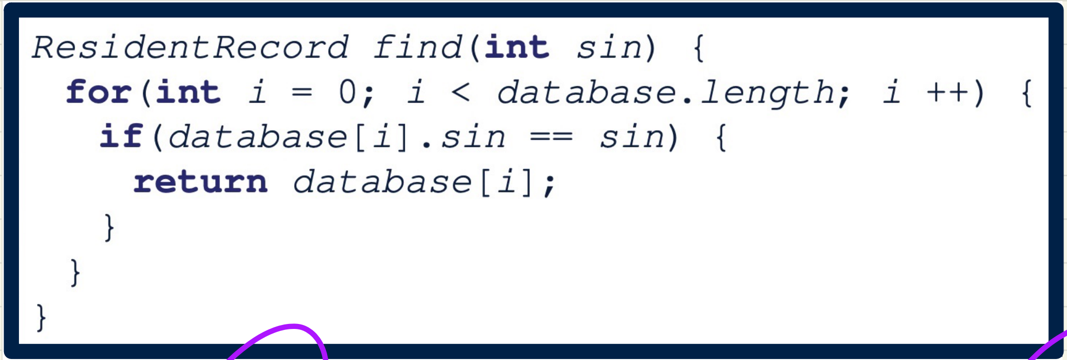
Lecture 2 - Wednesday, January 11

Lecture

Solving Problems via Data Structures

Routing & Compiler

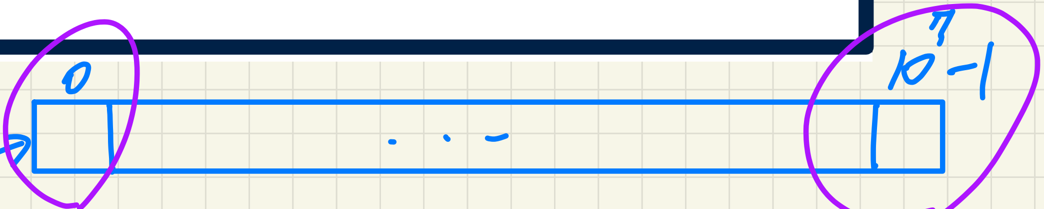
A Searching Problem



```
ResidentRecord find(int sin) {
  for(int i = 0; i < database.length; i ++) {
    if(database[i].sin == sin) {
      return database[i];
    }
  }
}
```

Efficient Solution

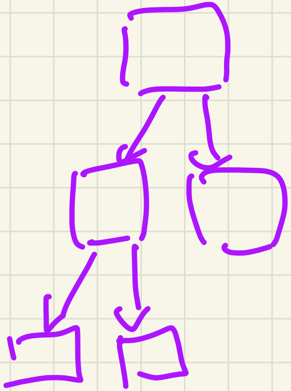
balanced binary search tree



database

best case

1 iteration



worst case

10^7 iterations

# residents

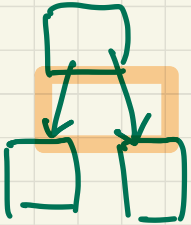
n.

not efficient

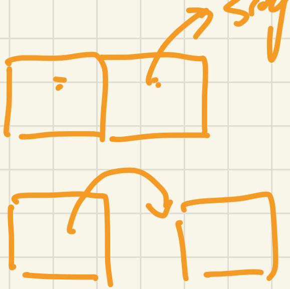
Balance Binary ✓ Search Tree vs. ~~Array~~

