

Lecture 23 - Nov 30

Recursion

***Tracing Recursions: Faibonacci
Recursions on Strings: Reverse
Recursions on Arrays***

Recursive Solution: Fibonacci Numbers

$$F = \overset{\cdot}{1}, \overset{\cdot}{1}, 2, 3, 5, 8, \overset{F_7}{13}, \overset{F_8}{21}, \overset{F_9}{34}, 55, 89, \dots$$

Base Cases

$$F_1 = 1$$

$$F_2 = 1$$

Recursive Cases

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

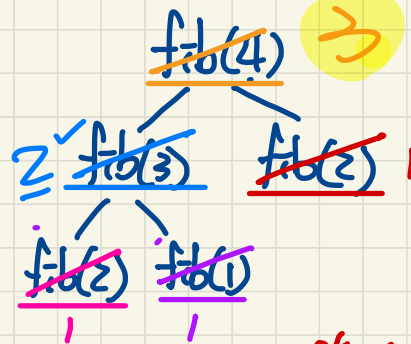
$n > 2$ strictly smaller than n

solved recursively by two recursive calls

$$F_9 = F_7 + F_8$$

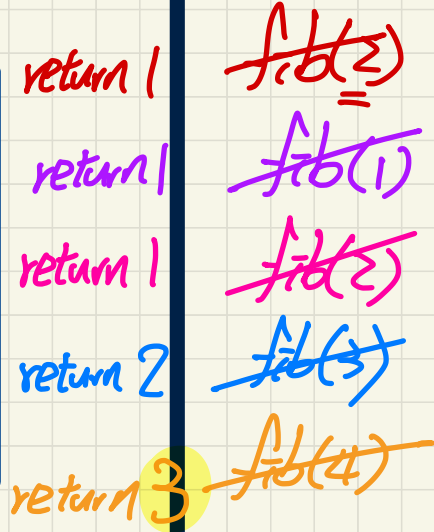
Recursive Solution in Java: Fibonacci Numbers

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



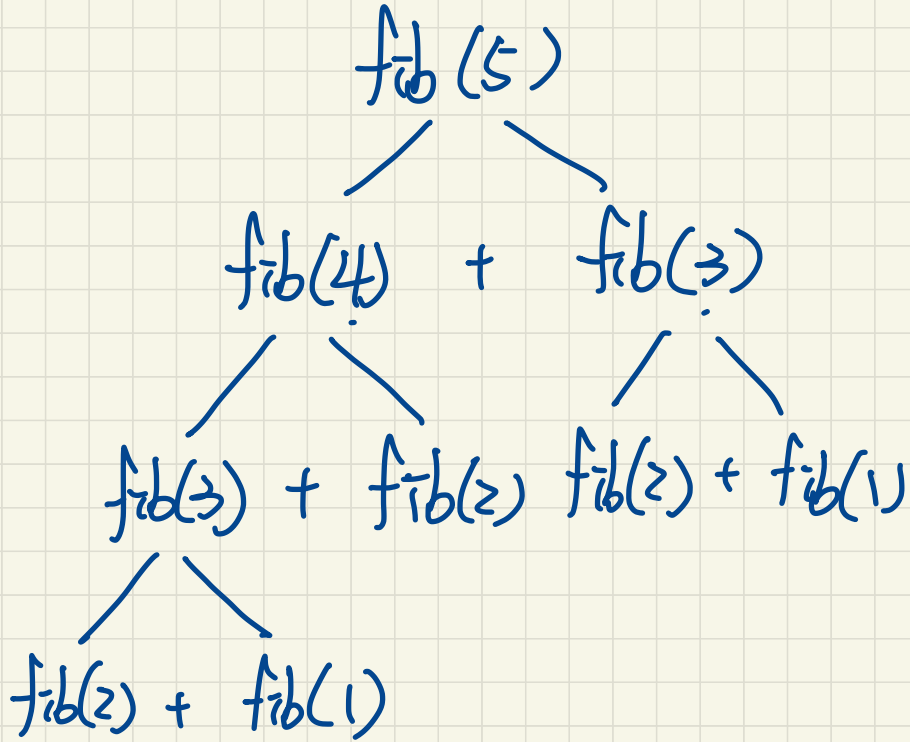
```
int fib (int n) {
    int result;
    if(n == 1) { /* base case */ result = 1; }
    else if(n == 2) { /* base case */ result = 1; }
    else { /* recursive case */
        result = fib (n - 1) + fib (n - 2);
    }
    return result;
}
```

$2 \text{ fib}(3) + \text{fib}(2) = 3$
 $\text{fib}(2) + \text{fib}(1) = 2$



Example: fib(4)

Runtime Stack



Recursions on Strings

Palindrome

→ "racecar" → T
"aracecars" → F
"raceacar" → F



strictly smaller problem

Reversal

"abcd"

"dcba"

reverse of

strictly smaller problem

solution strictly to smaller prob.

