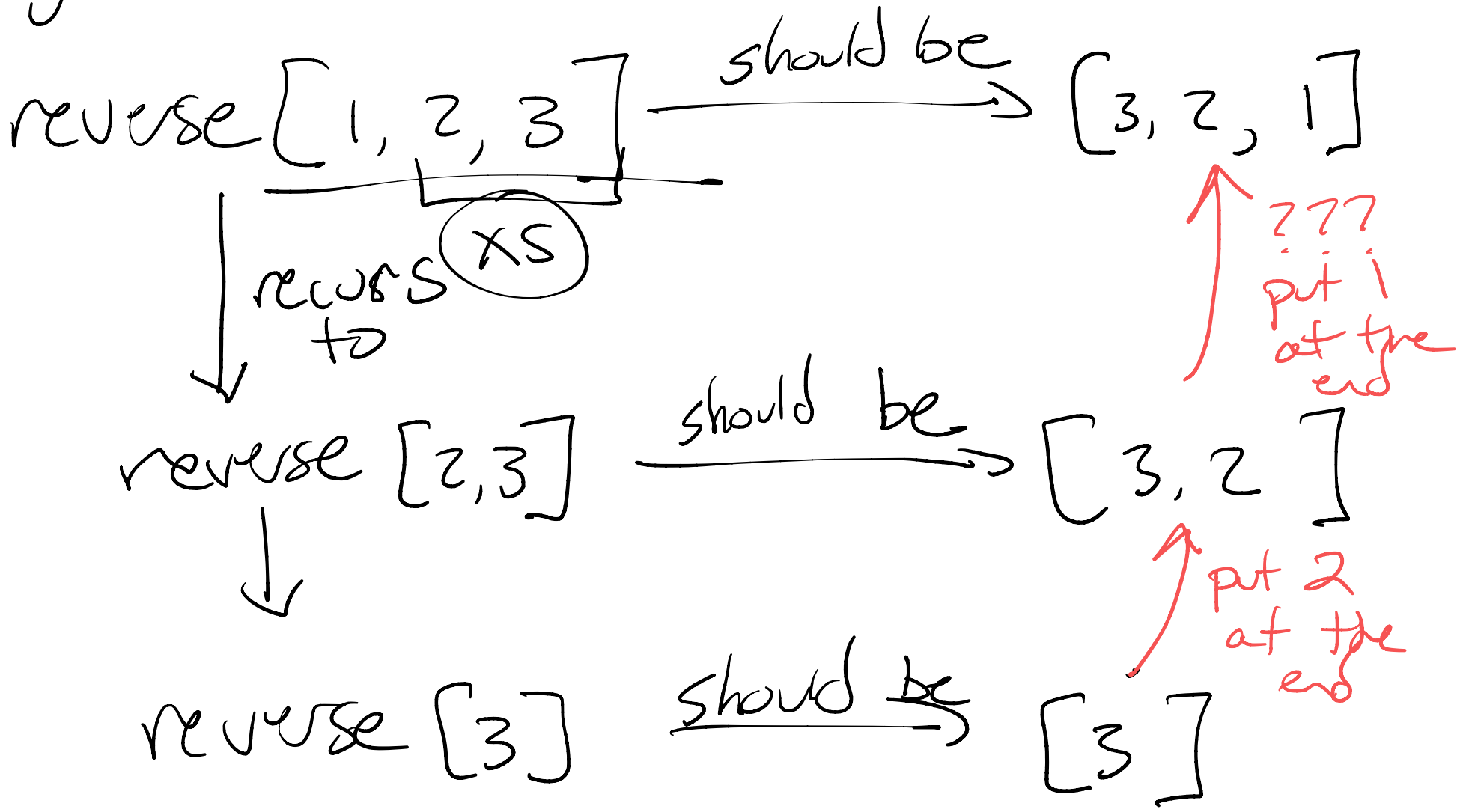
- `[]`, or
- `x::xs`, where `x : int` and `xs : int list`