linear

guarantees height of tree : $\log_2 N$

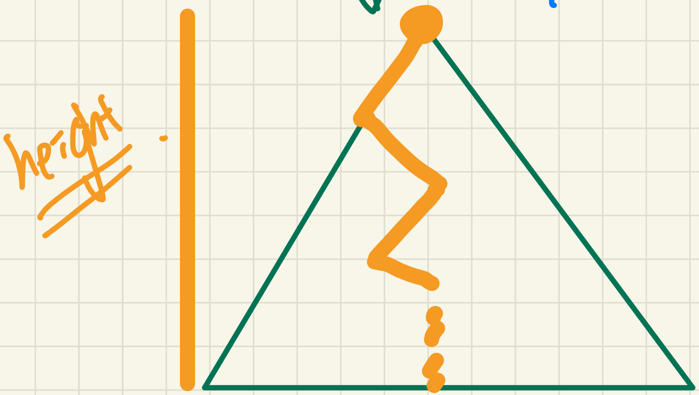


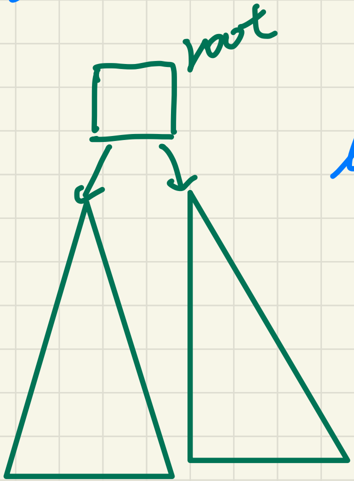
multiple successors



single → unique successor

$1000 \approx 2^{10}$





$\log_2 \boxed{10^7} = \log_2 (10^3)^{2.333}$

residents in city.

$\approx$

2^{10}

$\log_2 2^{23.3}$

Program Optimization Problem

EECS 4302 Compilers

```
b := ... ; c := ... ; a := ...
across ~~i~~ 1 |..| n is i
  loop
    read d
    a := a * 2 * b * c * d
  end
```

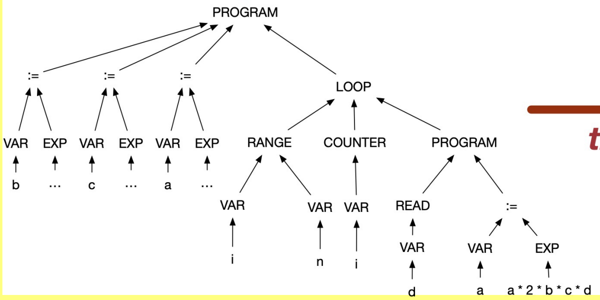
stays invariant between iterations

optimized

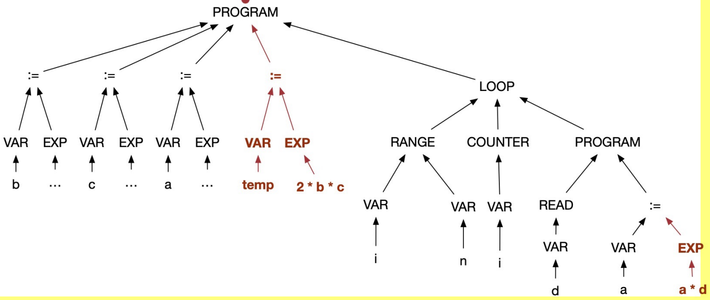
```
b := ... ; c := ... ; a := ...
temp := 2 * b * c * d
across i |..| n is i
  loop
    read d
    a := a * ~~d~~ temp
  end
```

