Lecture 23 - Nov 30

Recursion

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

Recursive Solution: Fibonacci Numbers

F_7 F_8 F_9

F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Base Cases

$F_1 = 1$

$F_2 = 1$

Recursive Cases

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

$n > 2$

$n-1$: strictly smaller than n

$n-2$: $< n$

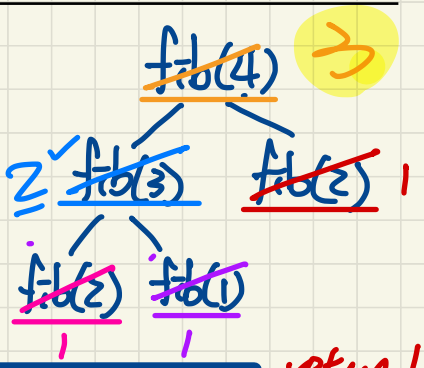
solved recursively by two recursive calls

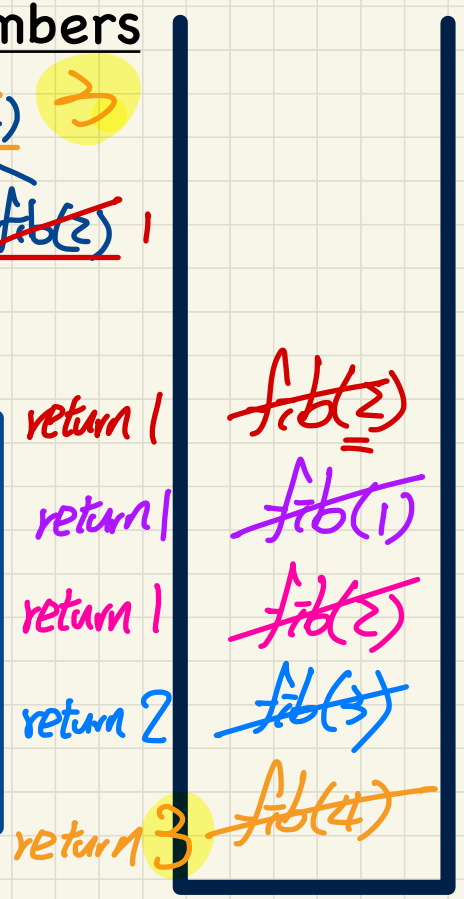
$F_9 = F_7 + F_8$.

