

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.

occ("abca", 'a')

isp("racecar")

Palindrome

↳ r == r && isp(aceca)

"racecar"

"aracecars"

"raceacar"

Reversal

reverseOf("abcd") → strictly smaller subproblem

reverseOf(bcd) + "a"
dcba

Number of Occurrences

occ("abca", 'a')

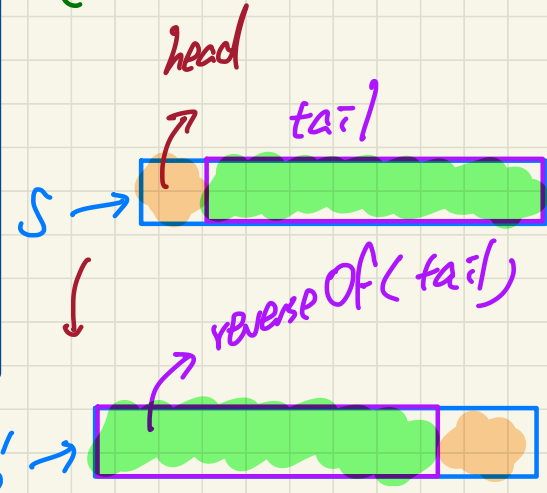
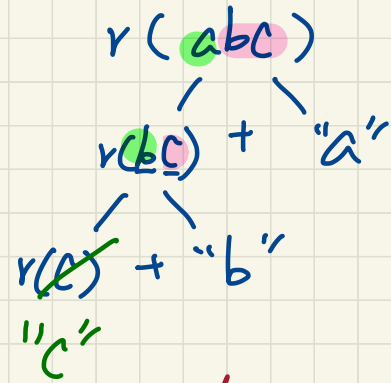
↳ 1 + occ(bca)

occ("abca", 'b')

↳ 0 + occ(bca)

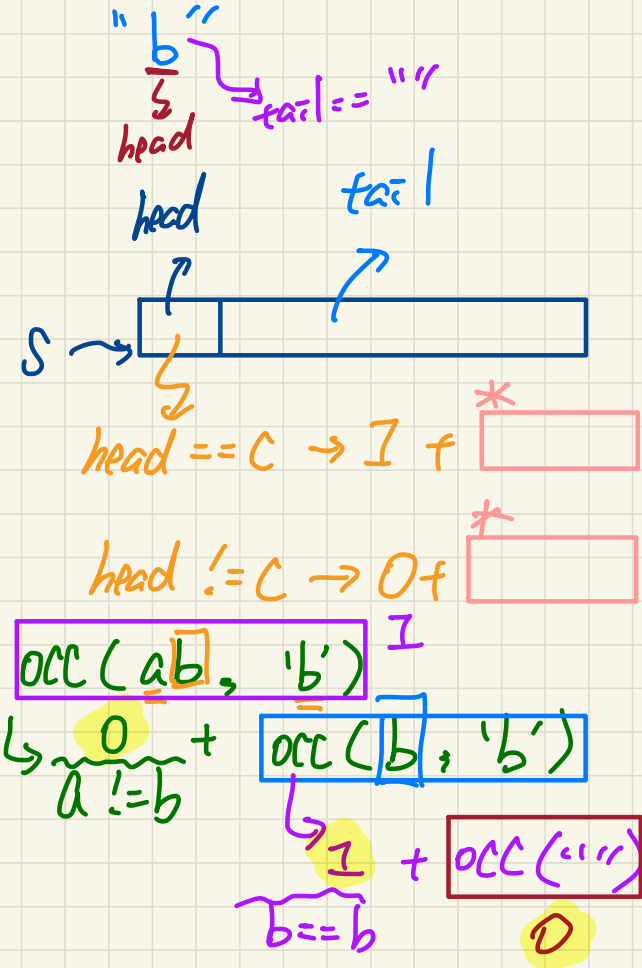
Problem: Reverse of a String

```
String reverseOf (String s) {  
    if(s.isEmpty()) { /* base case 1 */  
        return "";  
    }  
    else if(s.length() == 1) { /* base case 2 */  
        return s;  
    }  
    else { /* recursive case */  
        String tail = s.substring(1, s.length());  
        String reverseOfTail = reverseOf(tail);  
        char head = s.charAt(0);  
        return reverseOfTail + head;  
    }  
}
```