Number of Occurrences

"abca"

'a'

$$2 = 1 + 1$$

'b'

$$1 = 0 + 1$$

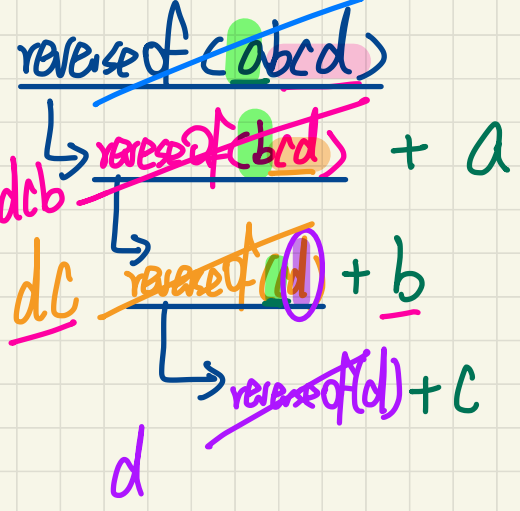
of occurrences of the char in tail of input string.

→ the char equal to the head of string.

Problem: Reverse of a String

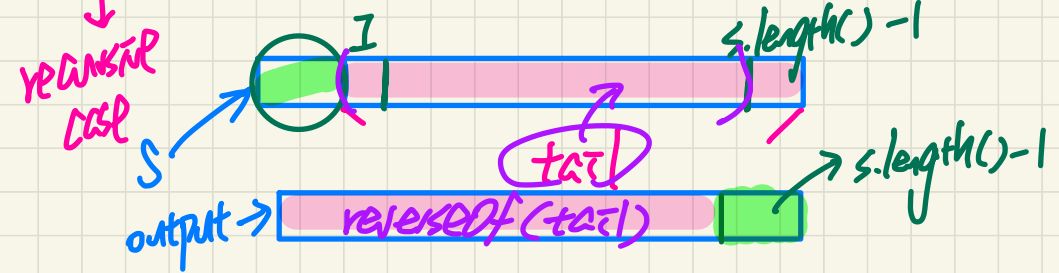
base cases

dcba



```
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;  
    }  
}
```

↪ recursive call to solve a strictly smaller problem.



Problem: Number of Occurrences

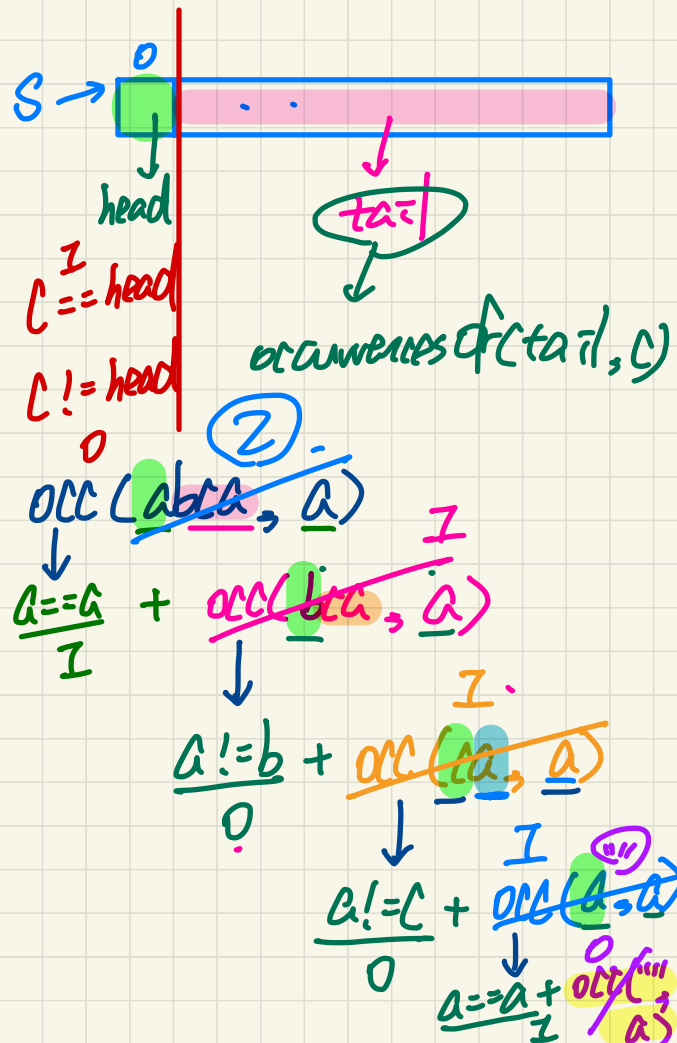
```

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);
        }
    }
}
    
```