Recursive Solution in Java: Fibonacci Numbers

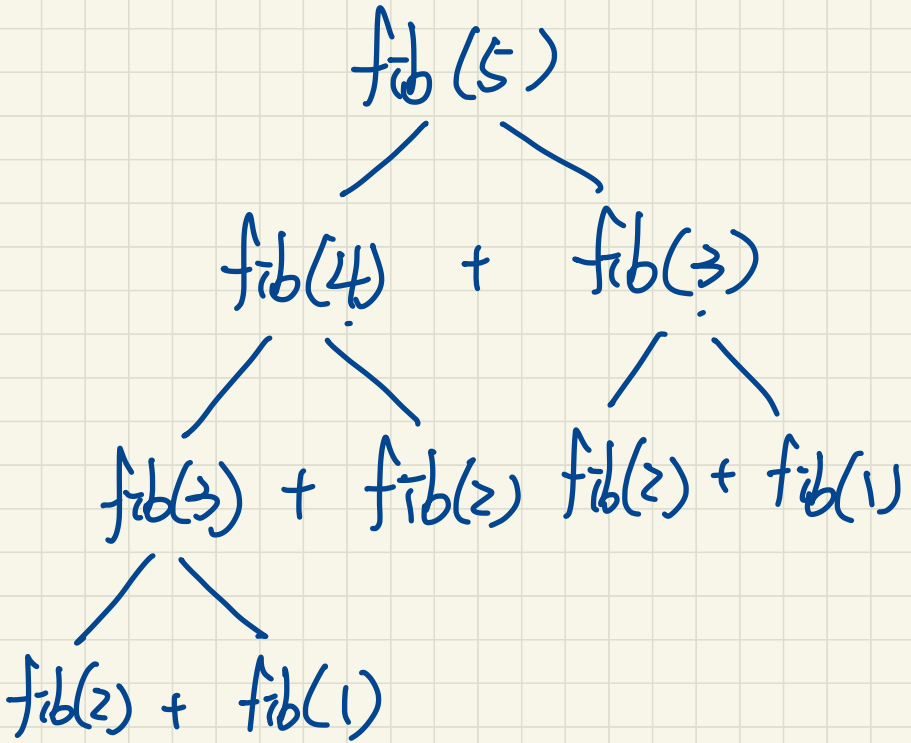
$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
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;
}
```



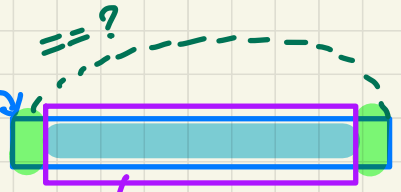


Example: fib(4)


fib(5)
fib(4) + fib(3)
fib(3) + fib(2) fib(2) + fib(1)
fib(2) + fib(1)

Recursions on Strings

Palindrome

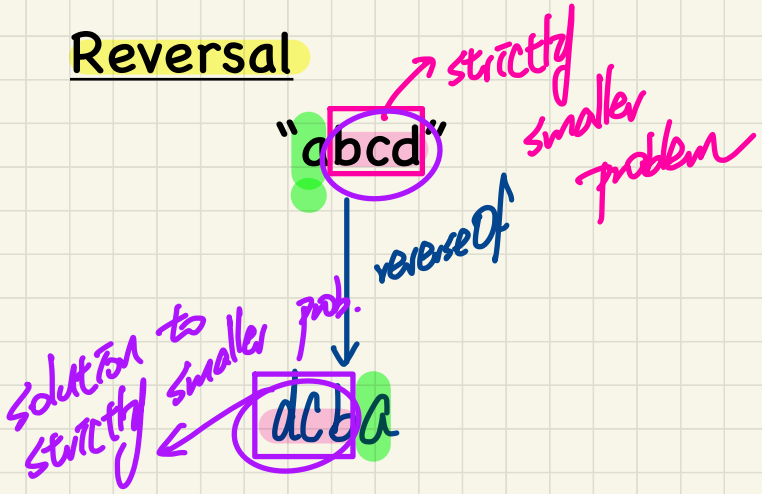


"racecar" → T

"aracecars" → F

"raceacar" → F

Reversal



Number of Occurrences

"abca"

'a' 2 = 1 + 1

'b' 1 = 0 + 1

# of occurrences of the char in tail of input string.

is the char equal to the head of string

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;
  }
}
```

base cases

recursive call to solve a strictly smaller problem.

dcba

reverseOf(abcd)

