Exam Review I

Monday December 9

Selection Sort

Insertion Sort (

$O(n^2)$

1000 elements

$1000^2 = 1M$ ..

Merge Sort

$O(n \cdot \log n)$

1000 elements

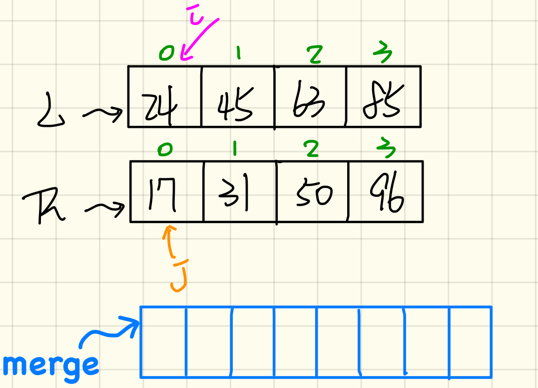
$1000 \cdot \log 1000$

10

$= 10000$

Arrays.sort .

Merge Sort in Java




```java
/* Assumption:  L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && j < R.size()) {
      if( L.get(i)  <= R.get(j) ) { merge.add(L.get(i)); i ++; }
      else { merge.add(R.get(j)); j ++; }
    }
    /* If i >= L.size(), then this for loop is skipped. */
    for(int k = i; k < L.size(); k ++) { merge.add(L.get(k)); }
    /* If j >= R.size(), then this for loop is skipped. */
    for(int k = j; k < R.size(); k ++) { merge.add(R.get(k)); }
  }
  return merge;
}
```

Exercise: why O(n)?

```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = merge(sortedLeft, sortedRight);
  }
  return sortedList;
}
```

may not be sorted.

