Lecture 23 - Nov. 28

Recursion

Recursion on Strings: occurrencesOf
Recursion on Arrays: Call by Value
Recursion on Arrays: allPositive, isSorted

Announcements/Reminders

- **Lab5** released
  + Required study: **Abstract Classes** & **Interfaces**
- Learning resources on **Recursion**
- **Exam** Review Sessions eClass Polling
- A tutorial session on **recursion**: 4pm on Sunday
- **ProgTest3** results to be released on Monday

Recursions on Strings

ex. "abcd"

Palindrome

"racecar" → strictly smaller problem

"aracecars" != "raceacar"

no need to recur further.

Reversal

rev("abcd")

rev("bcd") + a

d c b

Number of Occurrences

"abca"

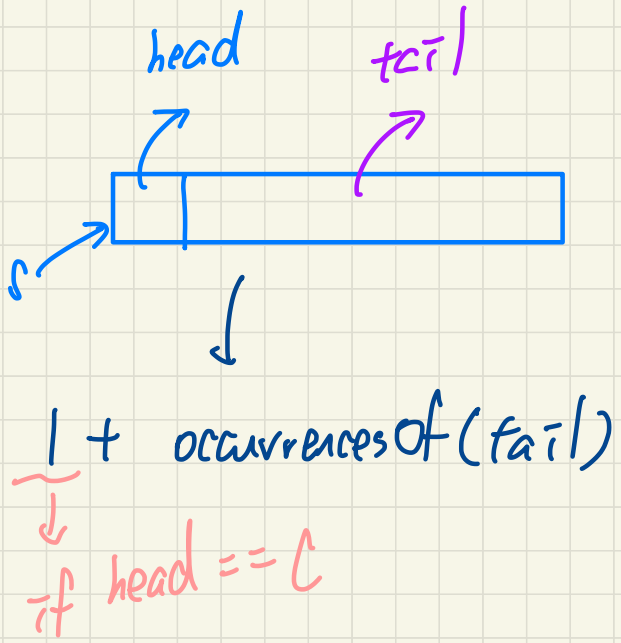
of 'a'

of 'a' → 1 + occurrences of ("bca", 'a')

of 'b' → 0 + occurrences of ("bca", 'b')

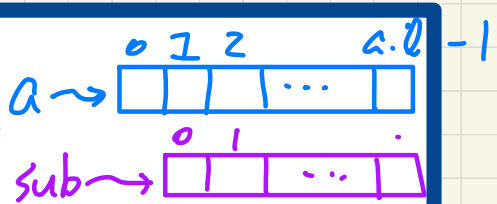
Problem: Number of Occurrences

```
int occurrencesOf(String s, char c) {
  if(s.isEmpty()) {
    /* Base Case */
    return 0;
  }
  else {
    /* Recursive Case */
    char head = s.charAt(0);
    String tail = s.substring(1, s.length());
    if(head == c) {
      return 1 + occurrencesOf(tail, c);
    }
    else {
      return 0 + occurrencesOf(tail, c);
    }
  }
}
```



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) } }
```

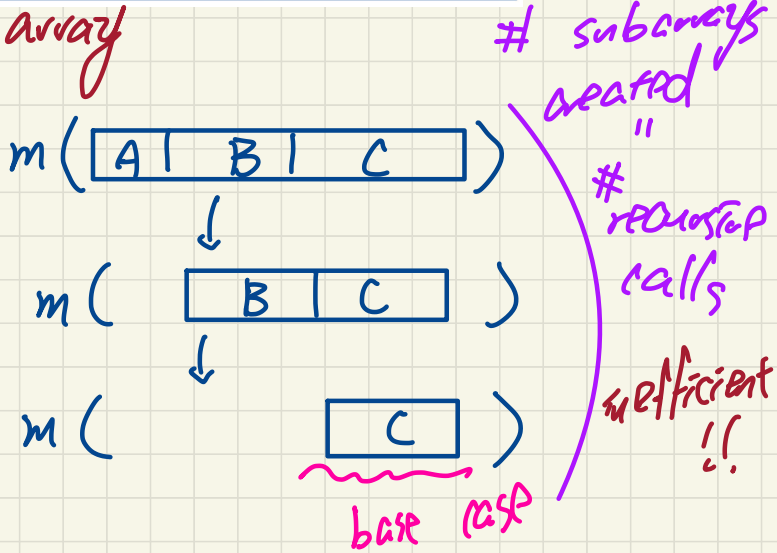


recursive call on a strictly smaller array

Say a1 = {}, consider m(a1)

↳ reached base case

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



Recursion on an Array: Passing Same Array Reference

non-base case: from < to