→ base case

← recursive case

what if s is "a"?
↳ ""



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-1]; }  
        m(sub) } }  
}
```

base cases (green arrow pointing to the if/else if conditions)

RECURSIVE CASE (pink arrow pointing to the else block)

$i-1$ (pink arrow pointing to the index in the for loop)

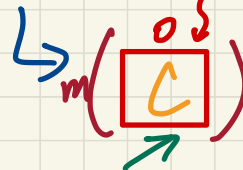
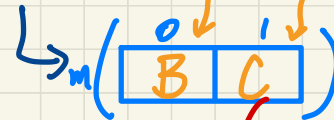
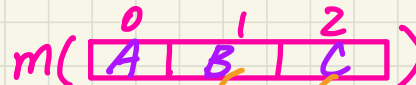
i (pink arrow pointing to the index in the array access)

$sub[0] = a[1]$ (orange arrow pointing to the assignment in the for loop)

$m(sub)$ (blue arrow pointing to the recursive call)

Say $a_1 = \{\}$ consider $m(a_1)$ → *execute the base case*

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



not space-efficient
(for each v.c., a new array is created)

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) } }

```

Empty array

array of length 1.

base cases

recursive case

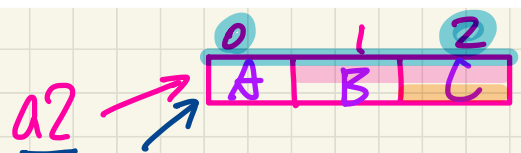
$[0, -1] \rightarrow$ empty range.

Say $a_1 = \{\}$, consider $m(a_1, 0, a_1.length - 1)$

\downarrow min index \downarrow max index
 $\rightarrow m(a, 0, -1)$

from to

Say $a_2 = \{A, B, C\}$, consider $m(a_2, 0, a_2.length - 1)$



$m(a_2, 0, 2)$

$m(a_2, 1, 2)$

$m(a_2, 2, 2)$

strictly smaller problem (last elem in array)

strictly smaller problem (elements from indices 1 to 2)

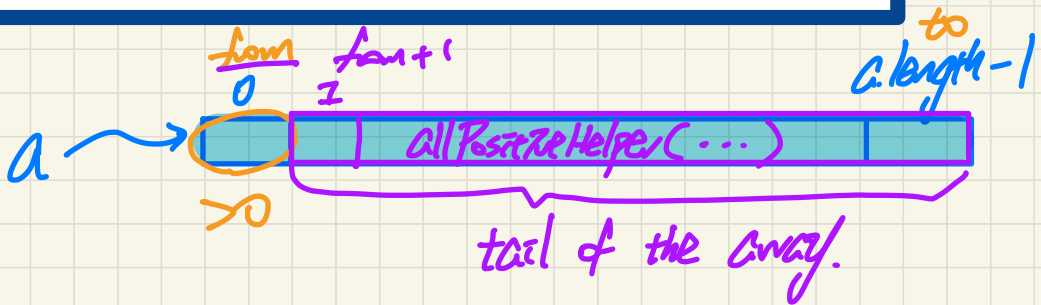
Problem: Are All Numbers Positive?

```
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);  
    }  
}
```