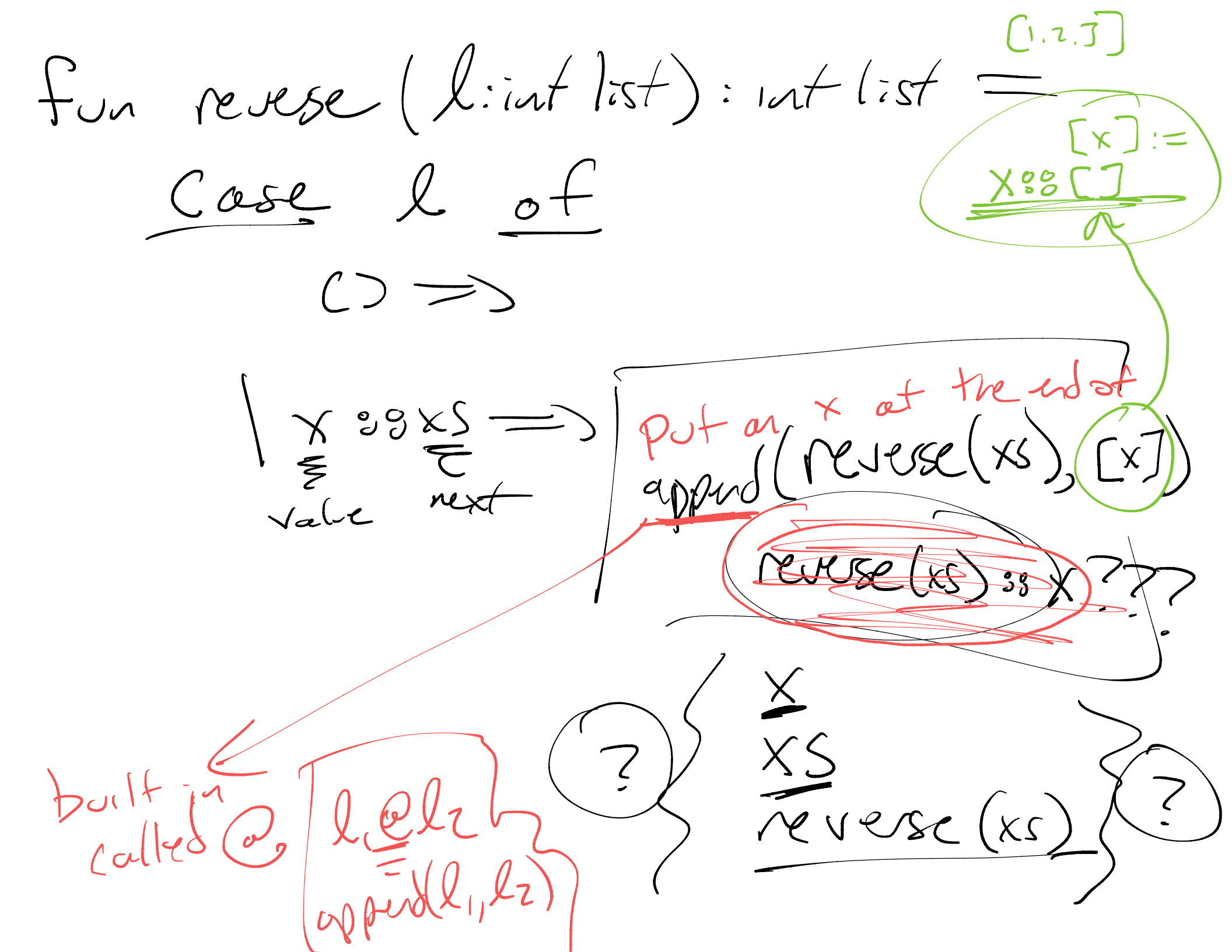
→ and that's it!