reverseOf(bcd) + a

reverseOf(cd) + b

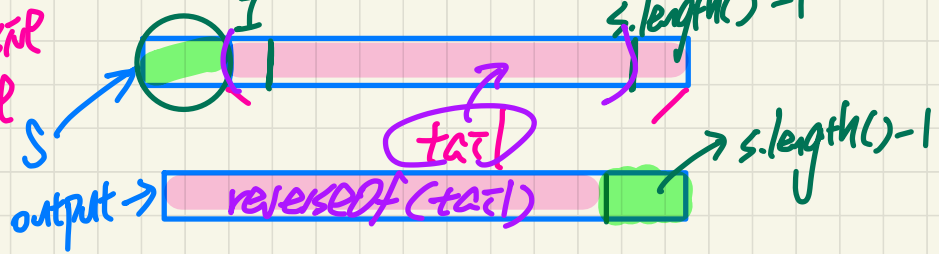
reverseOf(d) + c

d

dcb

dc

recursive case



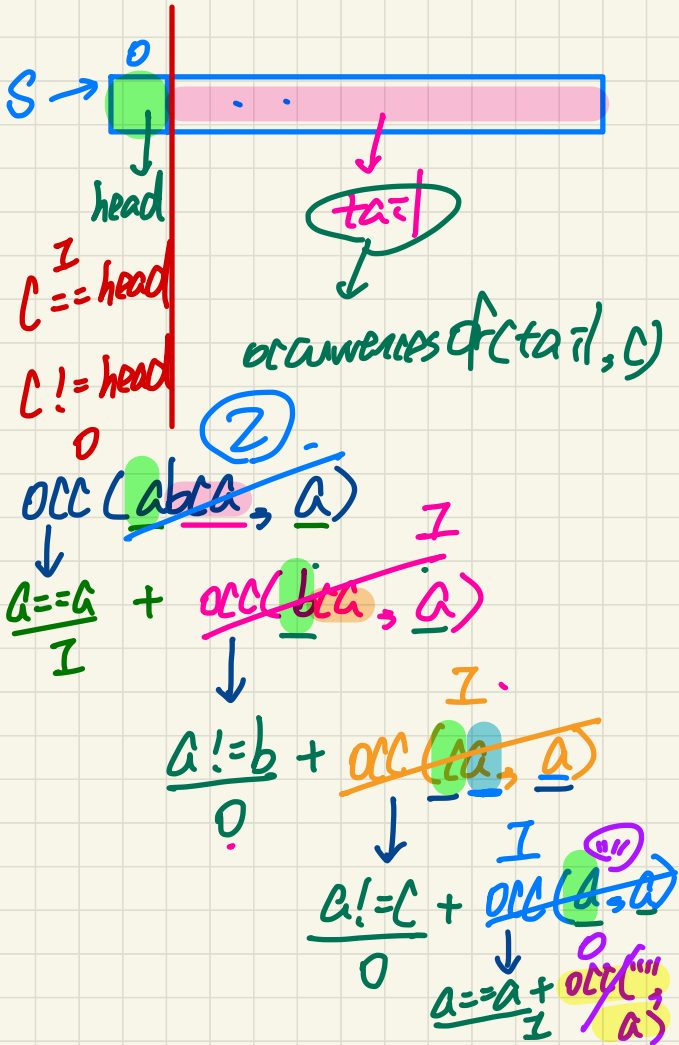
Problem: Number of Occurrences

```
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"? → "" (empty)



Recursion on an Array: Passing new Sub-Arrays

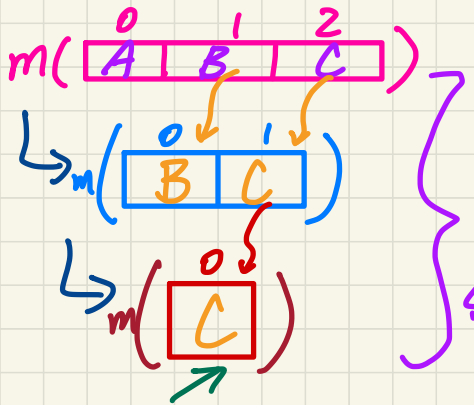
```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i-1] = a[i]; }
    m(sub); } }
```

base cases

recursive case

Say a1 = {} consider m(a1) → execute the base case

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



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) } }
```

array of length 1.

base cases

recursive case

empty array.

[0, -1] → empty range.

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

m(a1, 0, -1)

from to

min index

max index

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

m(a2, 0, 2)

↳ m(a2, 1, 2)

↳ m(a2, 2, 2)

strictly smaller problem (elements from indices 1 to 2)

strictly smaller problem (last item in array).

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);
  }
}
```

min index

max index

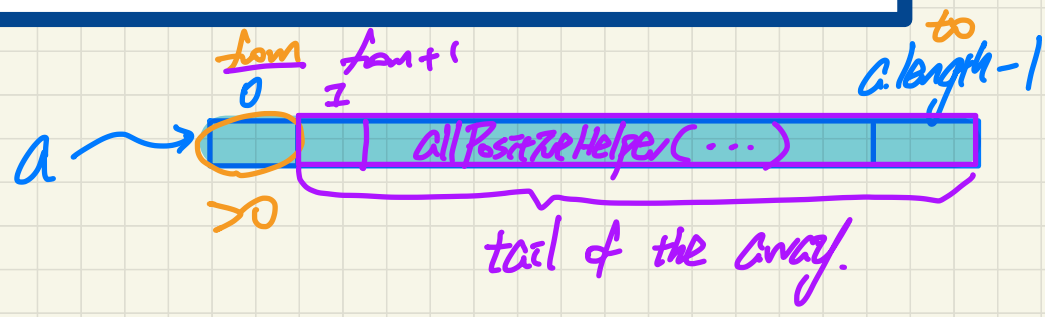
recursive helper method

empty array

array of length 1

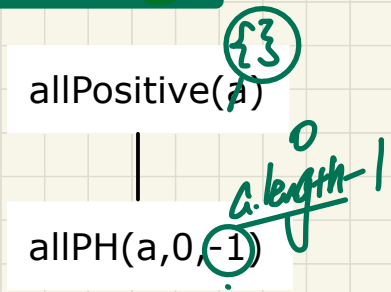
base cases

recursive case



Tracing Recursion: allPositive

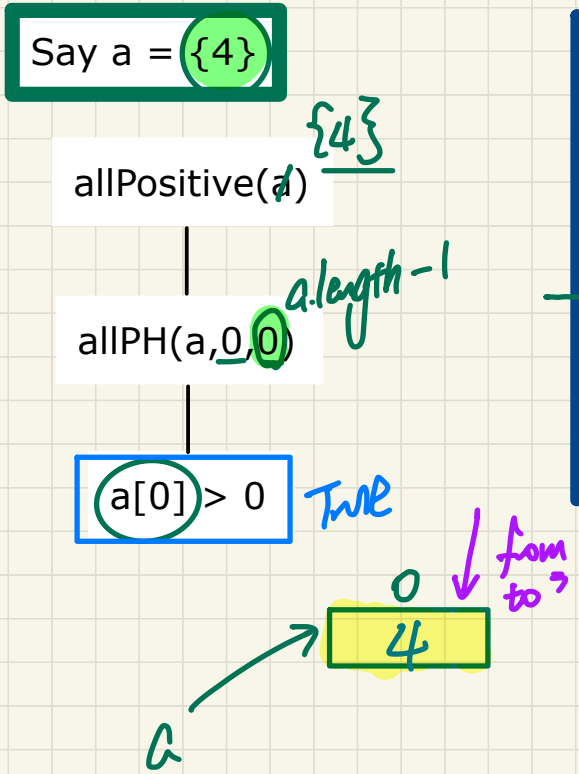
Say a = {}