```
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) } }
```

empty range

single ele. range

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

length 0

-1

0 > -1 → base case #1

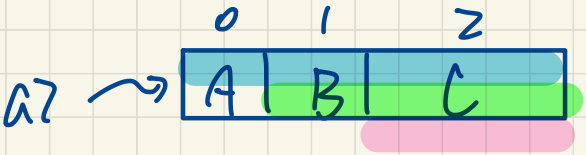
(from) (to)

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

m(a2, 0, 2)

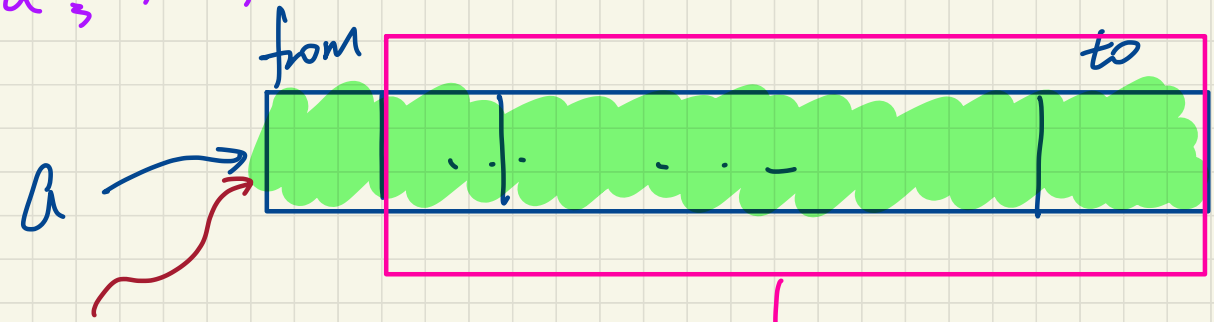
m(a2, 1, 2)

m(a2, 2, 2)



base case #2

m(a, from, to)



m(a, from+1, to)

strictly smaller subrange.

Problem: Are All Numbers Positive?

Is there some positive number?

$\exists x \cdot P(x)$ → False ∵ no witness to show satisfy.

$x \in \phi$

Say a = {}

$\forall x \cdot P(x) \equiv$ True → ∵ no witness in empty collection to show violation of property.

$x \in \phi$

Say a = {4} True

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

0 1 2 3

allPos({4,7,3,9}, 0, 3)

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

Say a = {5,3,-2,9} False

Problem: Are All Numbers Positive?

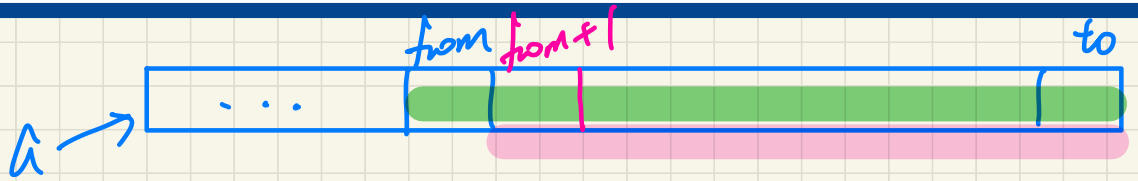
what a user needs to supply when using the solution

public

private

recursive helper method

```
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

non-descending : $\neg(a[i] > a[i+1])$

non-ascending ? → $a[i] \leq a[i+1]$

Say a = {}

allPositive(a)