# reverse a list

"induction on an example"

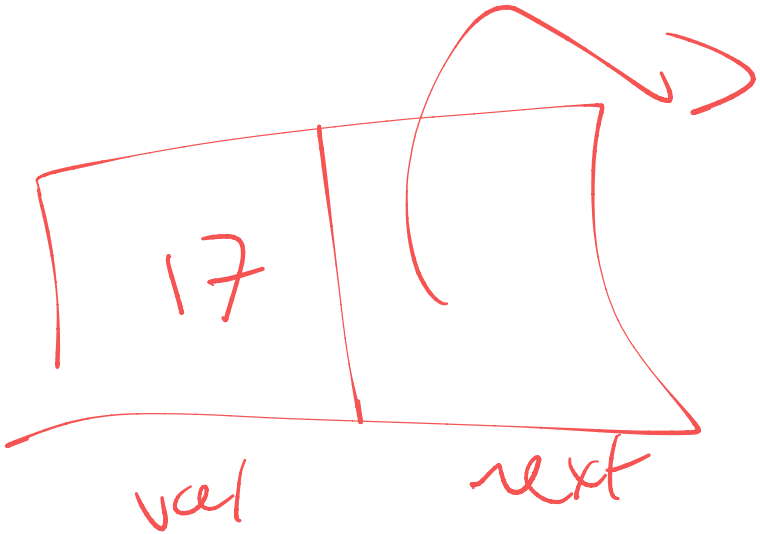
e.g.




fun reverse (l:int list) : int list =
case l of
[] =>
[1,2,3]
[x] :=
x::[]
| x::xs =>
value
next
put an x at the end of
append(reverse(xs), [x])
reverse(xs) :: x ???
built in
called @
l1@l2
append(l1,l2)
?
x
xs
reverse(xs)
?

reverse (xs) :: x
: int list
int

x :: xs
int
int list



```
fun putAtTheEnd(l:int list, x:int):int list =
    case l of
        [] => [x]
      | y::ys => y::putAtTheEnd(ys,x)
```

```
fun append(l1:int list, l2:int list):int list =
```

e.g. append([1,2,3],[4,5,6]) = [1,2,3,4,5,6]

append([3,7],[9,4]) = [3,7,9,4]

Thm

putAtTheEnd(l,x) = append(l,[x])

```
fun append(l1:int list, l2:int list):int list =
  case l1 of
      [] => l2
    | x1::xs1 => x1::append(xs1, l2)
```

e.g. append([], [1,2,3]) = [1,2,3]

append([1,2,3], []) = [1,2,3]

append([1,2,3], [4,5]) = [1,2,3,4,5]

append([2,3], [4,5]) = [2,3,4,5]

```
fun append(l1:int list, l2:int list):int list =
  case l1 of
    [] => l2
  | x1::xs1 => x1::append(xs1, l2)
```

append( [1,2,3], [4,5] )

1. case [1,2,3] of ...
2. 1 :: append([2,3], [4,5])
3. case [2,3] of ...
4. 1 :: 2 :: append([3], [4,5])
5. case [3] of ...
6. 1 :: 2 :: 3 :: append([], [4,5])
7. case [] of ...
8. 1 :: 2 :: 3 :: [4,5]

[1,2,3,4,5]

length
size of $l_1$ is 3
length of $l_2$ is 2

steps: 8

```
val x = [1,2,3]
val y = [4,5]
val z = append([1,2,3],[4,5])
```

[1,2,3,4,5]

```
fun reverse (l:int list) : int list =
  case l of
    [] => []
  | x::xs => append(reverse(xs), [x])
```
x = value, xs = next

Put an x at the end of append(reverse(xs), [x])

reverse([1,2,3])

↦ append(reverse[2,3], [1])  (1 = [2,3], 2 = [1])

↦ append(append(reverse[3], [2]), [1])

↦ append(append([3], [2]), [1])

↦ append([3,2], [1])

↦ [3,2,1]

Analyzing the running time of functions (work)

↳ predict how many steps a function takes in terms of the size of the input.