max index
max index
recursive helper method

base cases

recursive case



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

{4}

a.length - 1

True

4

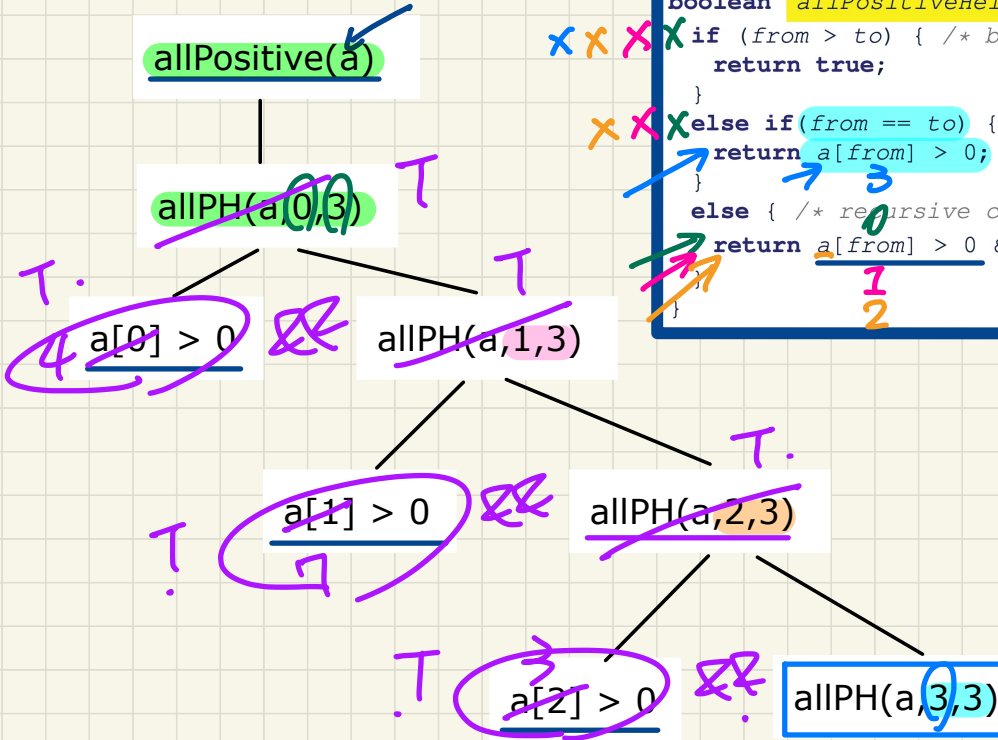
from
to ?

```
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}



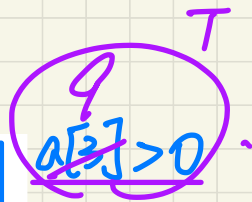
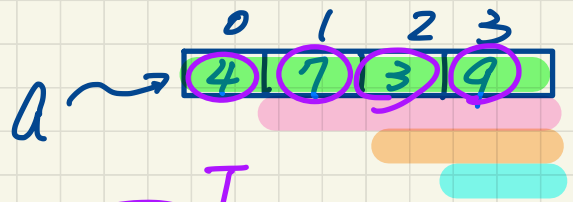
```