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);  
    }  
}
```

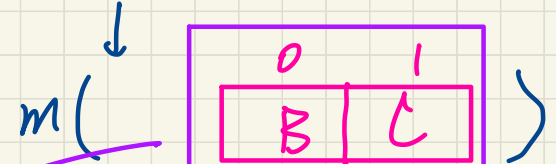
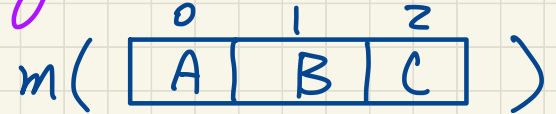
↪ recursive call on a new, strictly smaller array

Say $a_1 = \{\}$, consider $m(a_1)$

↪ base case # 1

Say $a_2 = \{A, B, C\}$, consider $m(a_2)$

time & space wasted to create these sub-arrays



↪ base case # 2

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) } }  
}
```

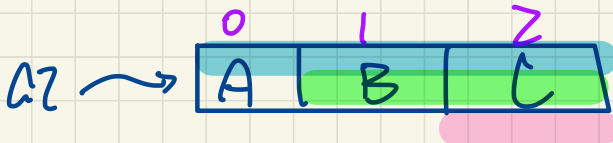
call by value

$0 - 1 = -1$

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

↳ base case # 1

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



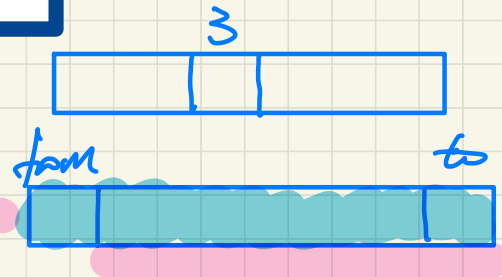
base case
2

$m(a2, 0, 2)$

↓
 $m(a2, 1, 2)$

↓
 $m(a2, 2, 2)$

for non-empty range: from ≤ to
form a range of elements to look at in the current method call



Problem: Are All Numbers Positive?

Say $a = \{\}$

$\forall x \in \emptyset. P(x) \equiv \text{True}$] there's no witness to show some violation

Say $a = \{4\}$

$\exists x \in \emptyset. P(x) \equiv \text{False}$] there's no witness to show some satisfaction

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

$\text{allP}(a, 0, 3)$ $\rightarrow a.length - 1$

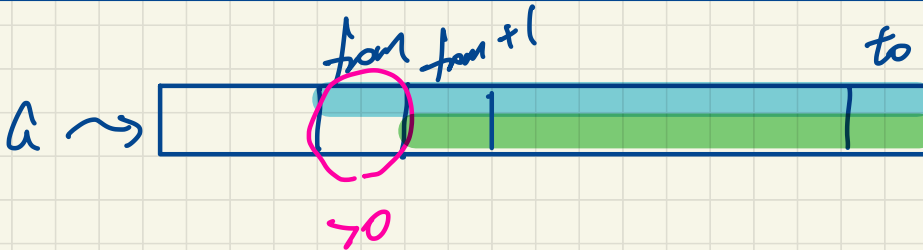
$\hookrightarrow a[0] > 0$ $\boxed{\text{xx}}$ $\text{allP}(a, 1, 3)$

$\nexists a[0] \leq 0$
 $\hookrightarrow$ by prev RHS

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

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);  
    }  
}
```



recursive
helper
method

public

private

Tracing Recursion: `allPositive`

Say `a = {}`

`allPositive(a)`

`allPH(a,0,-1)`