append([1,2,3],[4,5])

```
fun append(l1:int list, l2:int list):int list =
  case l1 of
    [] => l2
  | x1 :: xs1 => x1 :: append(xs1, l2)
```

input

what is the size?

idea: length of $l_1$ + length of $l_2$

A function mapping sizes to # of steps the function takes

Math

$W_{append}(n_1, n_2) =$ ------>

work, length of $l_1$, length of $l_2$

e.g. $W_{append}(3,2) = 8$

append( [], $l_2$ )

① ⇒ case [] of [] ⇒ $l_2$ | x::xs ⇒ x::append(xs, $l_2$)

② ⇒ $l_2$

$W_{append}(0, n_2) = 2$

(A)

append( x::xs, $l_2$ )

1 → case x::xs of [] ⇒ $l_2$ | x::xs ⇒ x::append(xs, $l_2$)

2 → x :: append(xs, $l_2$)

→ x :: ———

→ x :: ———

→ x ??? ———

→ x ??? ———

→ x ??? VS

$W_{append}(n_1, n_2) = 2 + W_{append}(n_1 - 1, n_2)$ [if $n_1 \neq 0$]

B

"Recurrence" recursive function from sizes to #'s

$$Wappend(\overbrace{0}^{\text{length } l_1}, \overbrace{n_2}^{\text{length } l_2}) = 2$$

$$Wappend(n_1, n_2) = 2 + Wappend(n_1 - 1)$$

---

$$Wappend(\overbrace{n}^{\text{length of } l_1}) = \begin{cases} 2 \text{ if } n = 0 \\ 2 + Wappend(n-1) \text{ o.w.} \end{cases}$$

Recurrence

$$Wapped(n) = \begin{cases} 2 \text{ if } n = 0 \\ 2 + Wapped(n-1) \text{ o.w.} \end{cases}$$

($n$ = length of $l_1$)

$$Wapped(n) = 2n + 2$$

Closed form: non-recursive solution to those equations

$$\underbrace{2 + 2 + 2 + 2 + \ldots + 2}_{n+1 \text{ times}} = 2n + 2$$

# Analyzing a function

Code → (1) → Recurrence → (2) → Closed form → (3) → big-O

$W_{append}(n) = 2n + 2$

small input

$W_{append}(n) = 2n + 3$

"constant factors"

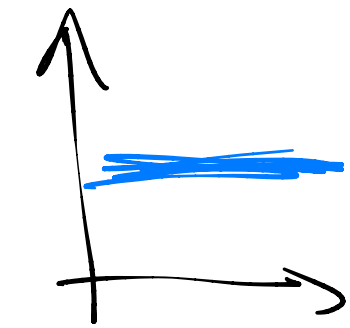
$W_{append}(n) = 3n + 2$

```
fun append(l1, l2) =
  case l1 of
    [] => l2
  | x::xs => let val zs = append(xs, l2)
             in x::zs end
```

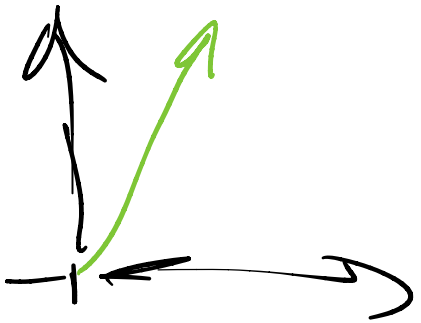
# big-O notation:

Summarize the "most important" information about the closed form

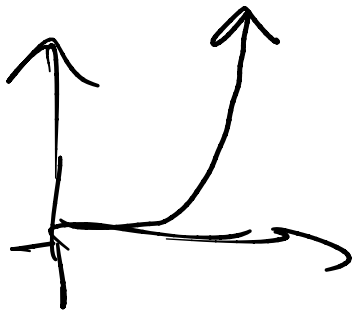
$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3)$$



constant



linear



quadratic

$W_{apples}(n) = 2n + 2$ is $O(n)$

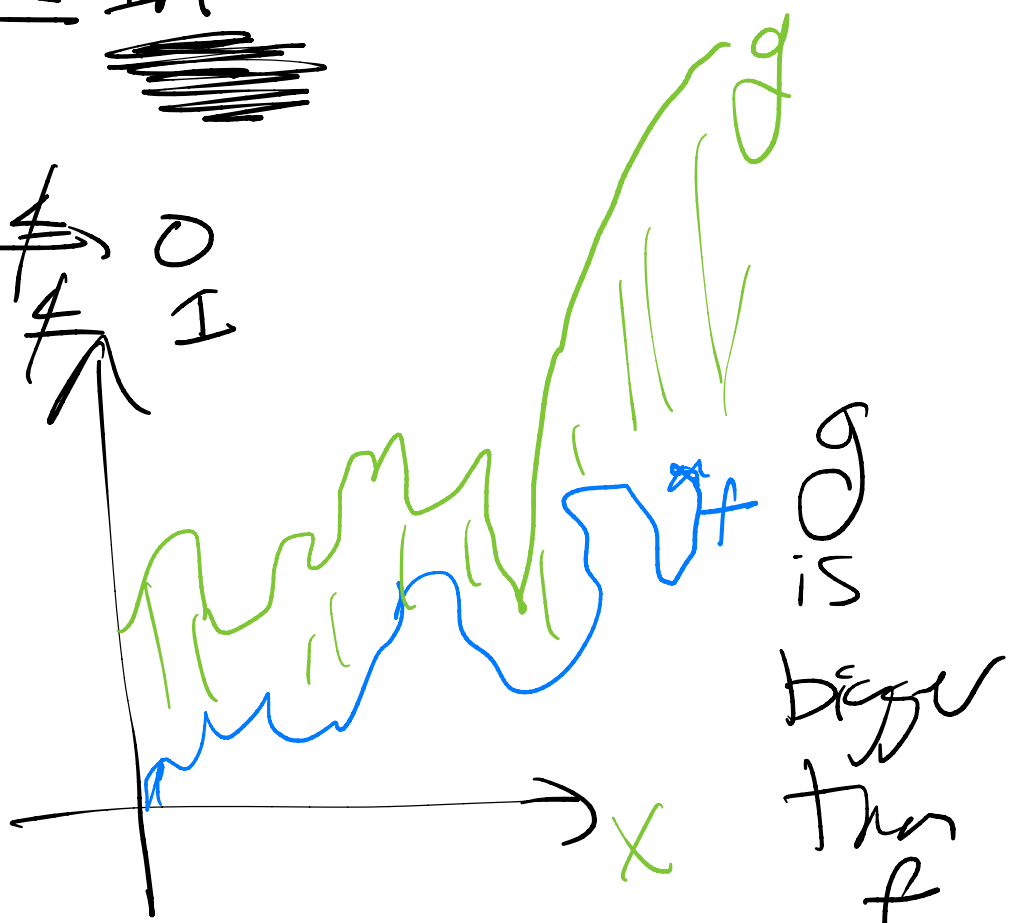
"linear time"

$2n + 2 \stackrel{?}{\leq} 1n$

e.g.

$n=0$ $2 \not\leq 0$

$n=1$ $4 \not\leq 1$




g is bigger than f

First try:

f(n) is O(g(n)) means

for all x,

$f(x) \leq g(x)$

$W_{append}(n) = 2n+2$ is $O(1)$

there exists a $k$ with

$2n+2 \leq k \cdot n$

e.g. $k = 100$

$2n + 2 \leq 100n$

| | | |
|---|---|---|
| $n=1$ | $4$ | $\leq 400$ |
| $n=2$ | $6$ | $\leq 600$ |
| $n=3$ | $8$ | $\leq 800$ |