parsed

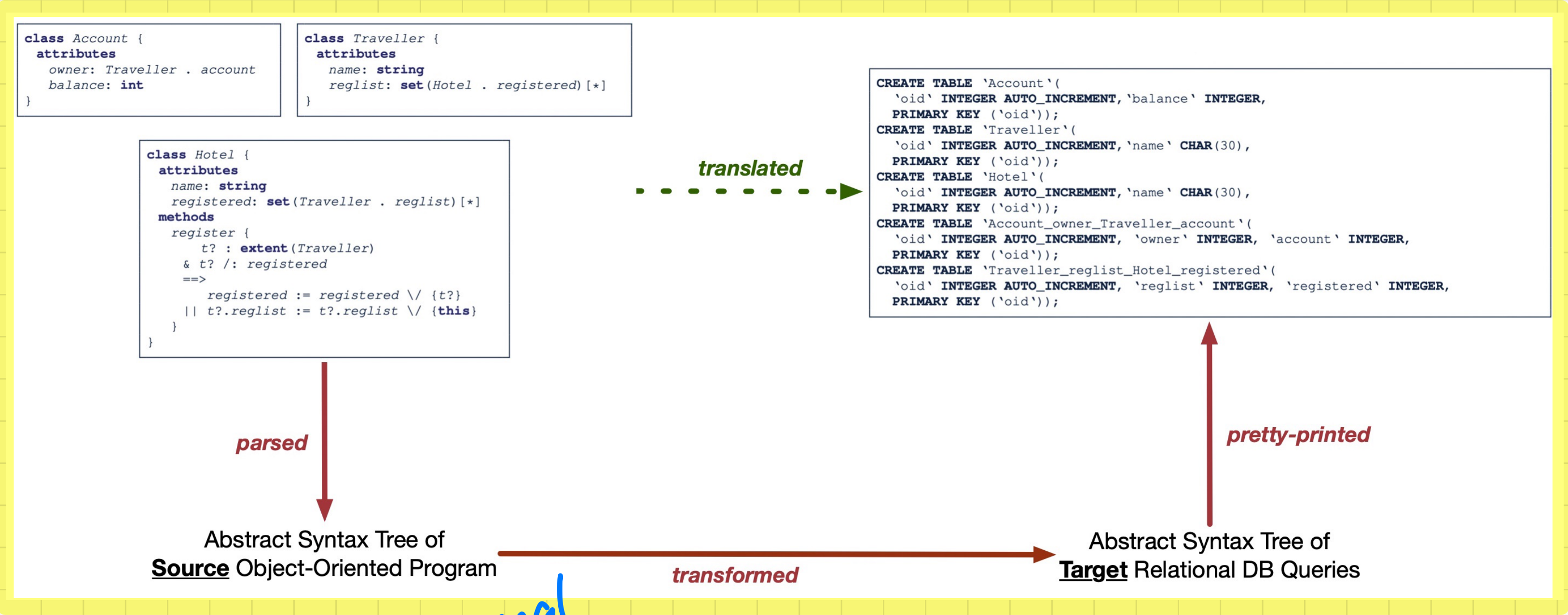
pretty-printed



transformed



Program Translation Problem



```
class Account {
  attributes
    owner: Traveller . account
    balance: int
}
```

```
class Traveller {
  attributes
    name: string
    reglist: set(Hotel . registered)[*]
}
```

```
class Hotel {
  attributes
    name: string
    registered: set(Traveller . reglist)[*]
  methods
    register {
        t? : extent(Traveller)
      & t? /: registered
      ==>
         registered := registered \/ {t?}
      || t?.reglist := t?.reglist \/ {this}
    }
}
```

translated

```
CREATE TABLE `Account`(
  `oid` INTEGER AUTO_INCREMENT,`balance` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller`(
  `oid` INTEGER AUTO_INCREMENT,`name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Hotel`(
  `oid` INTEGER AUTO_INCREMENT,`name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Account_owner_Traveller_account`(
  `oid` INTEGER AUTO_INCREMENT, `owner` INTEGER, `account` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller_reglist_Hotel_registered`(
  `oid` INTEGER AUTO_INCREMENT, `reglist` INTEGER, `registered` INTEGER,
  PRIMARY KEY (`oid`));
```