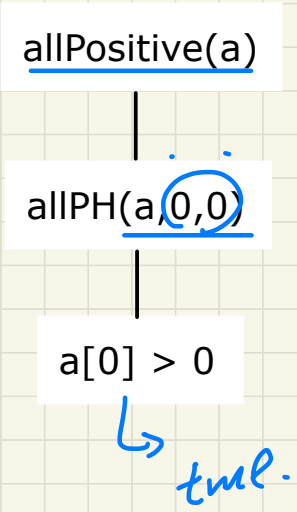
allPH(a,0,-1)

tail.

```
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

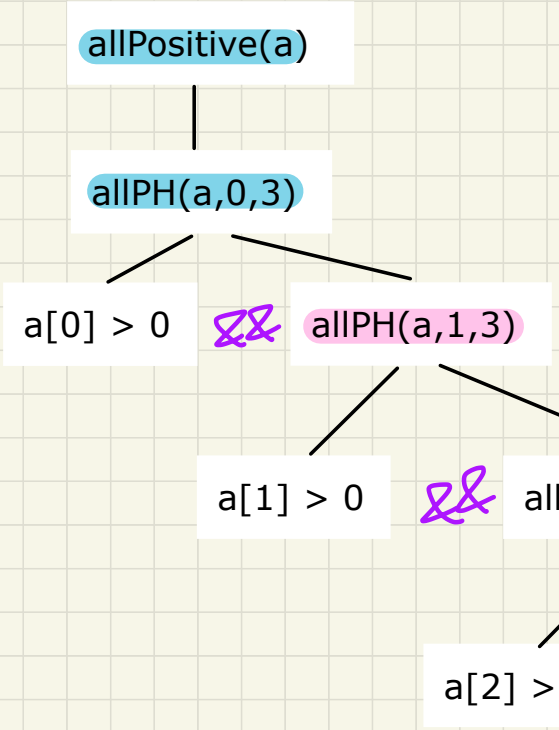
Say a = {4}



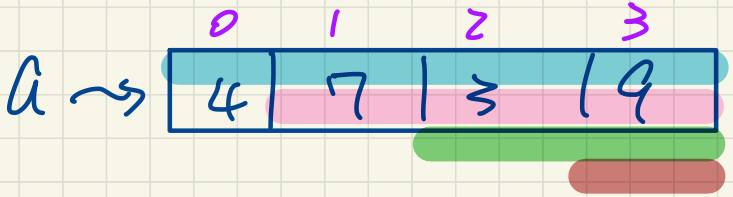

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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

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



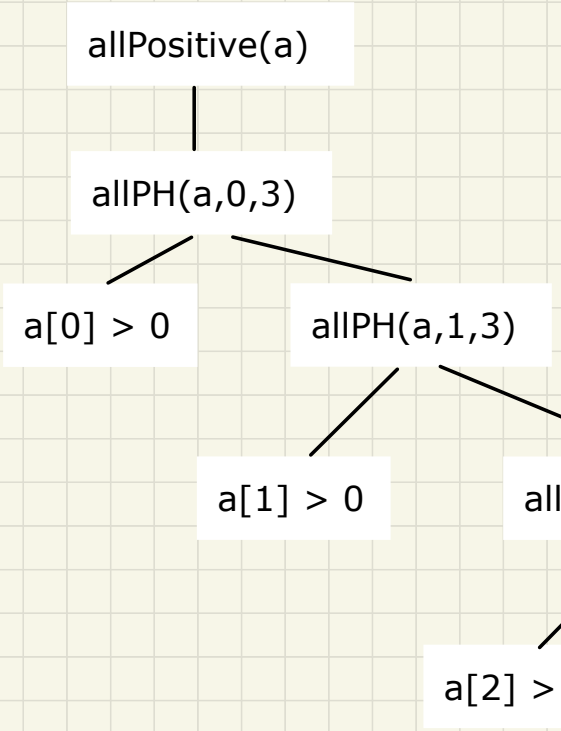

```java
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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[3]>0

Tracing Recursion: allPositive

Say a = {5,3,-2,9}




```
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);
  }
}
```

Problem: Are Numbers Sorted?