```
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}

allPositive(a)

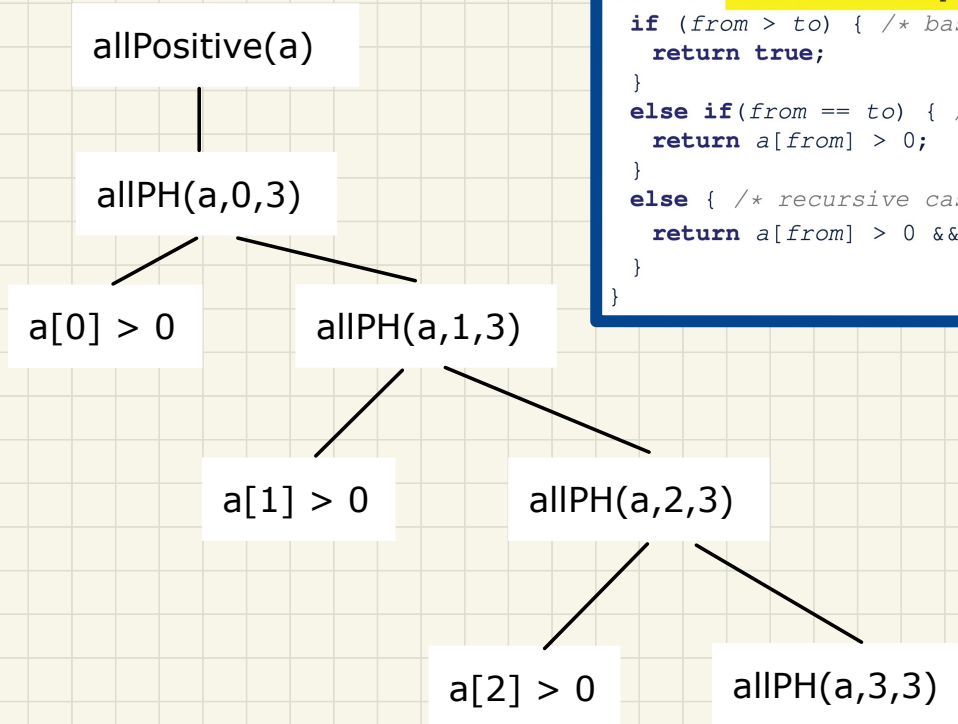
allPH(a,0,0)

a[0] > 0

```
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**

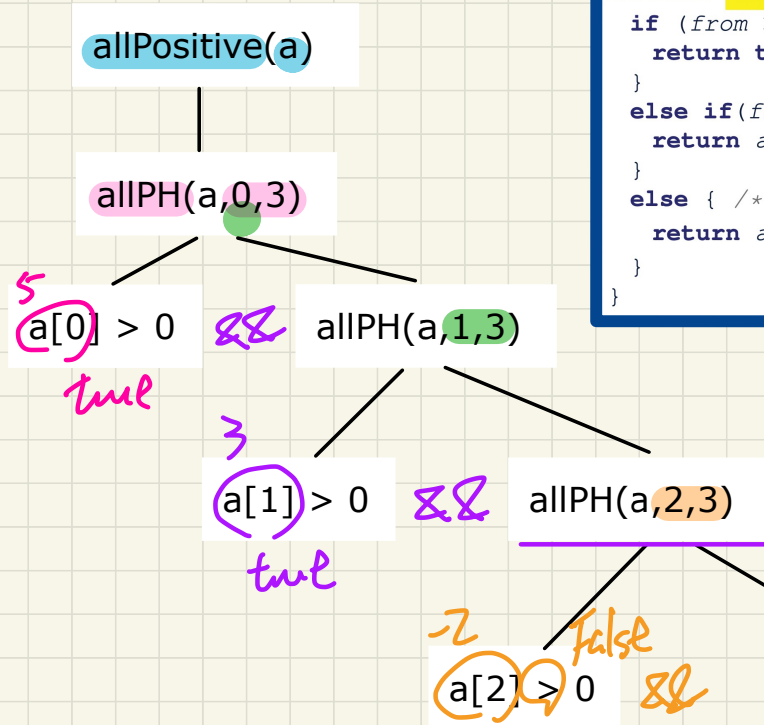
Say $a = \{4, 7, 3, 9\}$



