LECTURE 14

MONDAY OCTOBER 28

Solving Problems Recursively

Problem (P_n)	Base Case(s) (P_0, P_1, P_2)	Recursive Solution(s) to Sub-Problem(s) (P_{n-1}, P_{n-2})	Solution
$factorial(n)$	$P_0 = factorial(0) = 1$	$P_{n-1} = factorial(n-1)$	$n \times P_{n-1}$
$fib(n)$	$P_1 = fib(1) = 1$ $P_2 = fib(2) = 1$	$P_{n-1} = fib(n-1)$ $P_{n-2} = fib(n-2)$	$P_{n-1} + P_{n-2}$
$isP(s)$	$P_0 = isP(\text{""}) = true$ $P_1 = isP(\text{"a"}) = true$	$P_{n-2} = isP(s.substring(1, s.length() - 1))$	$s.charAt(0) == charAt(s.length() - 1)$ && P_{n-2}
$rev(s)$	$P_0 = rev(\text{""}) = \text{""}$ $P_1 = rev(\text{"a"}) = \text{"a"}$	$P_{n-1} = rev(s.substring(1, s.length()))$	$P_{n-1} + s.substring(0)$
$occ(s, c)$	$P_0 = occ(\text{""}, c) = 0$	$P_{n-1} = occ(s.substring(1, s.length()), c)$	$1 + P_{n-1}$ **if** $s.charAt(0) == c$ $0 + P_{n-1}$ **if** $s.charAt(0)$!= c
$allPosH(a,\ from,\ to)$	$P_0 = allPosH(a,\ from,\ to) = true$ **if** $from > to$ $P_1 = allPosH(a,\ from,\ to) = a[from] > 0$ **if** $from == to$	$P_{n-1} = allPosH(a,\ from + 1,\ to)$	$a[0] > 0$ && P_{n-1}
$isSortedH(a,\ from,\ to)$ $isSortedH(a,\ from,\ to)$	$P_0 = isSortedH(a,\ from,\ to) = true$ **if** $from > to$ $P_1 = isSortedH(a,\ from,\ to) = true$ **if** $from == to$	$P_{n-1} = isSortedH(a,\ from + 1,\ to)$	$a[from] \leq a[from + 1]$ && P_{n-1}
$binSearchH(a,\ from,\ to,\ k)$	$P_0 = binSearchH(a,\ from,\ to,\ k) = false$ **if** $from > to$ $P_1 = binSearchH(a,\ from,\ to,\ k) = a[from] == k$ **if** $from == to$	$P_{left} = binSearchH(a,\ 0,\ \lfloor \frac{from + to}{2} \rfloor - 1,\ k)$ $P_{right} = binSearchH(a,\ \lfloor \frac{from + to}{2} \rfloor + 1,\ to,\ k)$	P_{left} **if** $k < a[\lfloor \frac{from + to}{2} \rfloor]$ P_{right} **if** $k > a[\lfloor \frac{from + to}{2} \rfloor]$ $true$ **if** $k == a[\lfloor \frac{from + to}{2} \rfloor]$

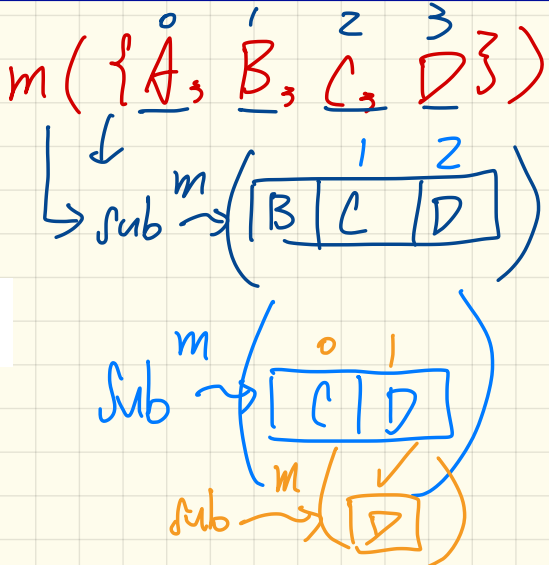
Recursion on an Array: Passing new Sub-Arrays

```
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[0] = a[i - 1]; }
    m(sub) } }
```

Say a1 = {}, consider m(a1)

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

Say a3 = {A, B, C, D}, consider m(a3)