```
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

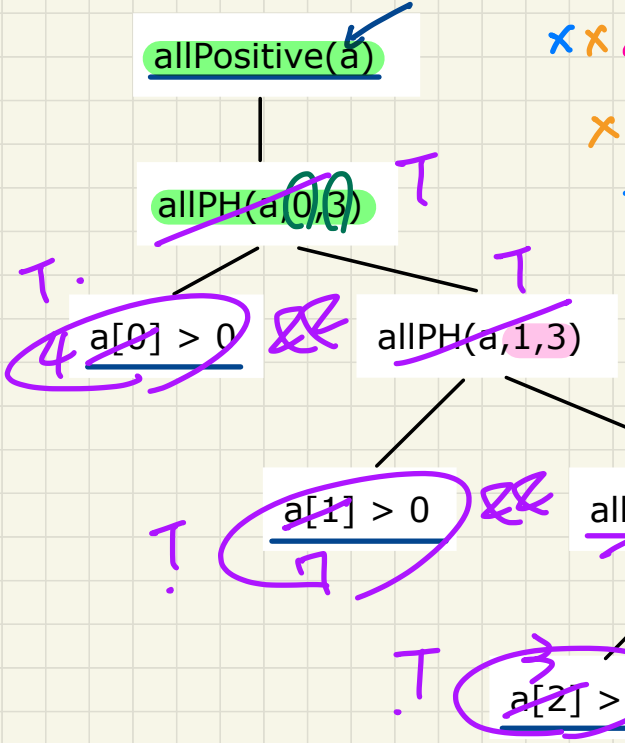



```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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);
  }
}
```