$k = 4$

$2n+2 \leq 4n$

$4 \leq 4 \cdot 1$

$6 \leq 4 \cdot 2$

$8 \leq 4 \cdot 3$

⋮

**Def 2** (attempt)

$f(n)$ is $O(g(n))$

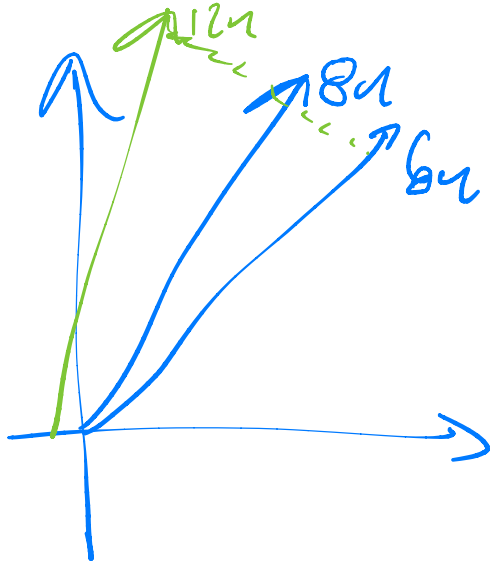
iff

there exists a $k$ such that (not dependent on $x$)

for all $x$,

$f(x) \leq k\, g(x)$

E.g. can show that




$6n$ is $O(8n)$

$8n$ is $O(6n)$

$O(kn)$ for any constant $k$ are the same

$O(n)$

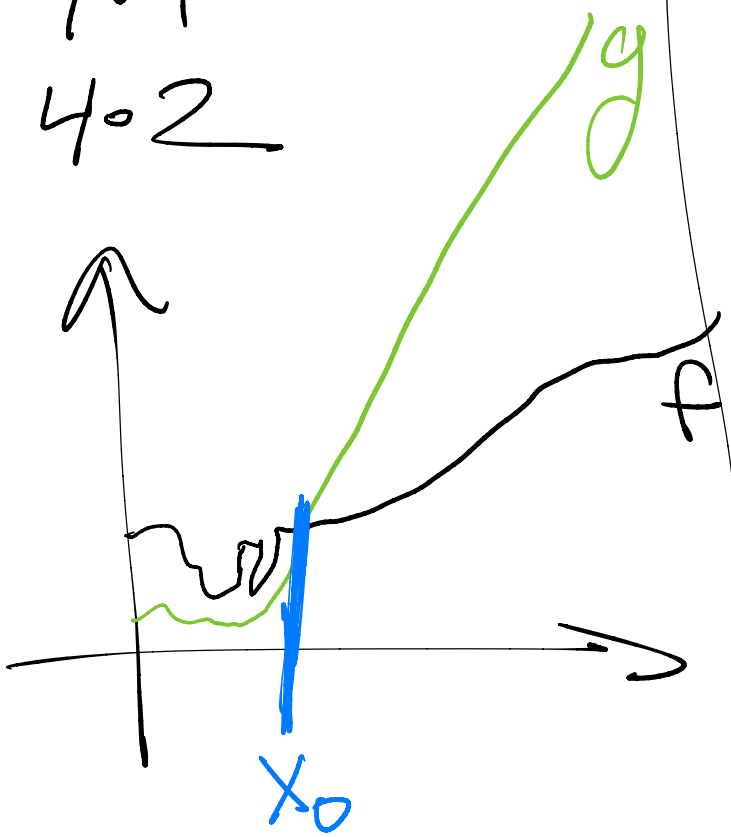
$W_{append}(n) = 2n+2$ is $O(1)$

e.g. $k = 4$

$x_0 = 1$

$2 \cdot 1 + 2 \leq 4 \cdot 1$

$2 \cdot 2 + 2 \leq 4 \cdot 2$




**Def 3** (final)

$f(n)$ is $O(g(n))$

iff

there exists a $k$ and $x_0$ such that for all $x \geq x_0$

$f(x) \leq k\,g(x)$

# Analyze reverse

$W_{reverse}(n) =$ recurrence

$=$ closed form

$=$ big O