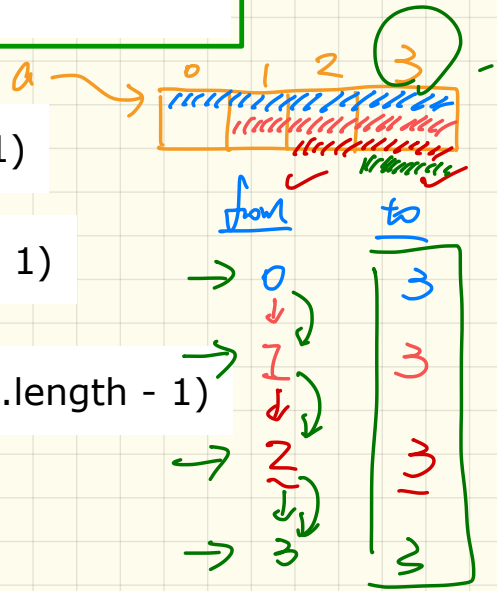
Recursion on an Array: Passing Same Array Reference

```
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) } }
```

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

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

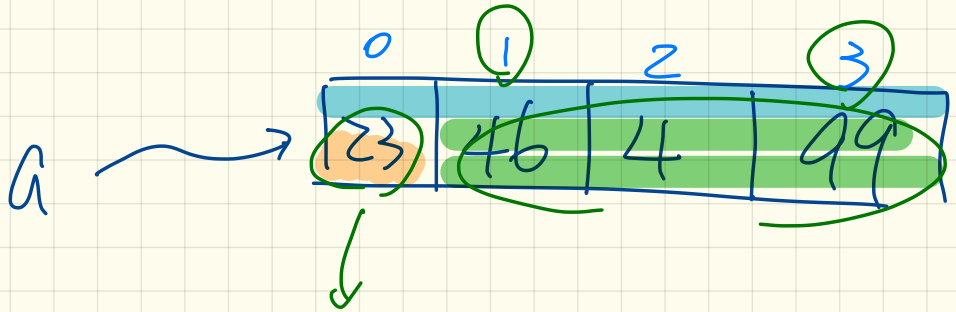
Say a3 = {A, B, C, D}, consider m(a3, 0, a3.length - 1)



$$\left(\forall x \mid\mid \underbrace{\text{false}}_{\text{range}} \cdot P(x)\right) \equiv \text{True}.$$

Are all numbers in an empty array positive?

↳ Is it possible to find a witness in a s.t. it is not positive?



allP(a) true a[0] > 0

&&

true [allP(sub array from index 1 to index 3)

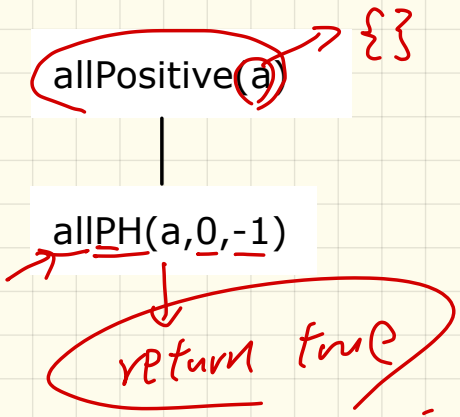
Problem: Are All Numbers Positive?

```
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);
  }
}
```

Tracing Recursion:

allPositive



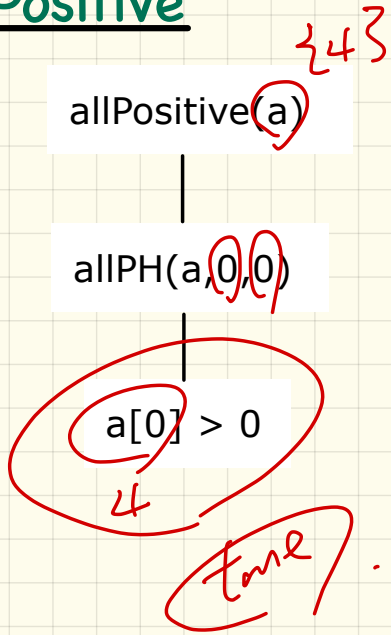

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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);
  }
}
```

Say a = {}

Tracing Recursion:

allPositive




```
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);
  }
}
```

a[0] > 0

false.

Say a = {4}

Say a = {-10}

Tracing Recursion:

allPositive