Say a = {} True

Say a = {4} True

Say a = {3,6,6,7}

isSorted(a, 0, 3)

a[0] ≤ a[1] && isSorted(a, 1, 3)

3 ≤ 6

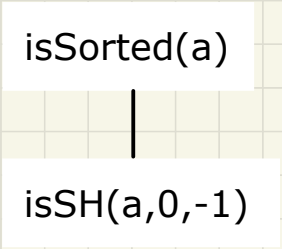
Say a = {3,6,5,7}

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

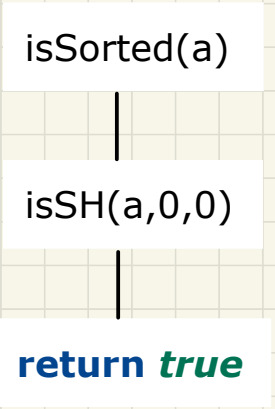
Say a = {}




```
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

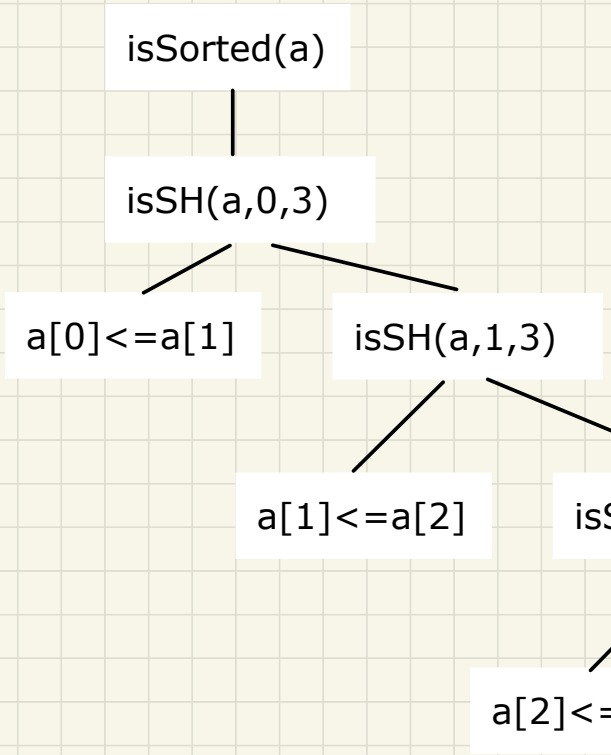
Say a = {4}




```
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

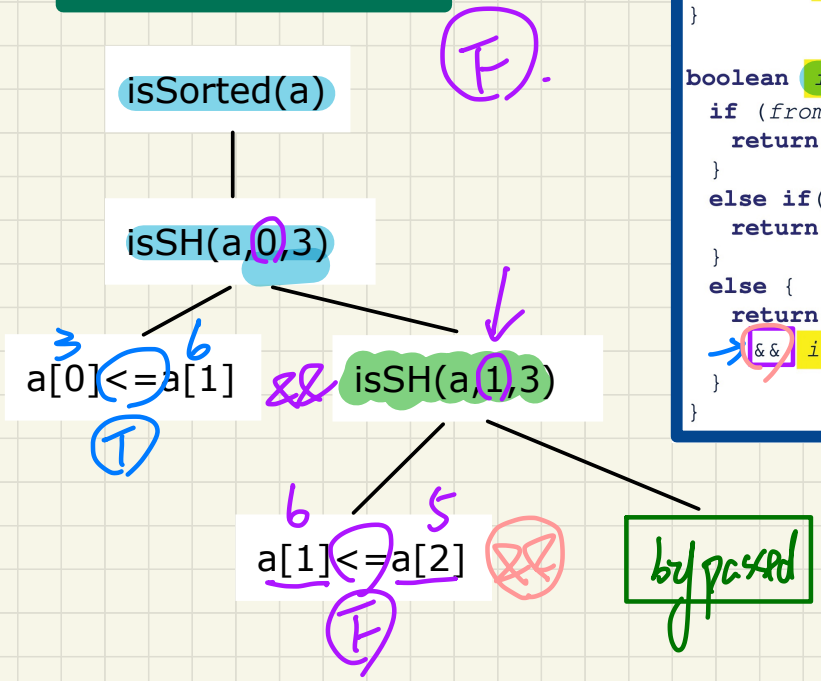
Say a = {3,6,6,7}




```
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

Say a = {3,6,5,7}




```
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);
  }
}
```