split

$T(0) = 1$

$T(1) = 1$

$T(n) = 2 * T(n/2) + n + 1$

split

left & right

merge

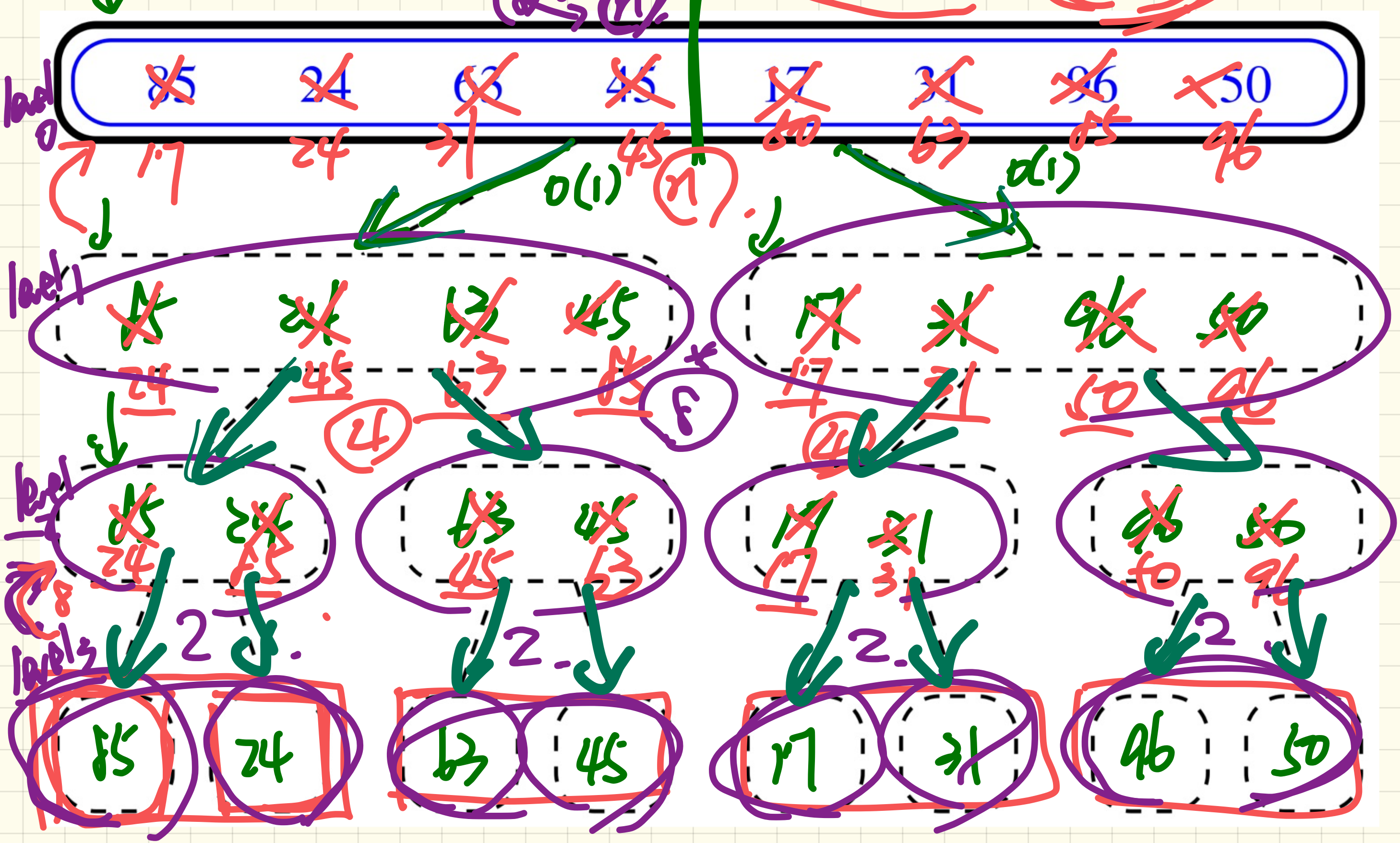