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);
    }
}
  
```

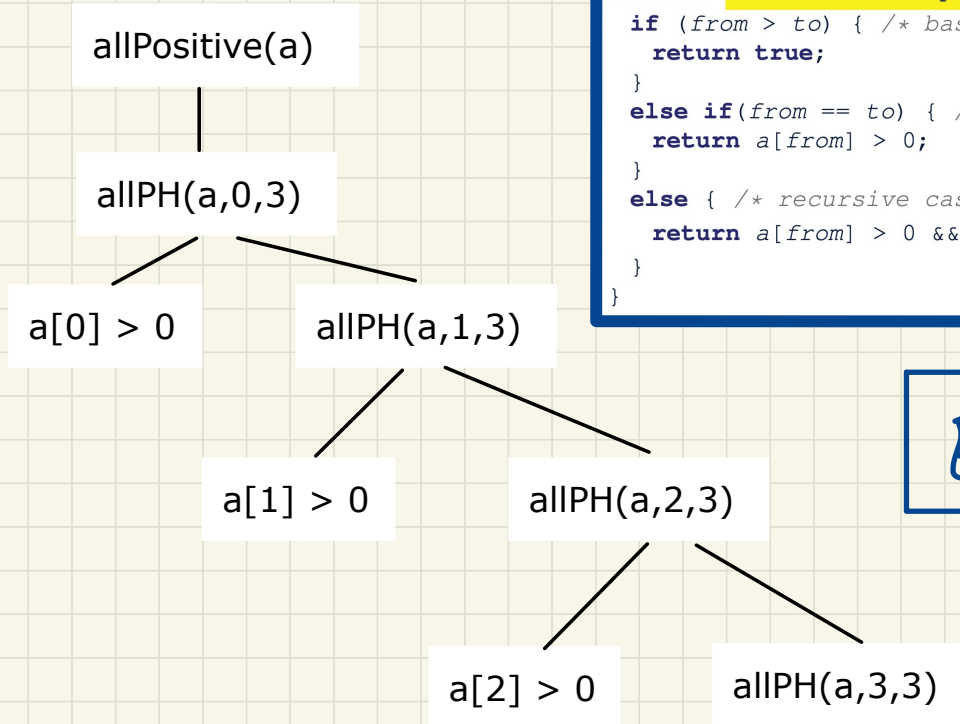
Handwritten annotations on the code:

- Method call: allPositive(a) with 'T' above 'a' and '3' above 'a.length - 1'.
- Base case 1: from > to (crossed out with 'XXXX').
- Base case 2: from == to (crossed out with 'XXX').
- Recursive case: from + 1 (crossed out with '0').
- Array indices: 0, 1, 2, 3 written above the array elements in the tree.
- Array elements: 4, 7, 3, 9 written below the array indices.



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);  
    }  
}
```

Exercise: Trace!

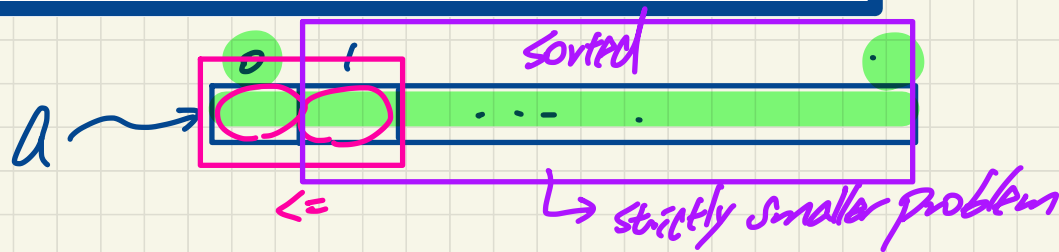
Problem: Are Numbers Sorted?

```
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);  
    }  
}
```

recursive helper method.

base cases

recursive case



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

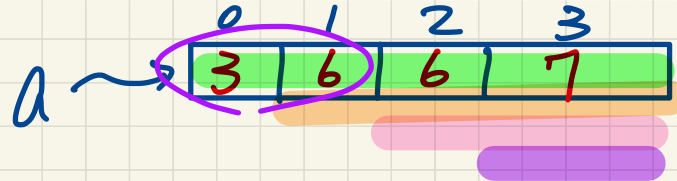
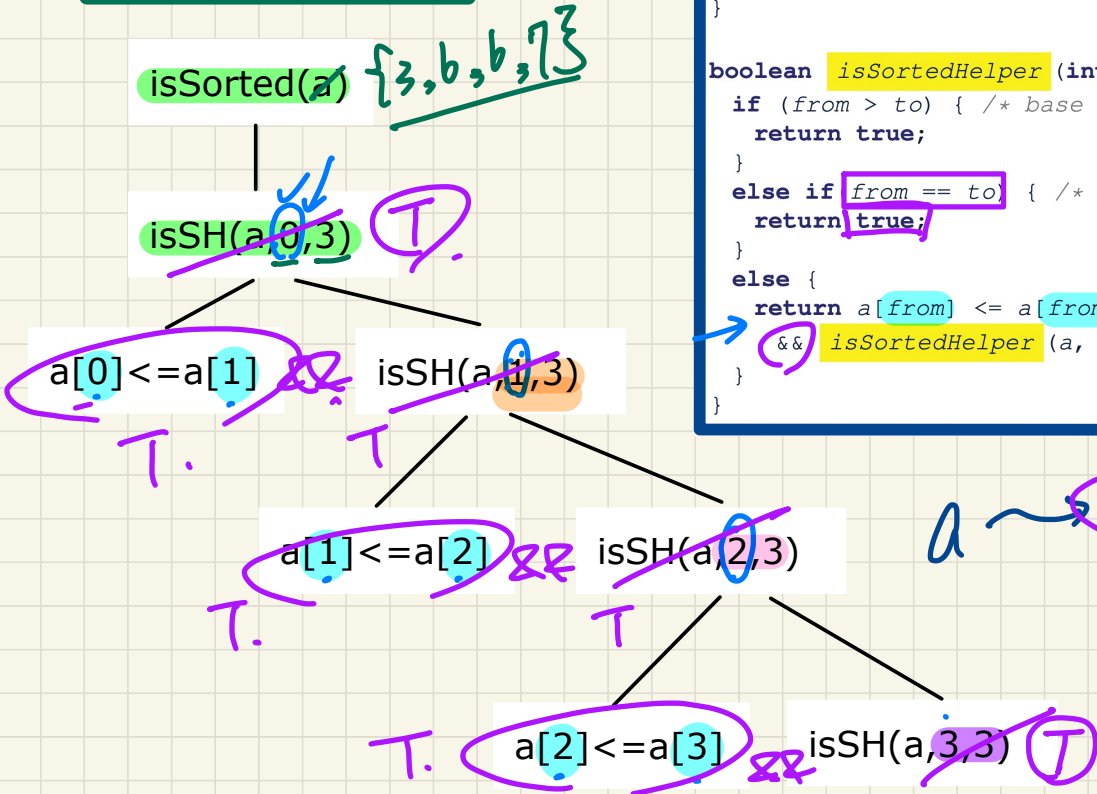
{4}

```
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);  
    }  
}
```

