Lecture 8

Time Complexity

Define efficiency

Time

Memory

(* 2 (if (> x 5) (- x 1) (+ x 1)))

Efficiency is measured in seconds.

Your code make different amount of time in different machines. (Machine dependent)

Number of "operations"

(* 2 (if (> x 5) (- x 1) (+ x 1)))

(> x 5) ⇒ 1 operation

(- x 1) ⇒ 1 operation
(if) ⇒ 1 operation
(* 2 output) ⇒ 1 operation

Any thing involving atomic data types - integers, booleans, 'symbols, reals.

Not atmocic - list, structures

Example: Sorting

Suppose you have a list of integers you wont to sort that list.

Suppose you have a function

1. (sort (list 8 1 5 6))

2. (sort (list 9 1))

(sort (list 8 1 5 6))

=> (list 5 6 8)

(sort (list 8 1))
⇒ (list 1 8)

case 1:

Is number of operations in case 1 is same number of operations is case 2.

Number of operations should depend on the input.

First working definition

Time complexity of a "function" / program is a function (racket)

↑ math sense

which tells me the number of operations which my "function" will take given a certain input.

Time complexity of sort is T

T((list 8 5 6 100))
= 10 operations.

We want our time complexity to depend on the size of the input.

(sort (list 10 9 7 6 4))

→ 100 operations

(sort (list 1 3 5 6 7))

→ 10 operations.

Both inputs have same size.
= 5.

Suppose time complexity of sort is T

What is value of T at 5.
T(5) := How many

operations sort will take to sort a list of size 5.

We take the worst-possible value.

In our example
T(5) is at least 100 operations.

Definition: The worst-case time complexity of a "function" / program / algorithm is a function T which tells the maximum

number of operation which your program can take on a input of size s.

$T(s) \triangleq$ Max number of operations which the program can take on on any input of size s.

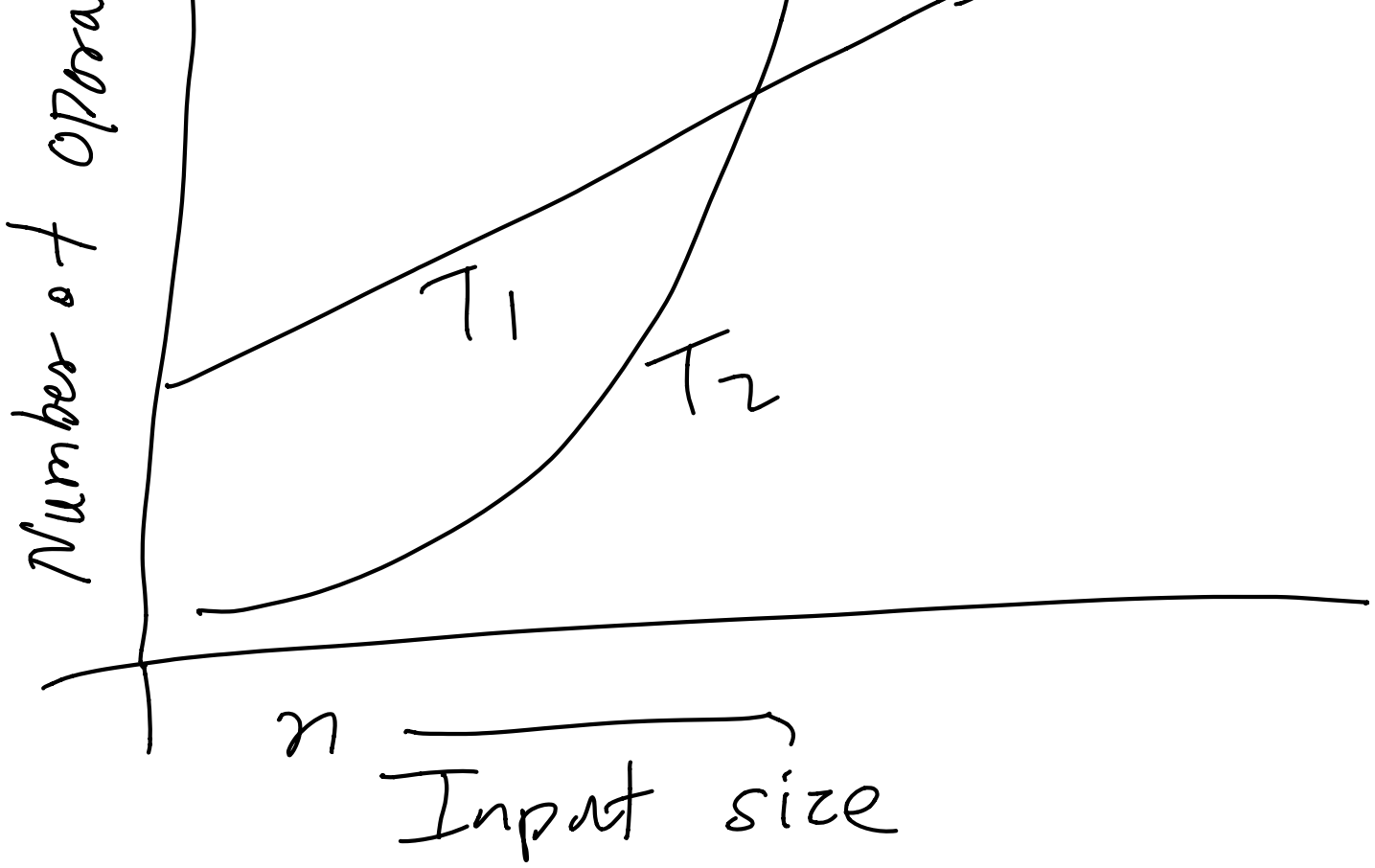
Give a racket function how do you $T(n)$?