```
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);
  }
}
```

allPositive(a)

allPH(a,0,3)

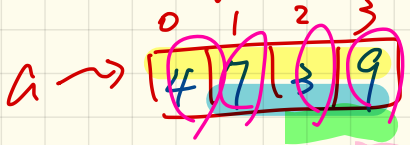
a[0] > 0 && allPH(a,1,3)

a[1] > 0 && allPH(a,2,3)

a[2] > 0 && allPH(a,3,3)

Say a = {4,7,3,9}

a.length 4



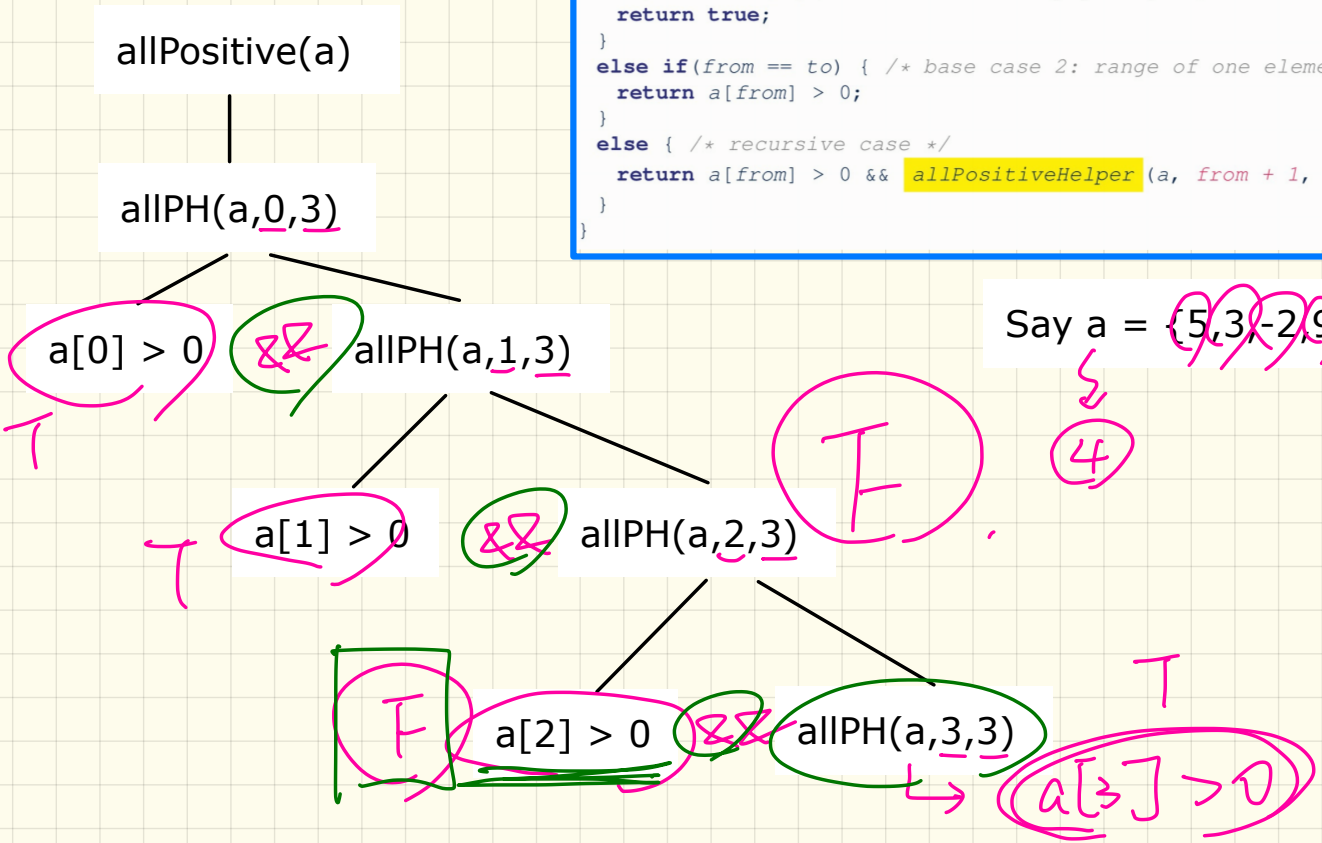
a[3] > 0

Tracing Recursion:

allPositive