Merge Sort: Tracing

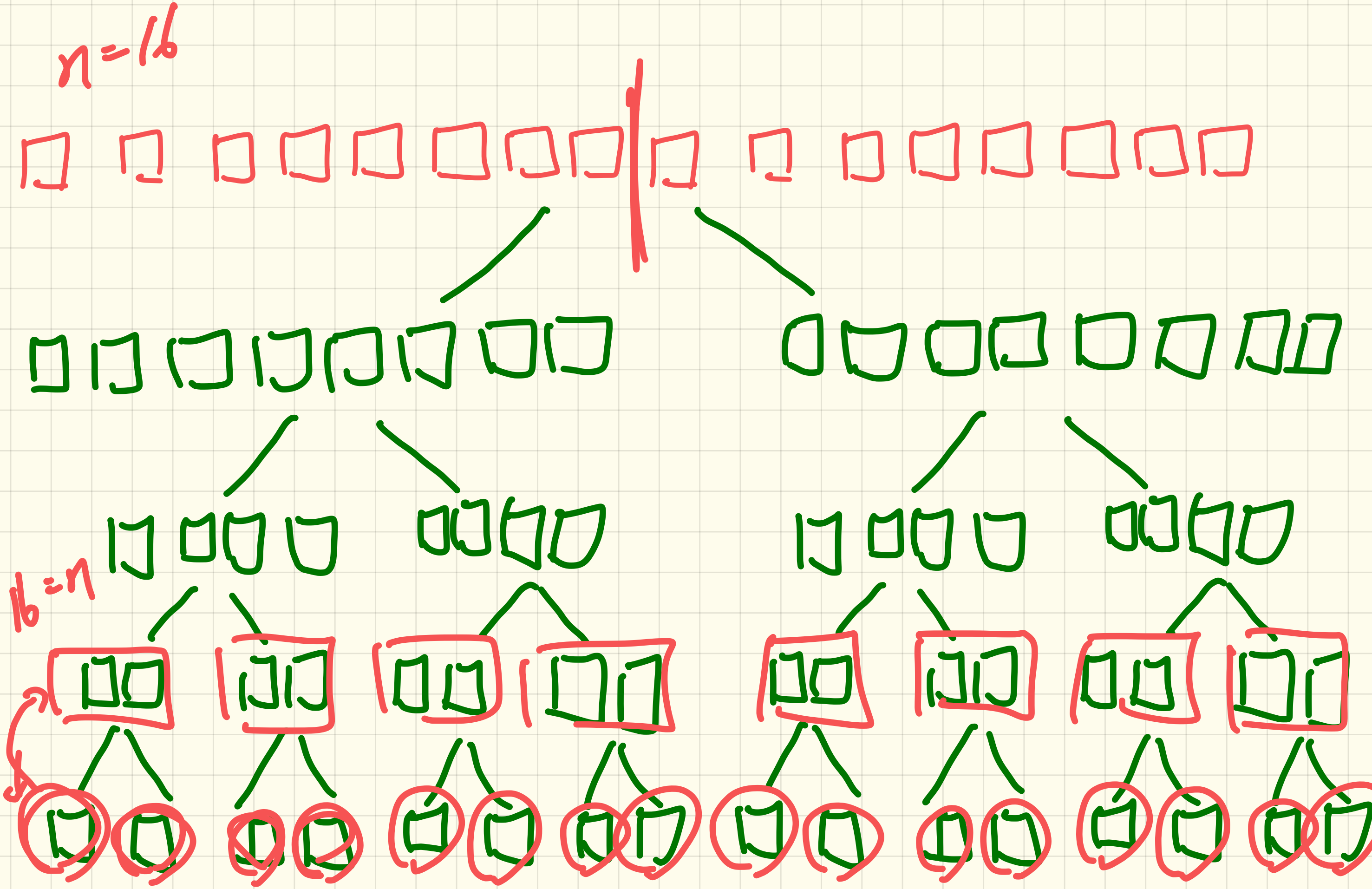
RT from L3 to L2: 8 elements n

RT from L2 to L1:

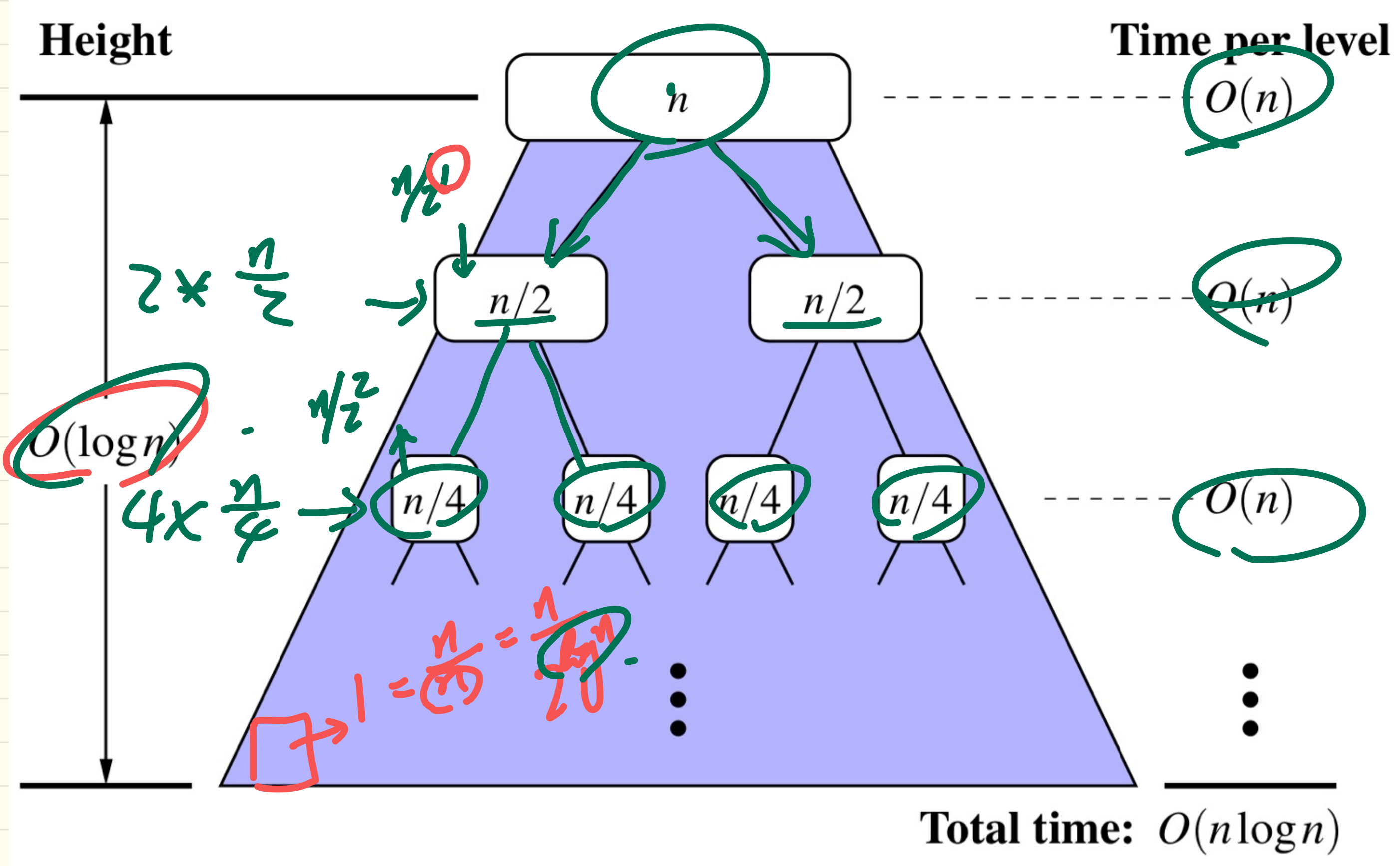
split

merge




n=16
16=n

Merge Sort: Running Time



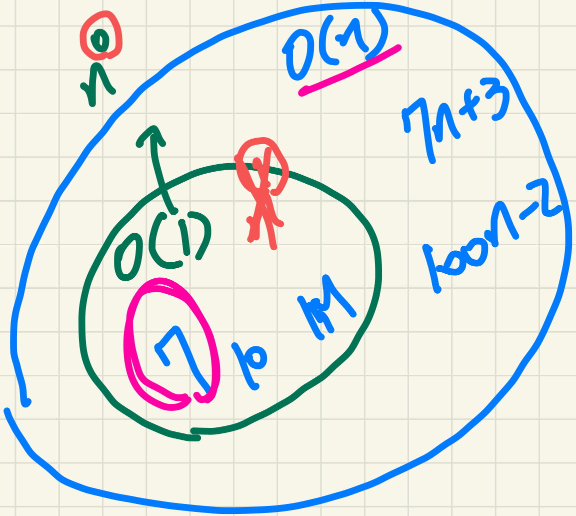
$n^k \qquad k \geq 0$

$n^0 = 1 \qquad O(1)$

$n^1 = n \qquad O(n)$

$\vdots$

$O(?)$ a set of functions which can be upper bounded by ?