1. We can also never compute $T(n)$ precisely,

2. We don't have to compute $T(n)$ precisely.

Rather we are interested in its asymptotic behaviour.

How $T(n)$ scales an n increases.



$T_1(n)$ is time complexity of sort function 1

$T_2(n)$ is time complexity of sort function 2

Asymptotically T1 performs

better than T2.

Big-Oh notation

Formal definition

Suppose $g(n)$ and $f(n)$

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

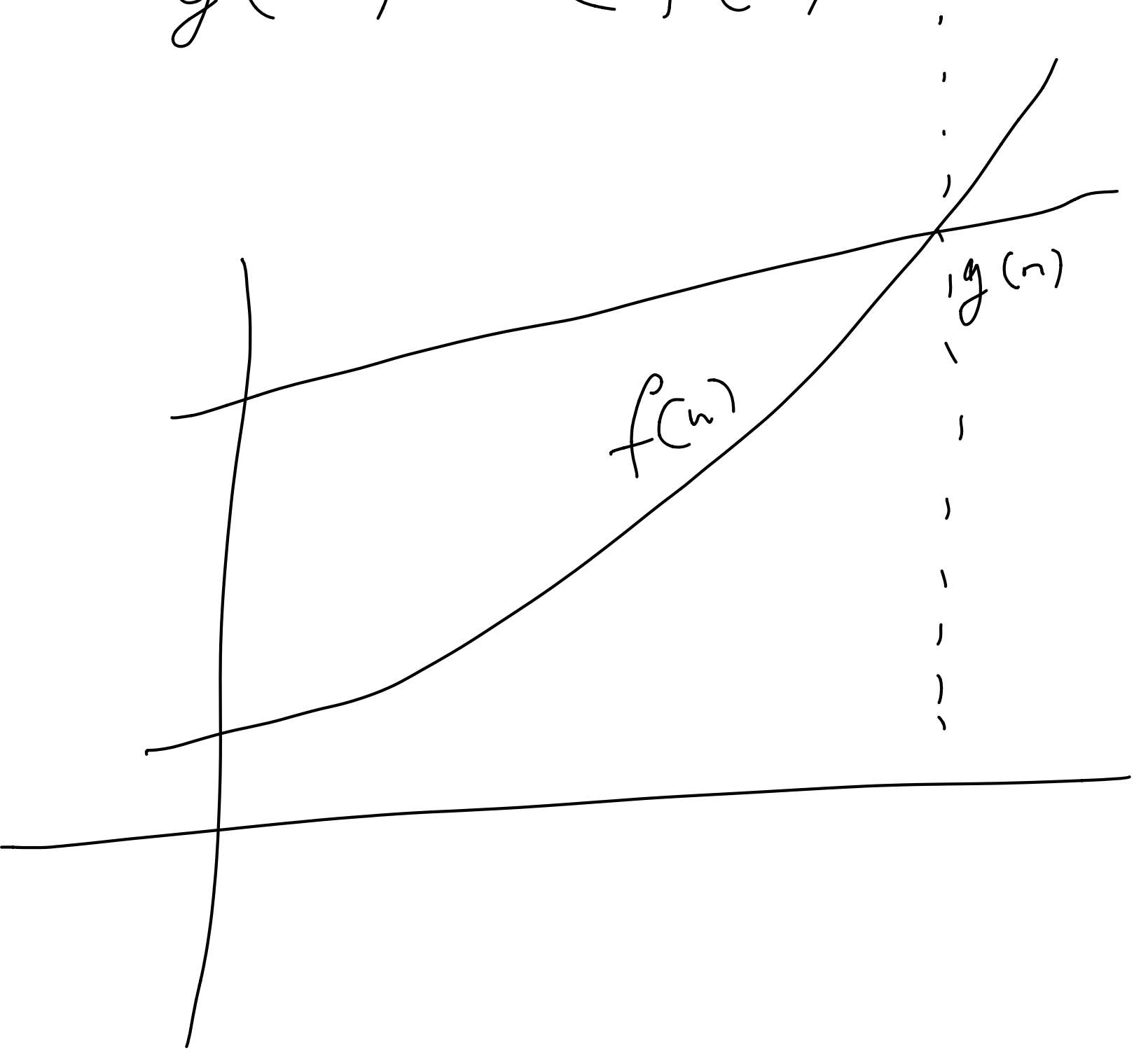
($g(n) \in O(f(n))$, $g(n) = O(f(n))$)