```
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);
  }
}
```



An array is sorted in:

(a) ascending order ✓ {1, 3, 4, 5}

{1, 3, 3, 4, 5}

3 < 3 ✗

(b) non-ascending order

! (a[0] < a[1])

≡ a[0] >= a[1]

(c) ✓ descending order

(d) non-descending order

! (a[0] > a[1])

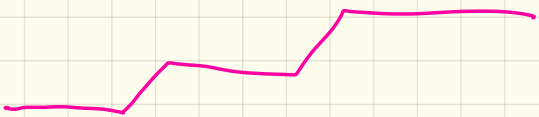
≡ a[0] <= a[1]

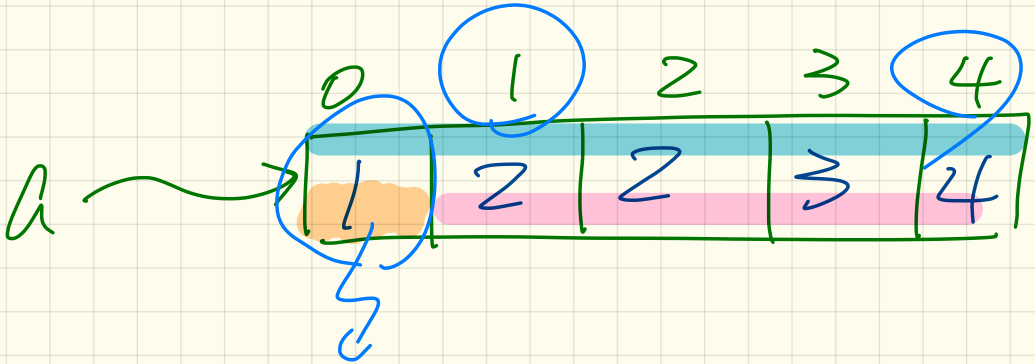
non-decreasing

$$!(a[0] > a[1])$$

$$\equiv$$

$$a[0] \leq a[1]$$





isSorted(a)

= a[0] ≤ a[1] && isSorted(

Sub array from indices 1 to 4)

Problem: Are Numbers Sorted?

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

boolean isSortedHelper(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 true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion:

isSorted

isSorted(a)

isSH(a,0,-1)

return T.

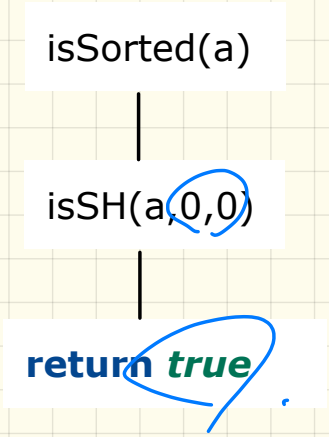
```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(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 true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Say a = {}

Tracing Recursion:

isSorted




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

boolean isSortedHelper(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 true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