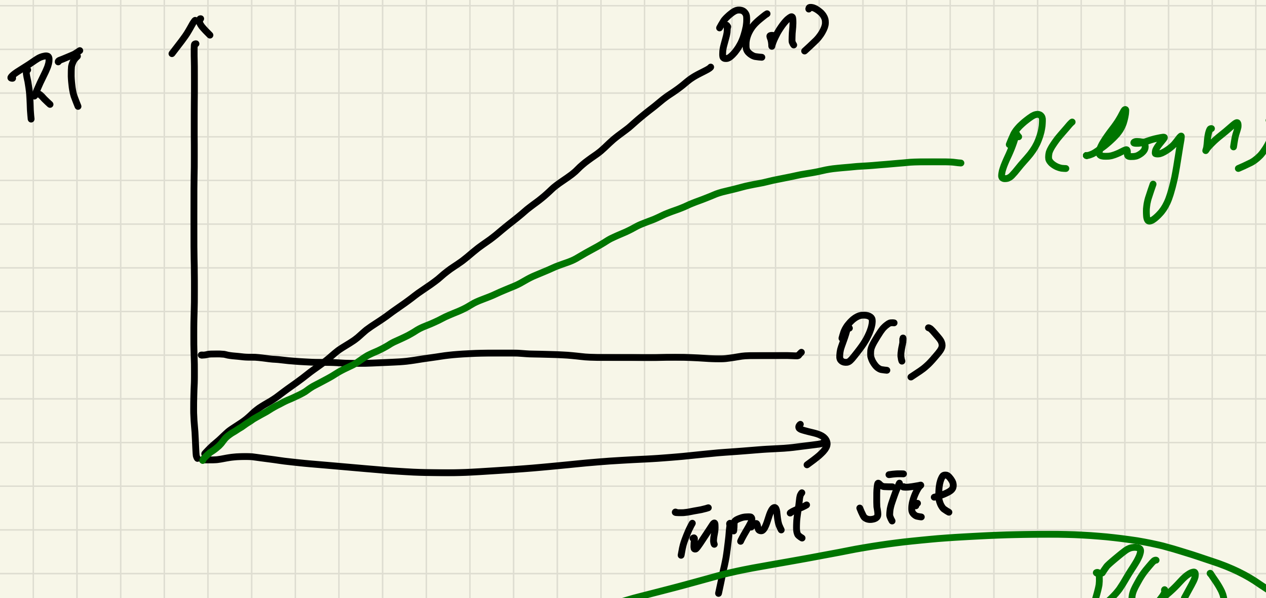


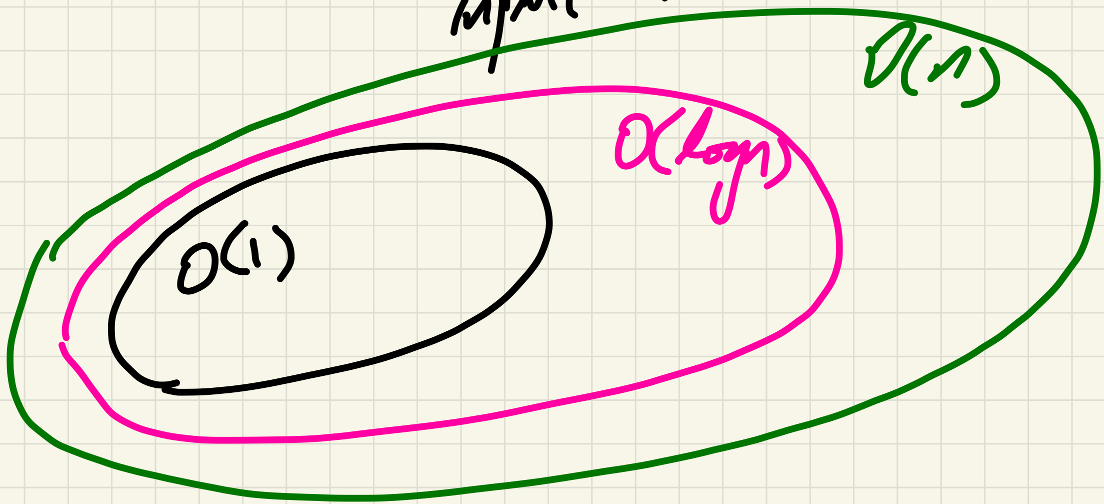
✓ $7 \in O(1)$

✓ $7 \in O(n)$

$7 \in O(1)$

$c = 7$


RT
O(n)
O(log n)
O(1)
Input size


O(n)
O(log n)
O(1)

$$f(n) = 5n^2 + 3n \cdot \log n + 2n + 5$$

$\leftarrow O(n^2)$ $\quad 5 \cdot 1^2 + 3 \cdot 1 \cdot \log 1 + 2 \cdot 1 + 5$

$5 + 3 + 2 + 5 = \boxed{15}$

Prove.

choose $\boxed{C} = ?$

$n_0 = ?\ 1$ s.t.

$f(n) \le \cancel{C} \cdot n^2$

15

$f(1) \le 15 \cdot 1^2$

12

$$f(n) = \boxed{3} \cdot \log n + \boxed{2}$$

$$f(n) \in O(\log n).$$

$$3 \cdot \log \cancel{n} + 2 \le \cancel{c} \cdot \log \cancel{n}$$

$$\quad 2 \qquad\qquad\qquad 5 \quad\;\; 2$$

$$3+2$$

$$5 \le 5 \quad ✓$$

Prove.

choose $c = \cancel{?}\ 5$

$n_0 = \cancel{?}\ \cancel{1}\ 2$ s.t.

$$3 \cdot \log \cancel{n} + 2 \le \cancel{c} \cdot \log \cancel{n} \quad \text{for } n \ge n_0$$

$$\quad 1 \qquad\qquad\qquad 5 \quad\; 1$$

$$2 \le 0$$

A (am) → B (bm) → C (cm)

E (em) → F (fm) → D (dm) → C

✓ D obj = new F();

① ((E) obj).em ✓ compile.

② ((E) obj).am ✗ not compile

downcast

(E) obj.em ✗

compile does not ∵ cannot expect em on D.

CCE

↳ ∵ DT F is not a descendant of cast type E.

em
fm
dm
cm

((E) obj).dm ✓

downcast

merge sort :

$\boxed{O(n \cdot \log n)}$
tightest.

merge sort $\in O(n^2)$
correct but
not accurate.

insertion sort :

$O(n^2)$
tightest.