if there exists $M \in \mathbb{N}$, $C > 0$

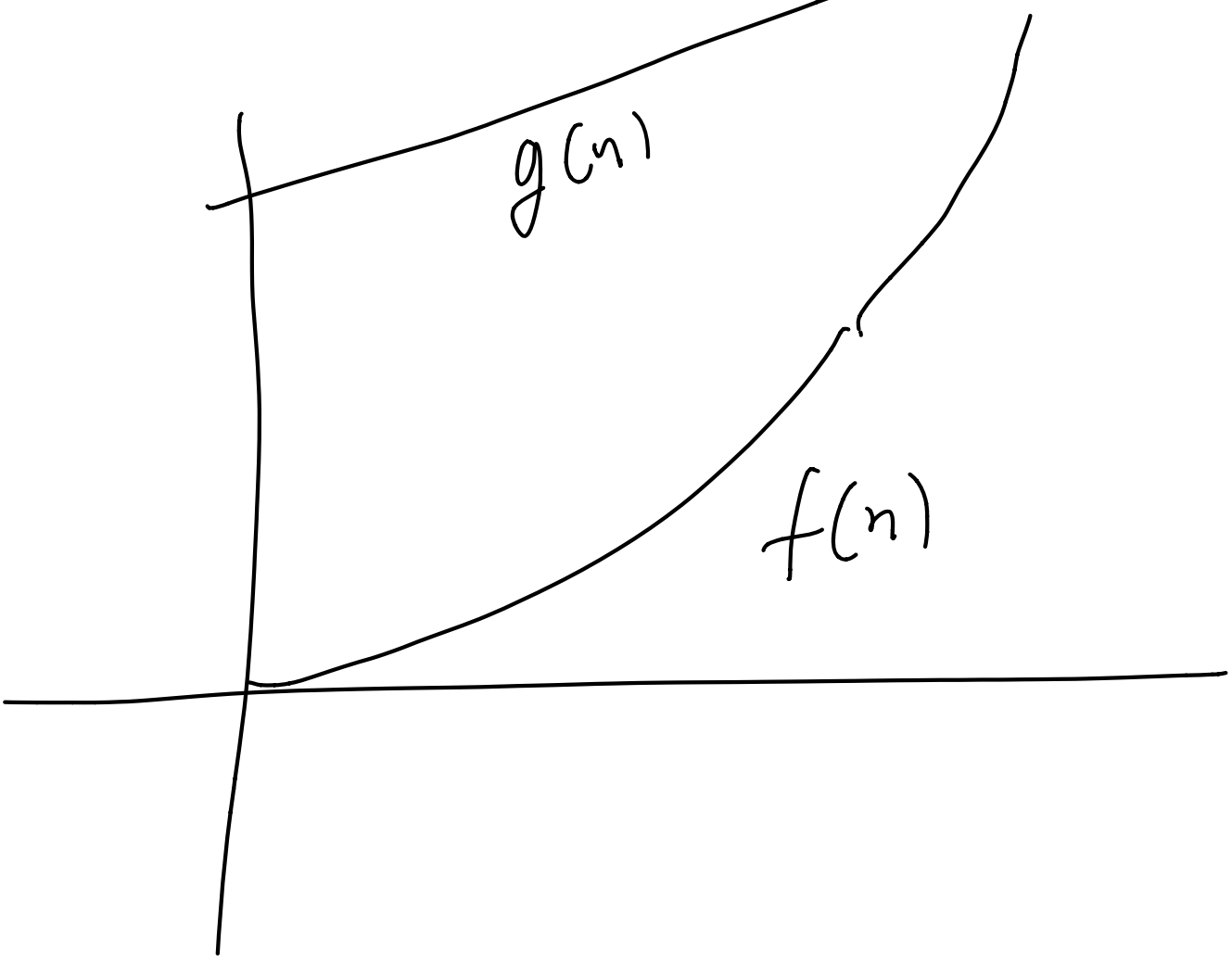
such that for every $n \geq M$

$a \in A$ (a belongs to A set ↑)

$g(n) \leq C f(n)$



Basically $g(n)$ grows at a slower or same speed as $f(n)$.



Wherever we are computing

T(n) : worst case time complexity

T(n) is $O(n^2)$

Example

$5n \in O(n)$

$Cn \in O(n)$

$n^2 \notin O(n^2)$

$6n^3 \in O(n^3)$

$n^3 + n^2 + 1 \notin O(n^3)$

↑ Dominating term

$5n^3 + n^2 + 1 \notin O(n^3)$

↑ Dominating term

dominating term

$n + \log n \in O(n)$

n grows faster than $\log n$.

Example

Step 1:

What is the time complexity

What is the input.
List of Numbers

Suppose your input is a list
, what is the input size?

My input size : number
of elements in the list.

Step 2:
If your function is recursive
you have to write the
recurrence relation

Step 3:
Solve recurrence relation.

Suppose $T(n)$: denotes time complexity of the length function.

```
(: length : (Listof Number)
             → Number)

(define (length my-list)
  (cond ← step 1 1 operation
step 2 → [(empty? my-list) 0]
         [else (+ 1 (length (rest my-list)))]))
```

step 3

$T(n) = 1 + 1 + 1$

↑ input size

↑ rest

$T(n) = 1$ ← cond

$+ 1$ ← empty?