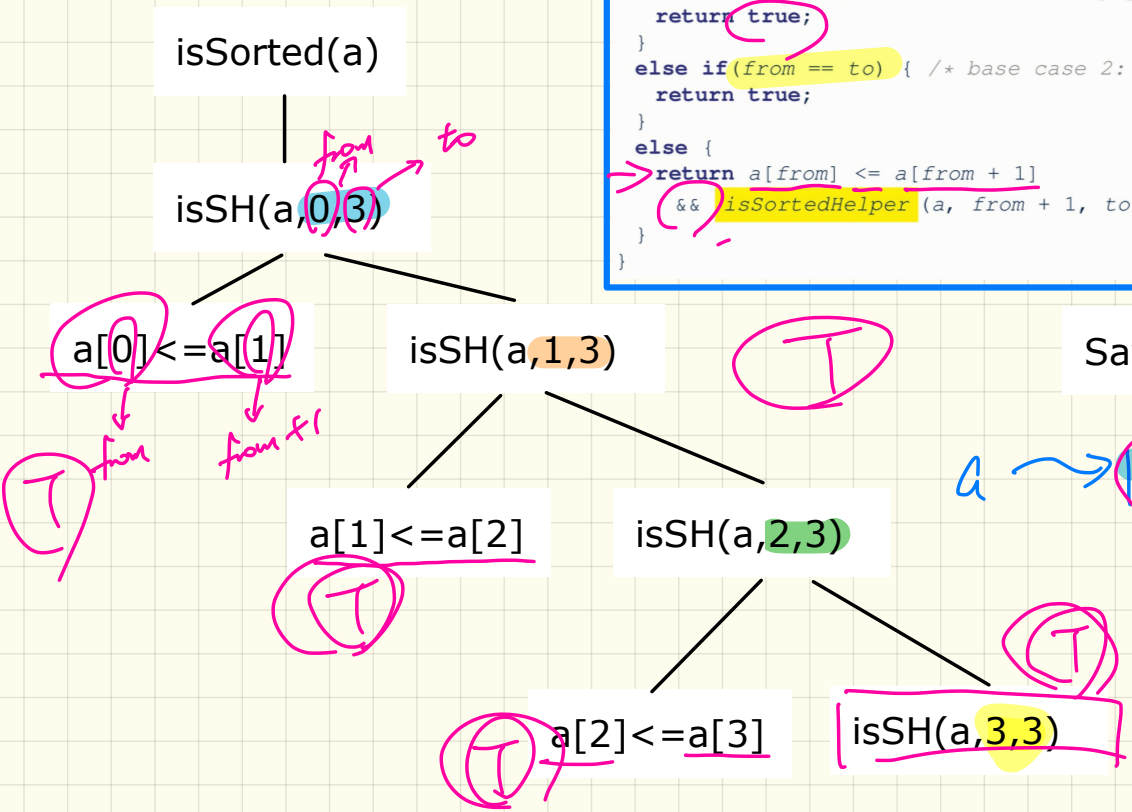
Say a = {4}

Tracing Recursion:

isSorted

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

boolean isSortedHelper(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 true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

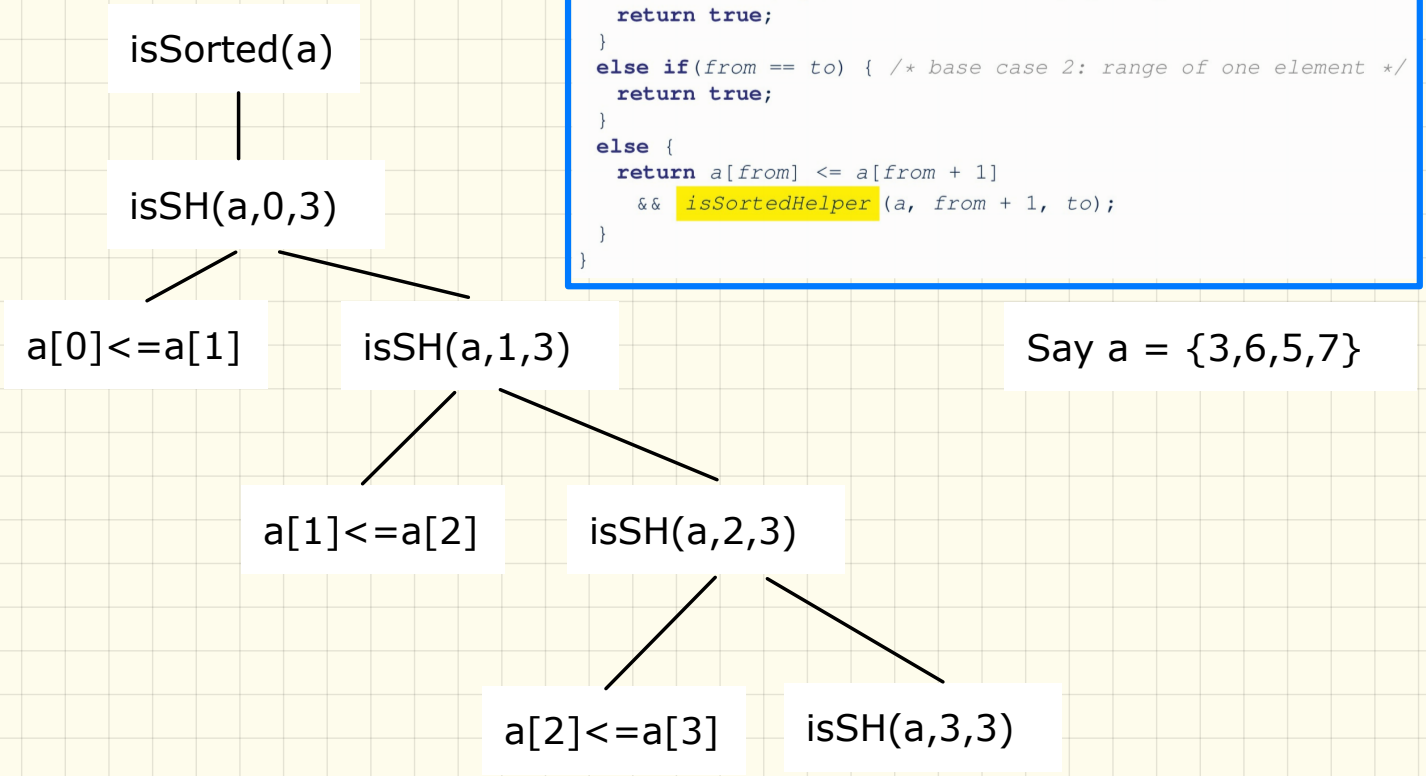


Tracing Recursion:

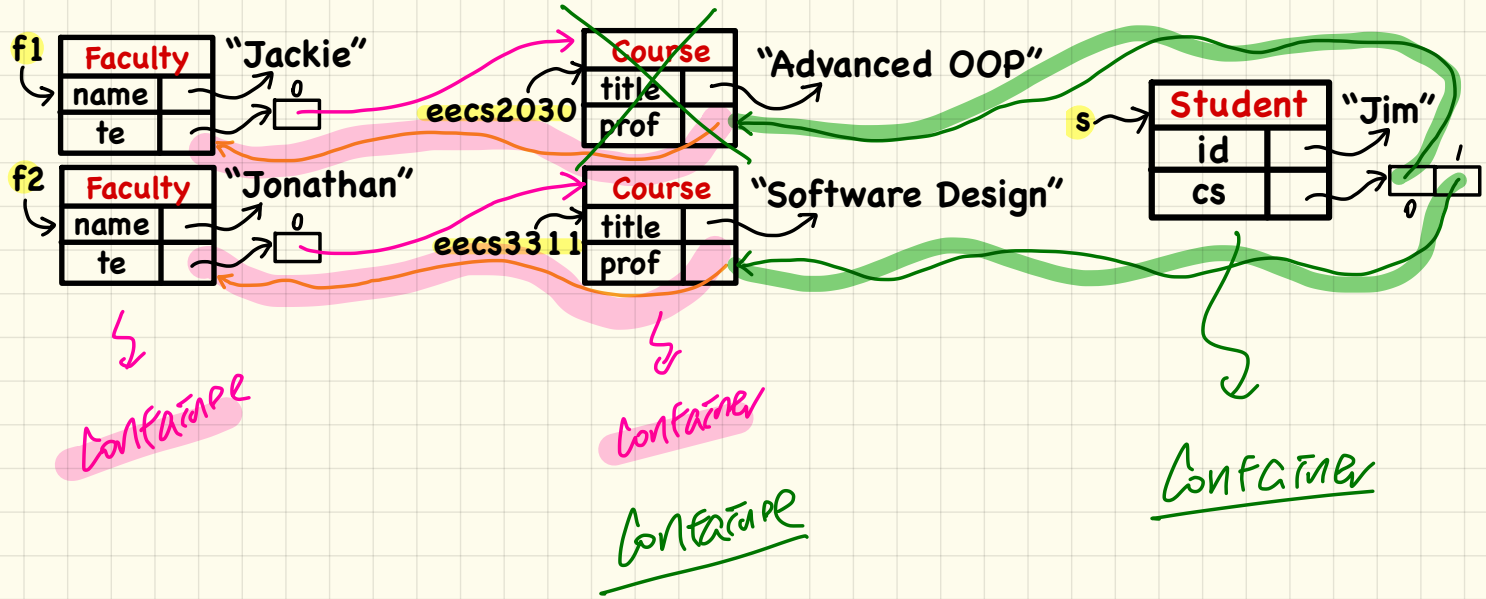
isSorted

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

boolean isSortedHelper(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 true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```



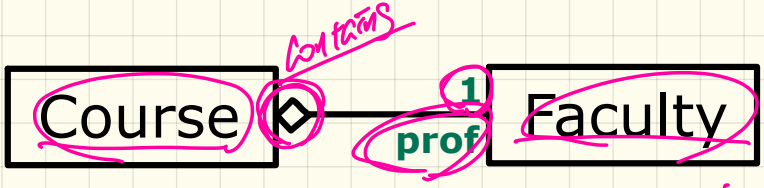
Container vs. Containee



What if a course is deleted?

Aggregation: Design

Design 1: Single Containee



Each Course object contains 1 Faculty object as its prof.

Design 2: Multiple Containees

Each Student object contains a collection of Course objects as their courses.

Java Implementation

```
class Course {
   Faculty prof;
   …
}
```

```
class Faculty {
   …
}
```

```
class Student {
   Course[] courses;
   …
}
```

```
class Course {
   …
}
```