parsed

pretty-printed

Abstract Syntax Tree of
Source Object-Oriented Program

transformed

Abstract Syntax Tree of
Target Relational DB Queries

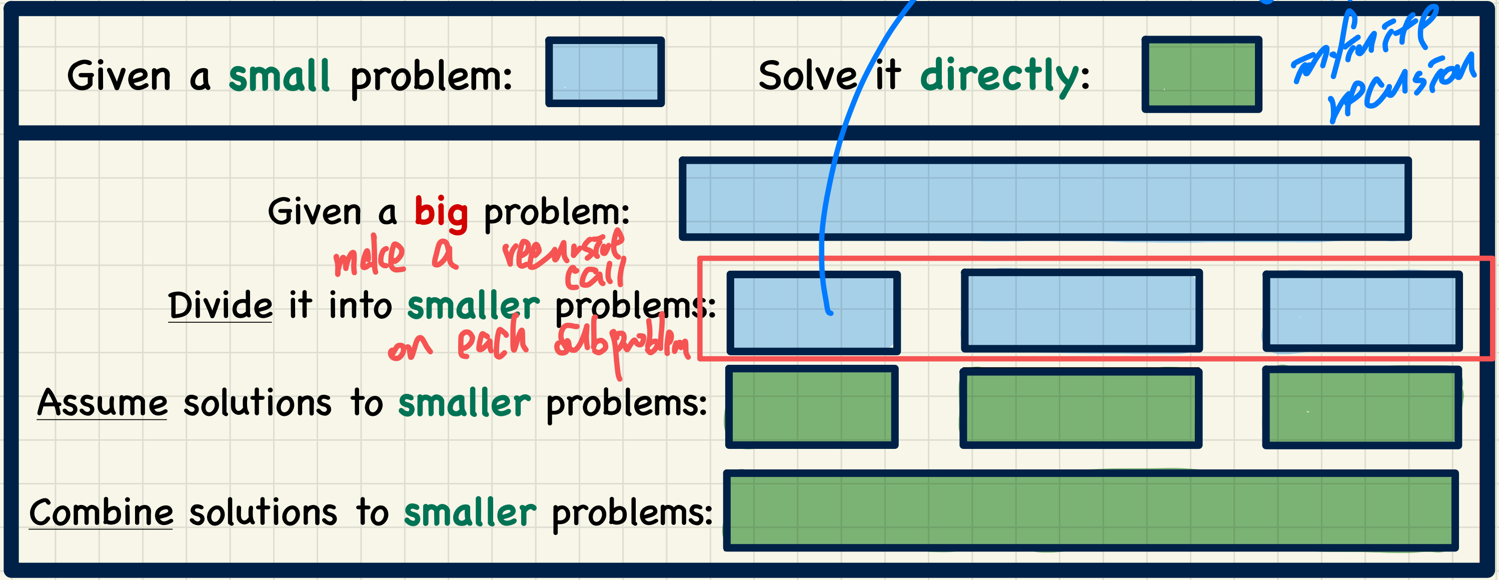
object-relational bridge

Lecture

Reviews on Recursion

Principle, Implementation, Tracing

Solving a Problem **Recursively**



Given a **small** problem: 🟦 Solve it **directly**: 🟩

Given a **big** problem: 🟦

make a recursive call

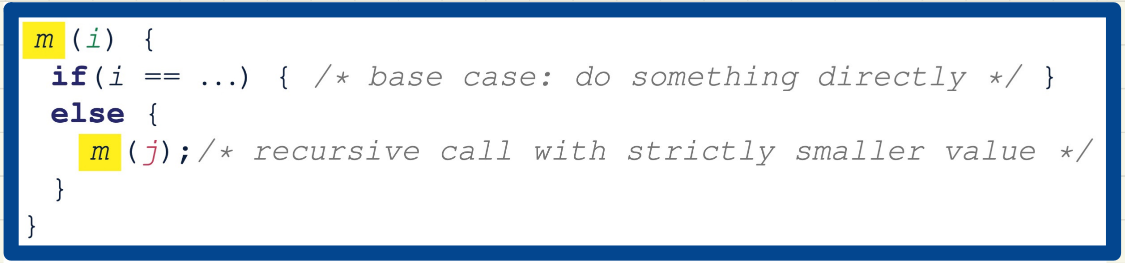
Divide it into **smaller** problems: 🟦 🟦 🟦