$+ 1$ ← (rest my-list)

$+ T(n-1)$

$+ 1$ ← (adding 1)

$T(n) = 4 + T(n-1)$

↓

Recurren Relation

Base case

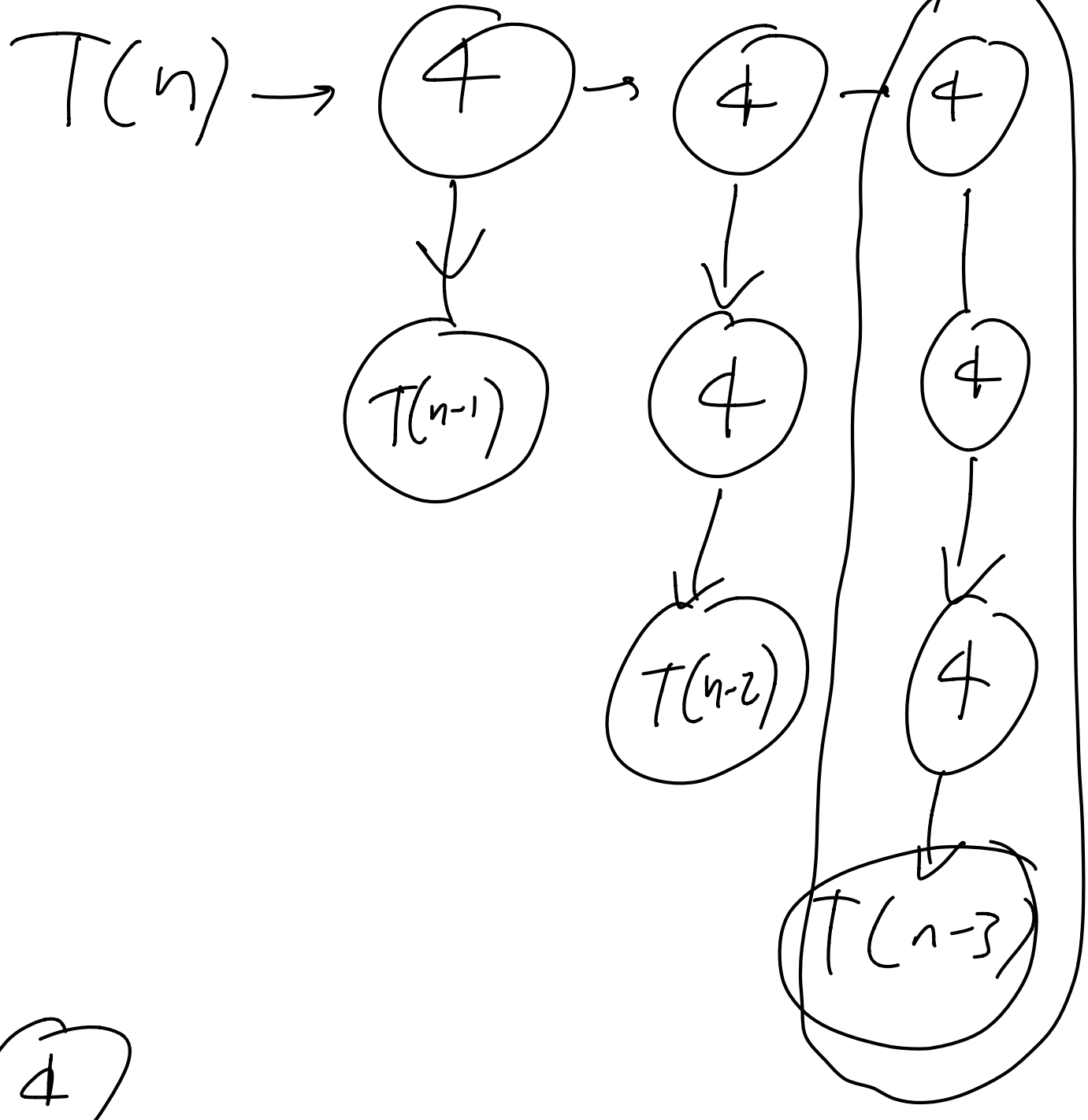
Input size is 1 or 0.

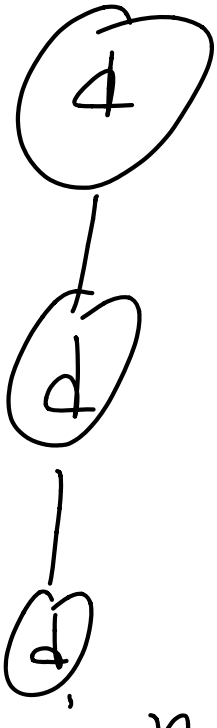
If $n = 0$

$T(0) = 2$

$$T(n) = T(n-1) + 4$$

$$T(0) = 2$$





: n- nodes.

Height of this tree will be n.

4n.

$$T(n) = 4n = O(n)$$

Time complexity of this function is $O(n)$.

$$T(n) = 4 + T(n-1)$$
$$= 4 + \underbrace{4 + T(n-2)}_{T(n-1)}$$

$= 4 + 4 + 4 + T(n-3)$

$= \underbrace{4 + 4 + 4 \ldots \ldots}_{n \text{ times}}$

$= 4n$

$T(n) = 4n \in O(n)$

Remark:

$T(n) = T(n-1) + 5.$

$T(0) = 2$

$$T(n) = S + T(n-1)$$
$$= S + S + T(n-2)$$
$$= S + S + S + T(n-3)$$
$$= \underbrace{S + S}_{n\text{-times}}$$
$$= Sn.$$

$$T(n) = Sn \in \mathcal{O}(n)$$

Moral of the story:

Don't worry about the about the constants

Remark

$$T(n) = C + T(n-1)$$

$$T(0) = K$$

C, K - constants.