```
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 $a = \{5, 3, -2, 9\}$



```
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);  
    }  
}
```

∴ short-circuit
Evaluation

bypassed

non-ascending ?

non-descending

i	j

$a[i]$

$a[j]$

$$\neg (a[i] > a[j])$$

$$\equiv a[i] \leq a[j]$$

Problem: Are Numbers Sorted?

Say $a = \{\}$

Say $a = \{4\}$

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

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

$isSorted(a, \overset{from}{0}, 3)$

$a[0] \leq a[1] \&\& isSorted(a, 1, 3)$

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 $a = \{\}$

isSorted(a)

isSH(a,0,-1)

```
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 $a = \{4\}$

isSorted(a)

isSH(a,0,0)

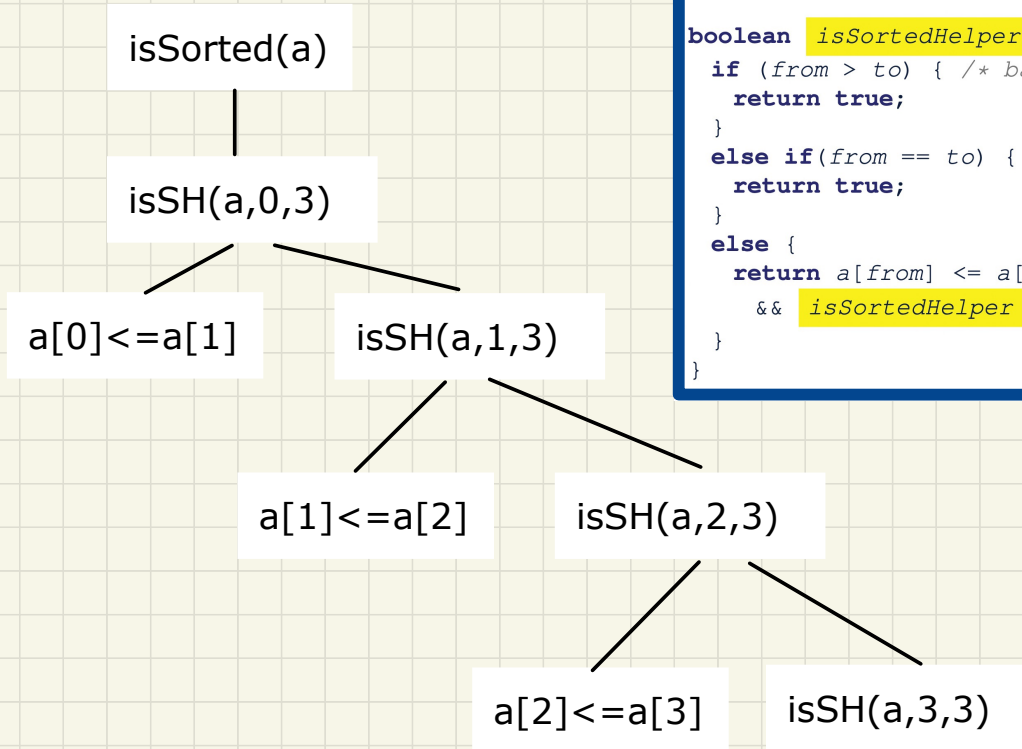
return **true**

```
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}

isSorted(a)

isSH(a, 0, 3)

a[0] <= a[1]

isSH(a, 1, 3)

a[1] <= a[2]

6 <= 5
False

```
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);  
    }  
}
```

bypassed