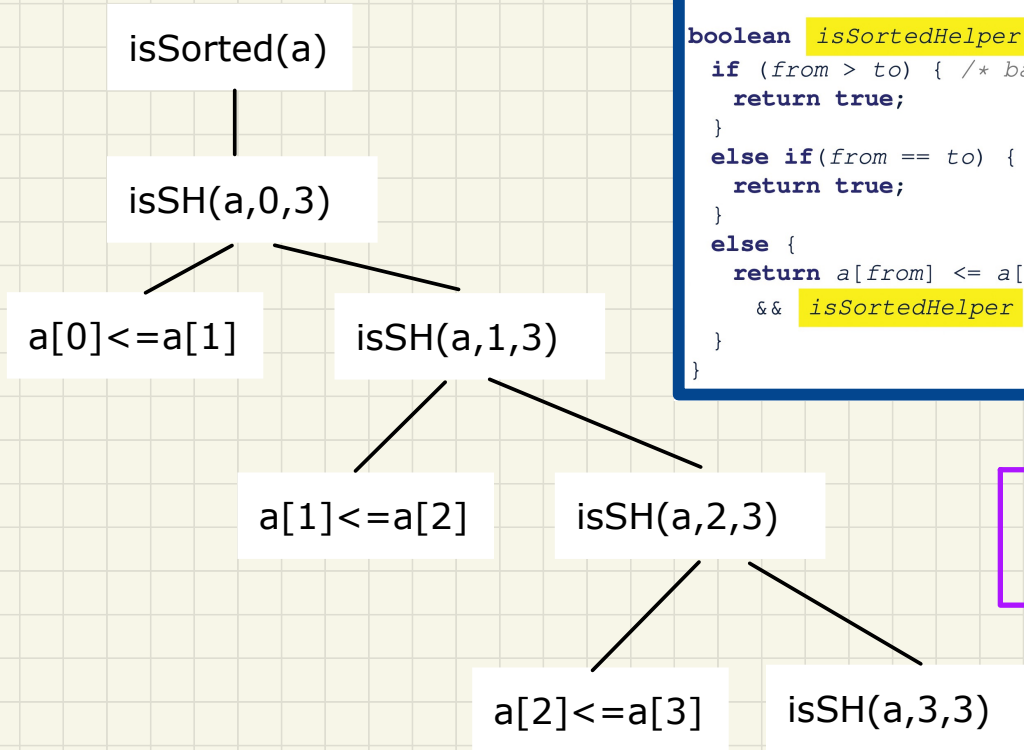


Tracing Recursion: isSorted

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

(F)

```
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);  
    }  
}
```



EXERCISE: TRACE