$$T(n) = C(n-1) + K \in O(n)$$

Solution $T(n) \in O(n)$

$[T(n) = O(n)]$ ✓

Example

$$T(n) = 2T(n-1) + C$$

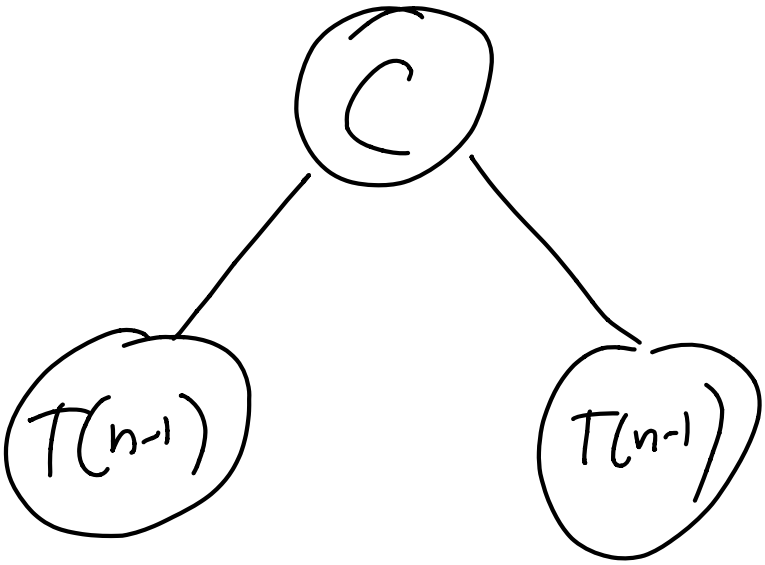
This is very importa.

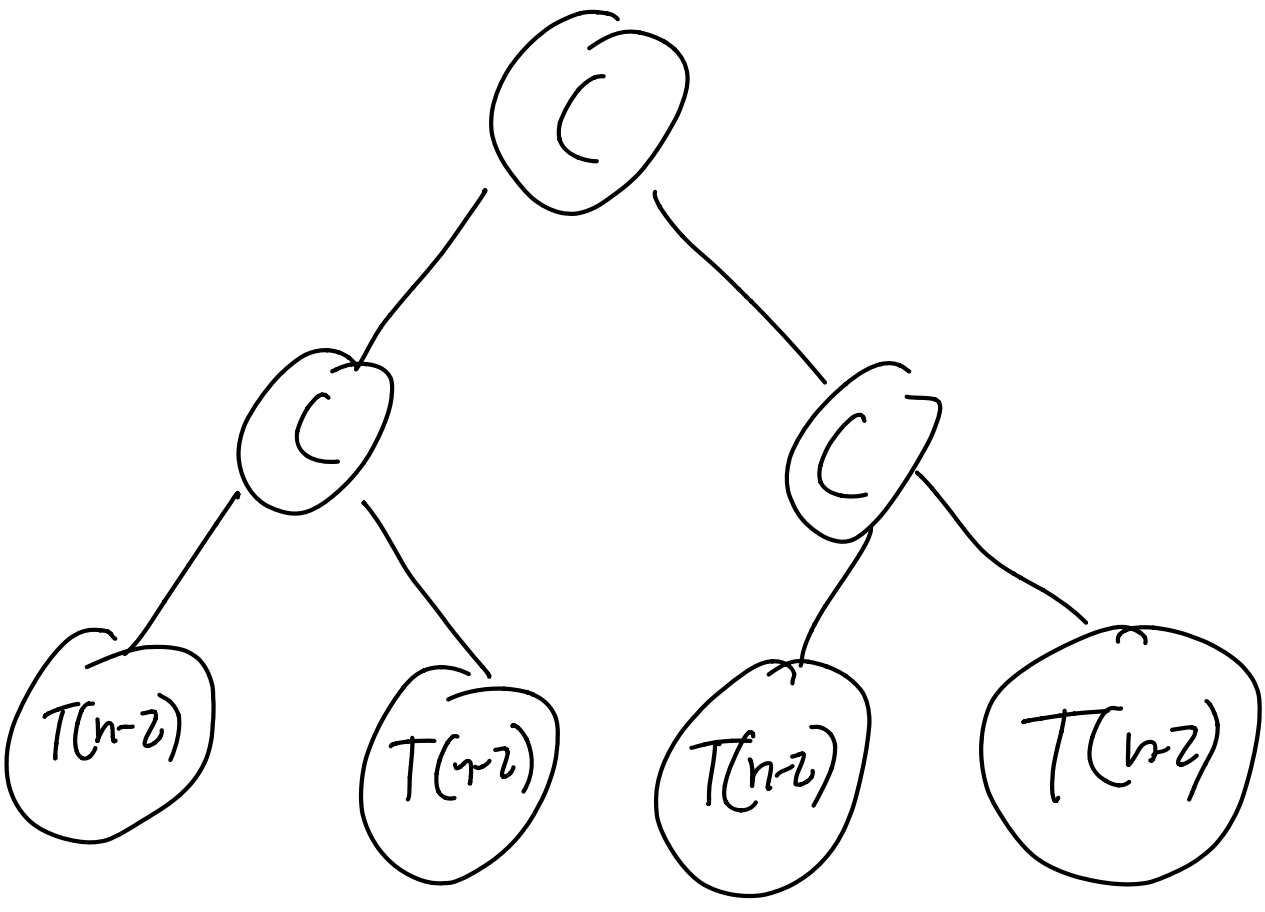
The actual valve of the constant is not importa