on each subproblem

Assume solutions to **smaller** problems: 🟩 🟩 🟩

Combine solutions to **smaller** problems: 🟩

each subproblem strictly smaller; otherwise infinite recursion



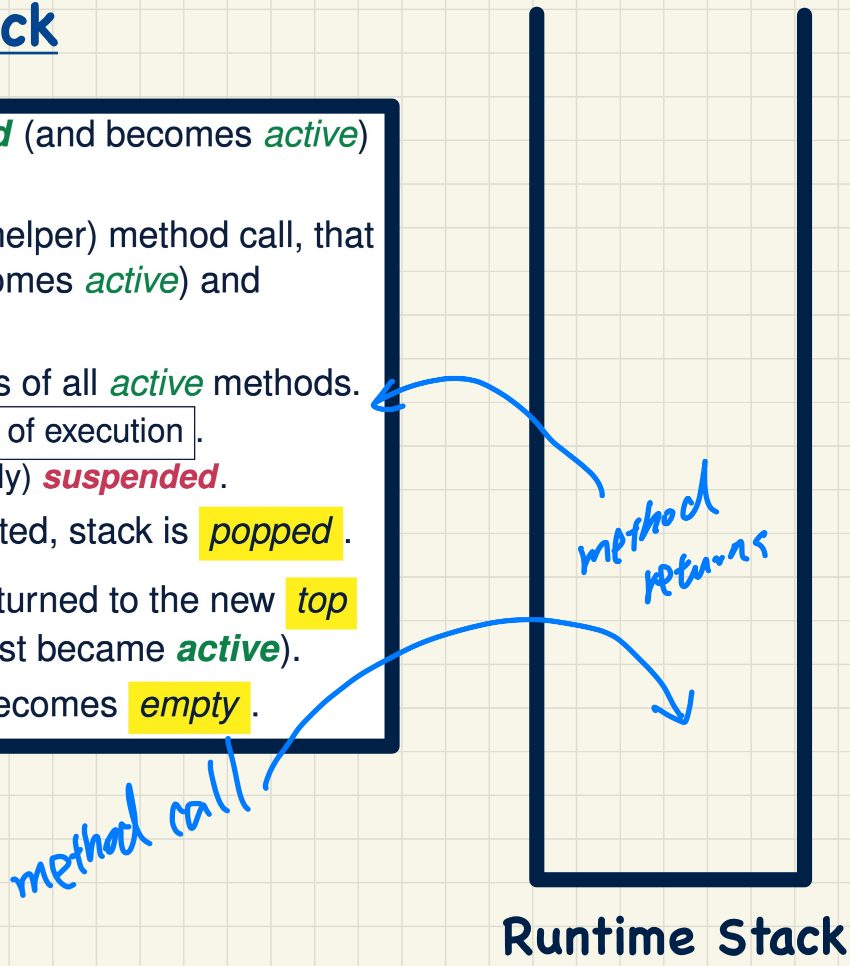
```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

Tracing Recursion via a Stack

- When a method is called, it is ***activated*** (and becomes *active*) and *pushed* onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and *pushed* onto the stack.

  ⇒ The stack contains activation records of all *active* methods.
  - *Top* of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

  ⇒ The current point of execution is returned to the new *top* of stack (which was ***suspended*** and just became ***active***).
- Execution terminates when the stack becomes *empty*.



Recursive Solution: Fibonacci Numbers

0 1 2

F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

base cases

F_0

F_1

recursive cases

$\underline{\underline{F_4}} + \underline{\underline{F_5}} = \underline{\underline{F_6}}$

$F_n = F_{n-1} + F_{n-2}$

↳ $n > 1$

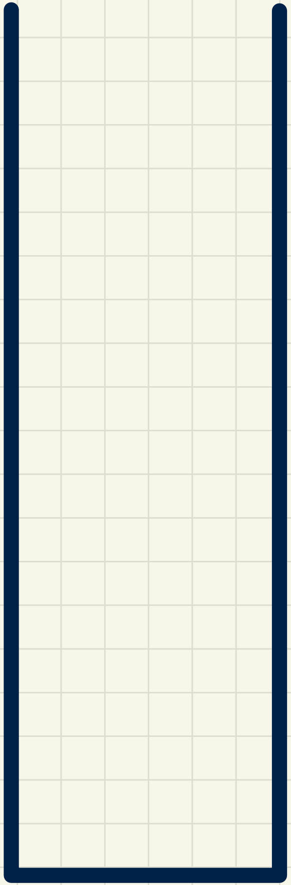
Recursive Solution in Java: Fibonacci Numbers

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```java
int fib (int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib (n - 1) + fib (n - 2);
  }
  return result;
}
```

Example: fib(4)

Exercise: Trace fib(4) via a call stack.

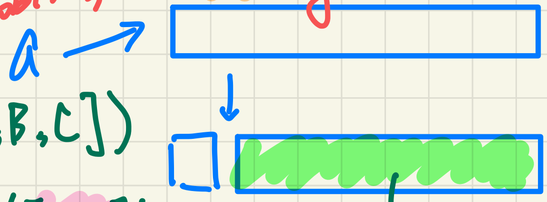


Runtime Stack

Recursion on an Array: Passing new Sub-Arrays

```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
   int[] sub = new int[a.length - 1];
   for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
   m(sub) } }
```

efficiency problem, use call by value



Say a1 = {}, consider m(a1)

to resolve space ←

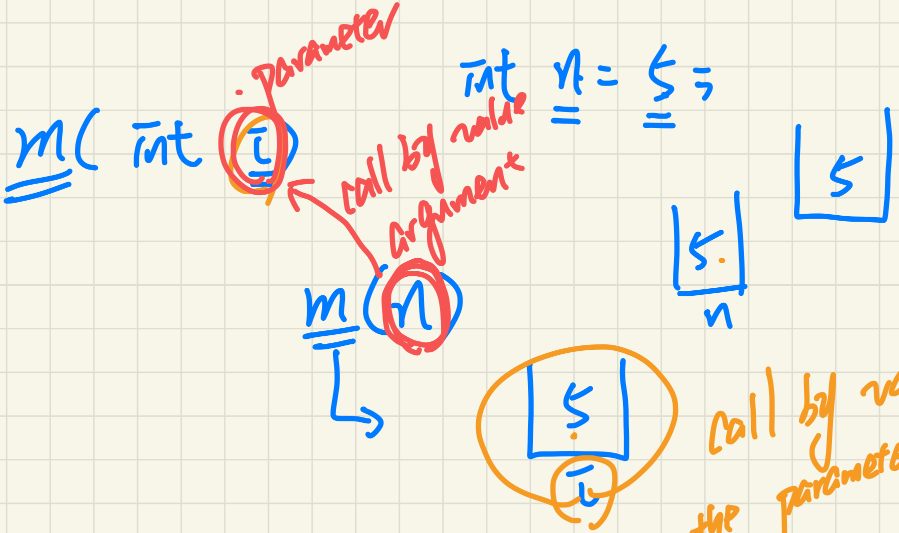
m([A,B,C])