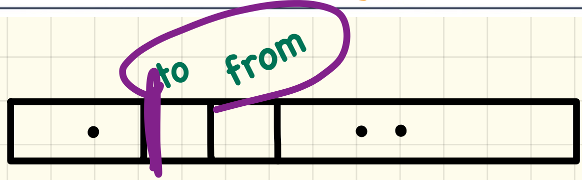
Correctness Proofs: Ideas

from ≤ to

```
boolean allPositive(int[] a) { return allPosH (a, 0, a.length - 1); }
boolean allPosH (int[] a, int from, int to) {
  if (from > to) { return true; }
  else if(from == to) { return a[from] > 0; }
  else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

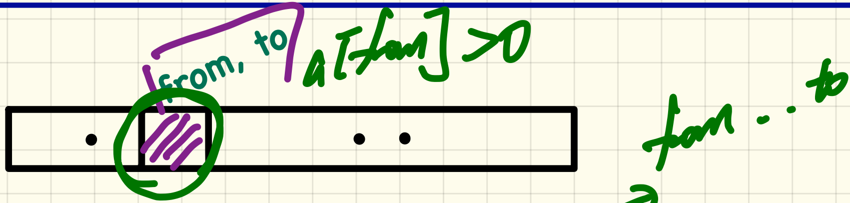
Base Case:

Empty Array



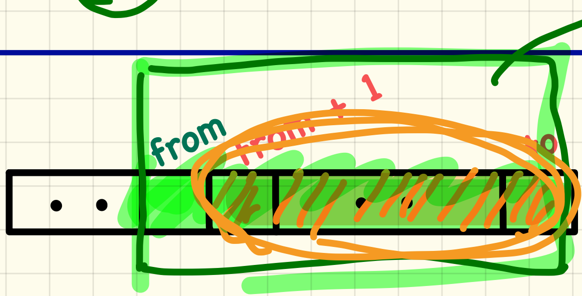
Base Case:

Array of Size 1



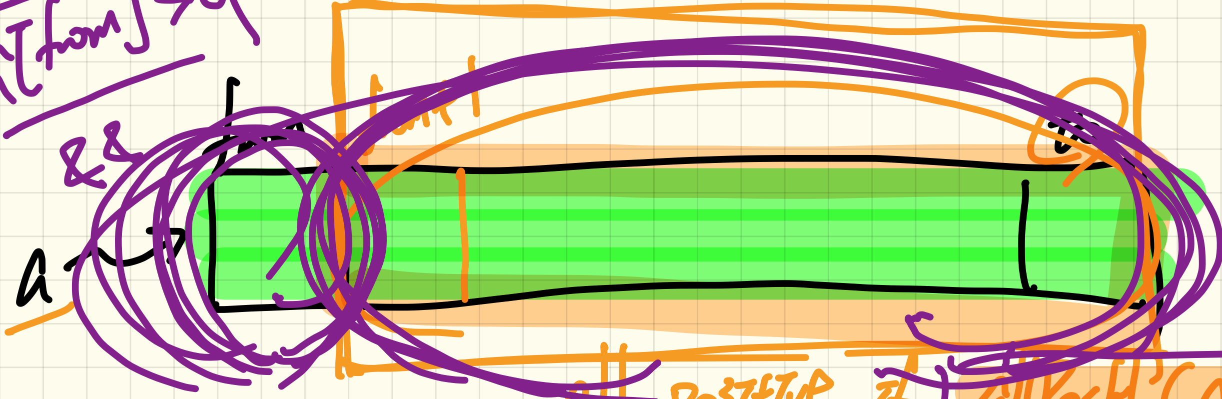
Recursive Case:

Array of size > 1



$\boxed{a[from] > 0}$

&&



all positive iff $\boxed{\text{allPosH}(a, from+1, to) \text{ returns}}$ T.

Problem. Are elements a[from], a[from+1], ..., a[to] all positive?

I.H. calling allPosH(a, from+1, to) will return T iff a[from+1], ..., a[to]

Correctness Proofs

a

```
1  boolean allPositive(int[] a) { return allPosH (a, 0, a.length - 1); }
2  boolean allPosH (int[] a, int from, int to) {
3    if (from > to) { return true; }
4    else if(from == to) { return a[from] > 0; }
5    else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

I.H.

- Via mathematical induction, prove that `allPosH` is correct:
  - **Base Cases**
    - In an empty array, there is no non-positive number ∴ result is ***true***. **[L3]**
    - In an array of size 1, the only one elements determines the result. **[L4]**
  - **Inductive Cases**
    - **Inductive Hypothesis**: `allPosH(a, from + 1, to)` returns ***true*** if a[from + 1], a[from + 2], . . . , a[to] are all positive; ***false*** otherwise.
    - `allPosH(a, from, to)` should return ***true*** if: **1)** a[from] is positive; and **2)** a[from + 1], a[from + 2], . . . , a[to] are all positive.
    - By *I.H.*, result is $a[from] > 0 \wedge$ `allPosH(a, from + 1, to)`. **[L5]**
- `allPositive(a)` is correct by invoking `allPosH(a, 0, a.length - 1)`, examining the entire array. **[L1]**

allP[]

expand

allP.