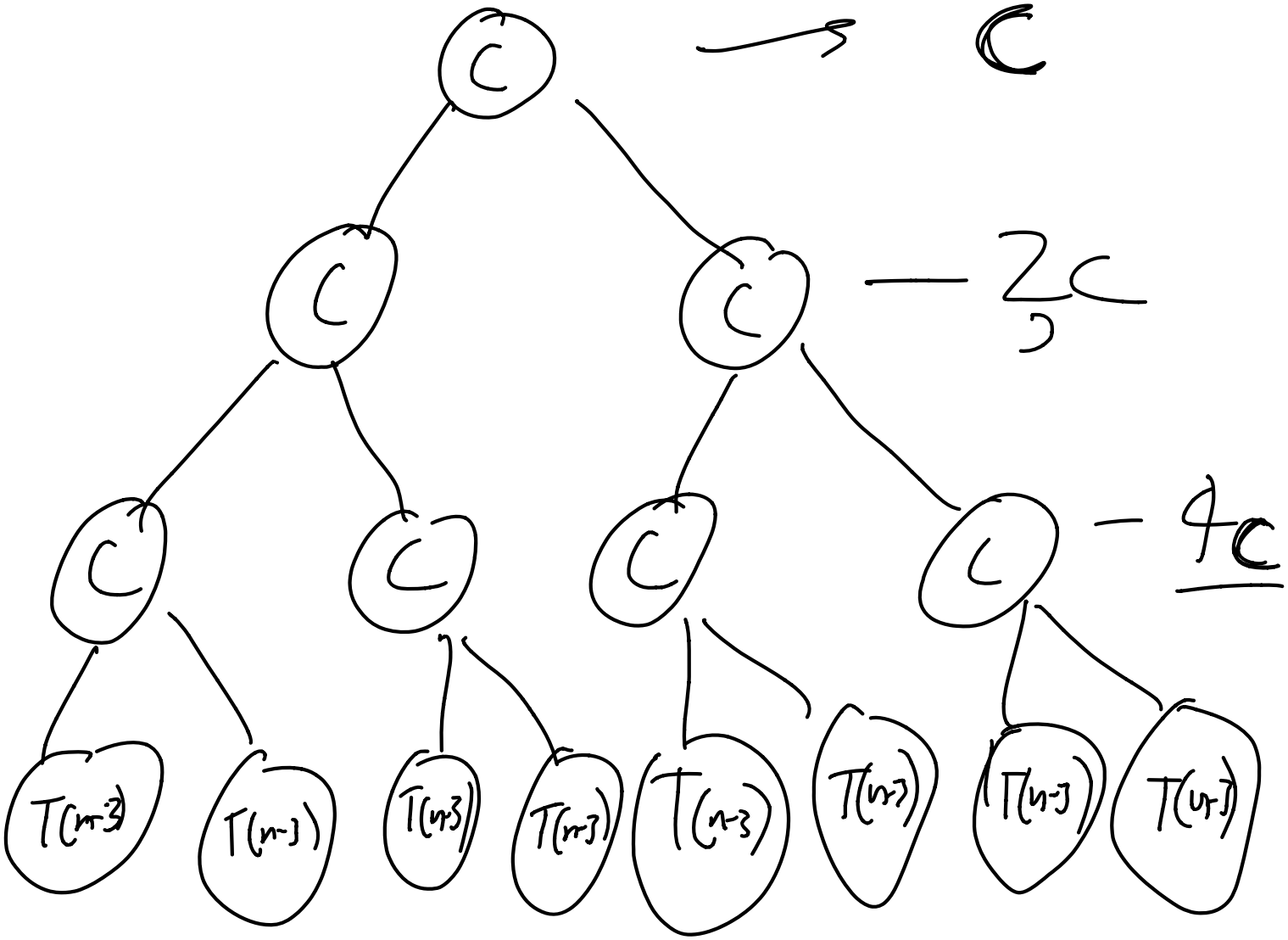
$$T(0) = K$$

The actual value of the constant is not importa.

Tree method for solving recurrence relations







C → C

$2C$

$4C$

$T(0)$ $T(0)$ $T(0)$

Height of the bee will n.