Say a2 = {A, B, C}, consider m(a2)

m([B,C])

m([C])

sub problem
↳ subject to RCs.



call by value:
the parameter i
stores a copy of
the primitive input
value of n.

Call by value : Reference Type

Person jim = new Person(..);

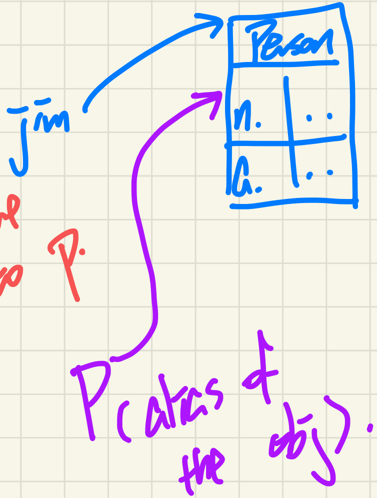
m(Person p)

↳ m(jim)

call by value:

p = jim;

↳ copy the address value of jim to p.



Recursion on an Array: Passing Same Array Reference

ref type (call by value)

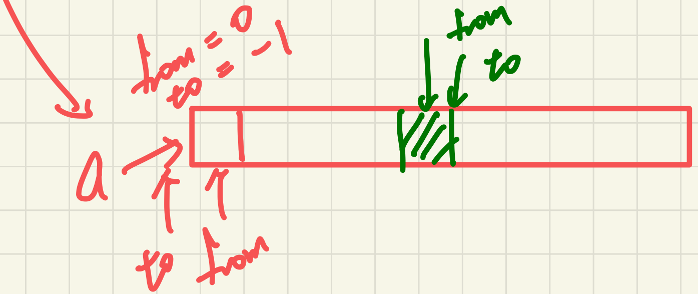
```
void m(int[] a, int from, int to) {
  if (from > to) { /* base case */ }
  else if (from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

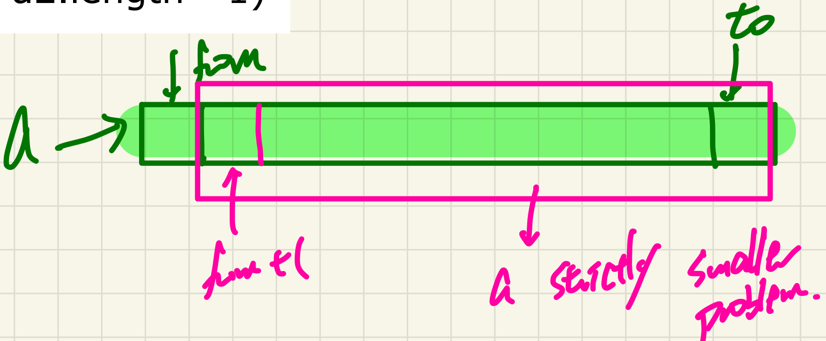
only a ref.

indicating the range of input array that's meant to be examined in the current recursive call.

Say a1 = {}, consider m(a1, 0, a1.length - 1)

Say a2 = {A, B, C}, consider m(a2, 0, a2.length - 1)





Problem: Are All Numbers Positive?

universal property

$(\forall x : False \bullet P(x))$ $\equiv$ True .

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

c.b.v.

recursive helper method

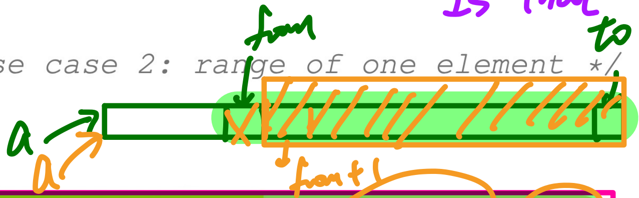
✓

B.C.

Is there

(existential prop? a positive #?

↳ false



Empty array: all elements are positive (True)

∵ no way to find a witness to show otherwise