$$T(n) = c + 2c + 3c$$

$$+ 4c + 5c + 6c$$

$$\cdots \quad nc$$

$$= c + 2c + 4c + 8c \cdots$$

$$+ 16c + 32c + 2^n c$$

$$= c(1 + 2 + 4 + 8 + 16 + 32 \cdots 2^{n-1})$$

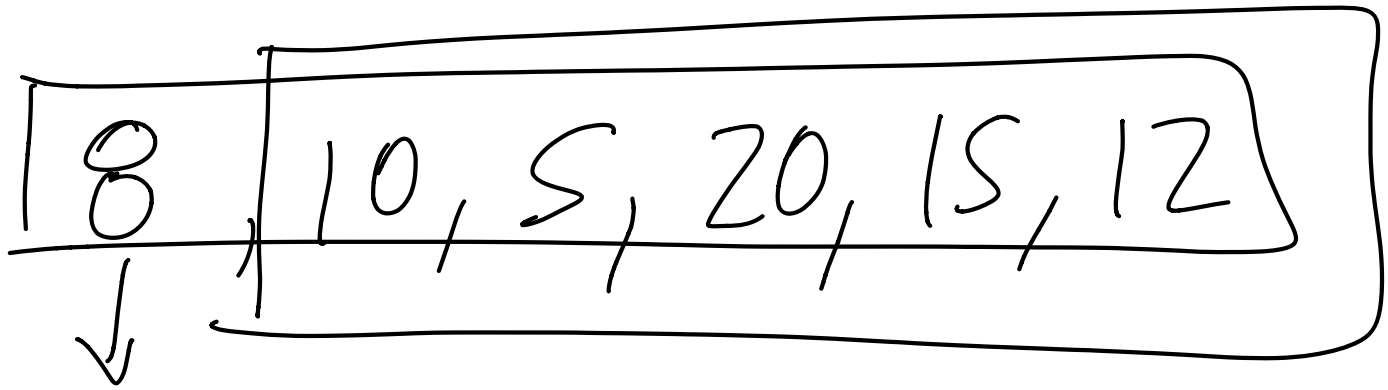
$$= c(2^n - 1) \in O(2^n).$$

$3^n \notin O(2^n)$

3

Sorting

Write a function which takes a list as input outputs the sorted list.



1. Extract the first element

2. Sort the rest

8 | 5, 10, 12, 15, 20

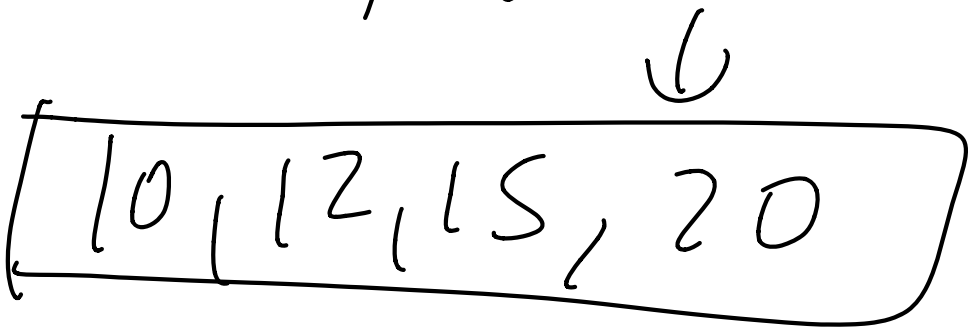
| 5, 8, 10, 12, 15, 20

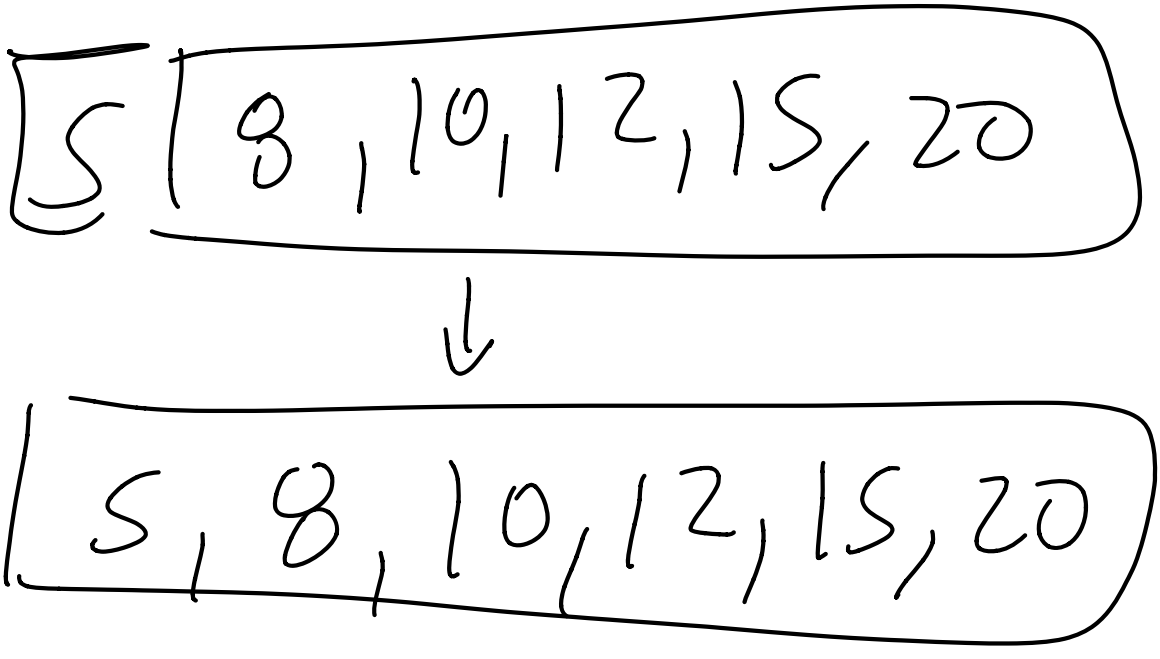
Insertion sort

8: | 5, 10, 12, 15, 20

5 | 10, 12, 15, 20

Recursively add 8 to






```
(define (insertion-sort my-list)
  (cond → constant
    [(empty? my-list) empty]
      → constant
    [else (insert (first my-list)
```

$O(n) = Cn$

(insertion-sort
↓ (rest my-list)))])

T(n-1)

(: insert : Integer (Listof Integer)

→ (Listof Integer))

(define (insert x my-list)

(cond → constant time

[(empty? my-list) (list x)]

[else (if (< x (first my-list))

Insert at the front

→ (cons x my-list)

↑ constant time

(cons (first my-list)
(insert x (rest my-list)))

→ constant $S(n-1)$

Time complexity

Time complexity of insertion-sort is $T(n)$

Time complexity of insert is $S(n)$

$S(n) = S(n-1) + \text{constant}$

$S(0) = \text{constant}$

$S(n) \in O(n)$.

→ n is the size of the second input which is a list.

$T(n) = \text{constant} +$

$T(n-1) + O(n)$

$T(0) = \text{constant}.$

Recurren Relation is

$$T(n) = C + T(n-1) + O(n)$$

$$T(0) = K$$

$$T(n) = C + T(n-1) + Cn$$

$$T(0) = C$$

$$T(n) = C + T(n-1) + Cn$$

$$= ((n+1) + T(n-1)$$

$$= ((n+1) + ((n) + T(n-2)$$

$$= ((n+1) + ((n) + ((n-1)$$

$$+ \; t(n-3)$$

$$= (t(n+1) + (t(n) + (t(n-1)$$

$$+ (t(n-2) + (t(n-3)$$

$$\ldots \quad \in$$

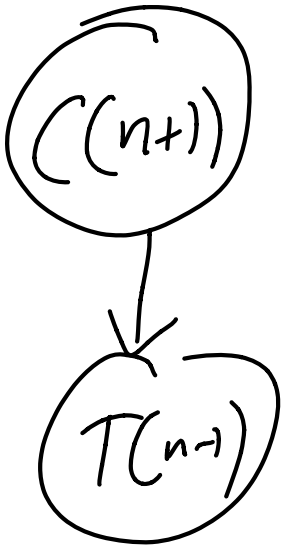
$$= (((t(n+1)) + n + (n-1)$$
$$+ (n-2) + 1)$$

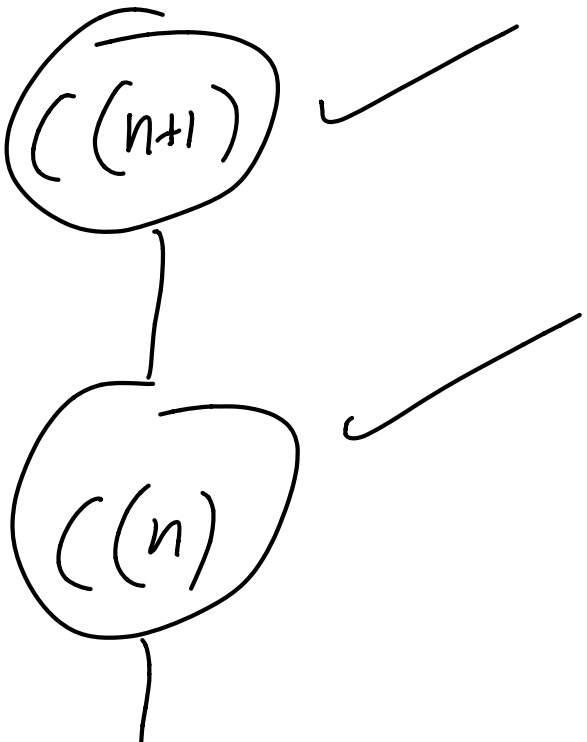
$$= (\frac{(n+1)(n+2)}{2} =$$

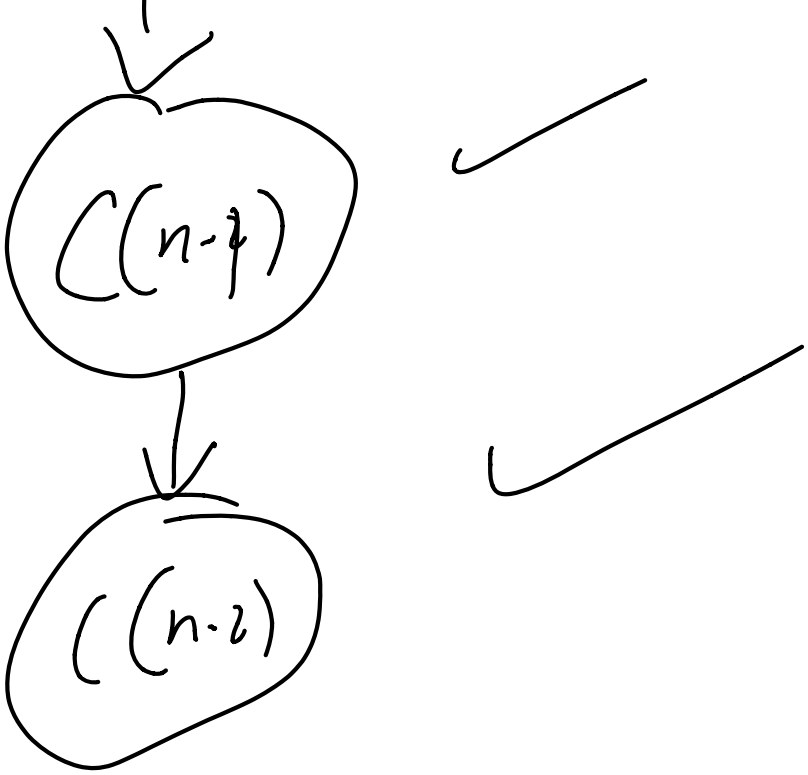
$$\in O(n^2)$$

$T(n) \in O(n^2)$

Tree method







Height of the tree is n.

$$C((n+1) + (n) + (n-1) + (n-2) + (n-3) + \ldots 1)$$

$$= C\left(\frac{(n+2)(n+1)}{2}\right) \in O(n^2).$$