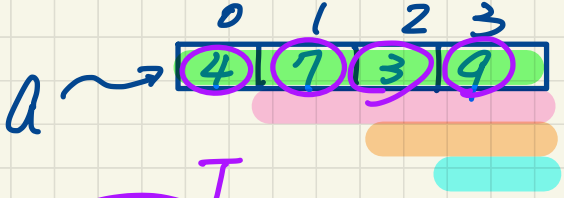
Tracing Recursion: allPositive

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



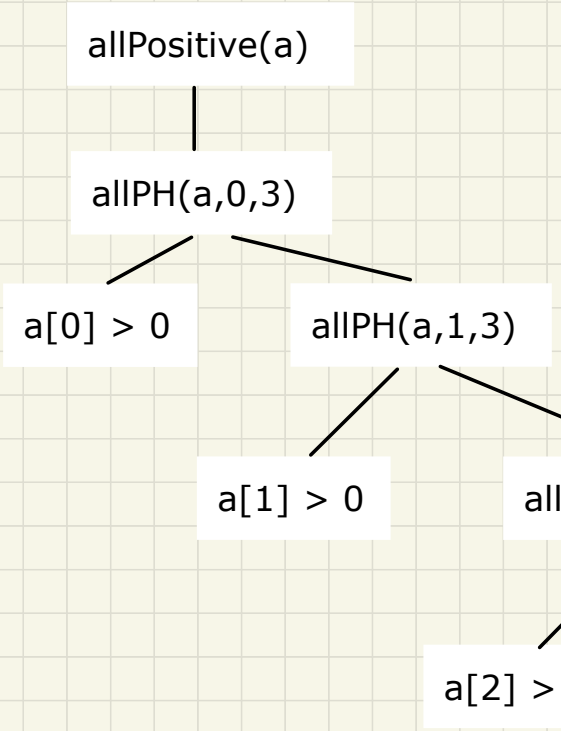

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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);
  }
}
```

Exercise: Trace!

Problem: Are Numbers Sorted?

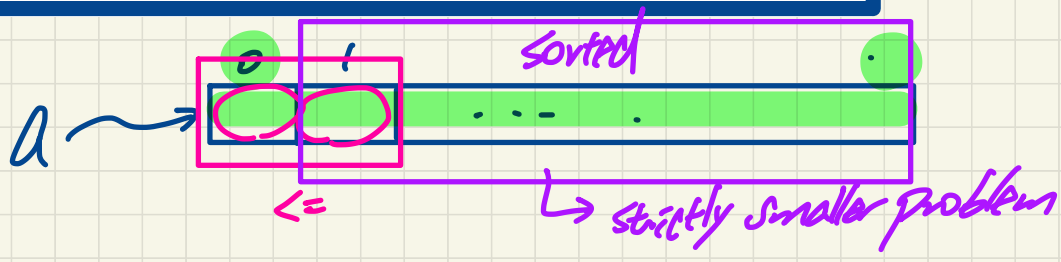
```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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)

|

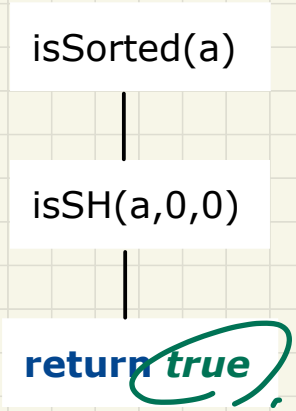
isSH(a,0,-1)

```
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}

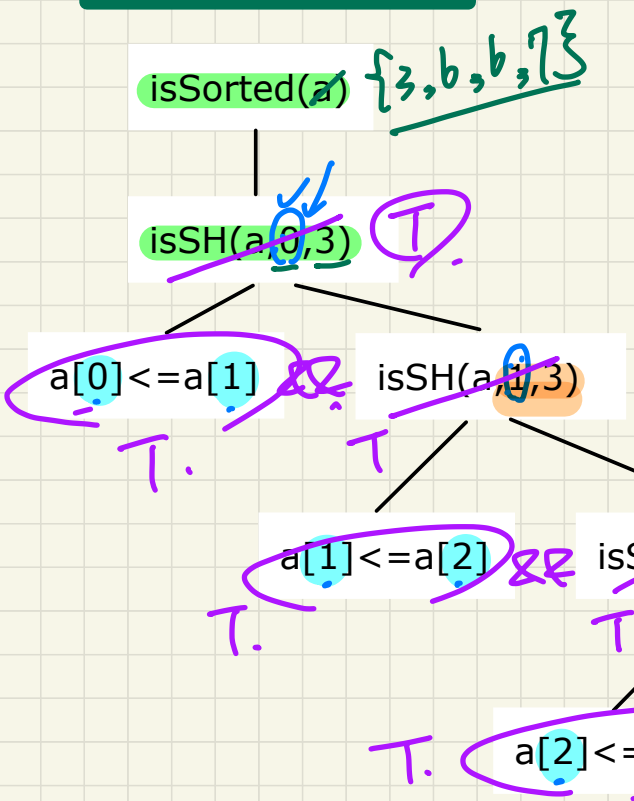



```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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);
  }
}
```

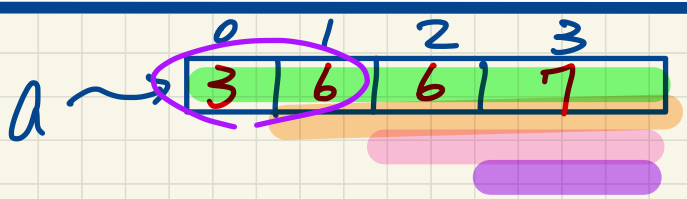
Tracing Recursion: isSorted

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



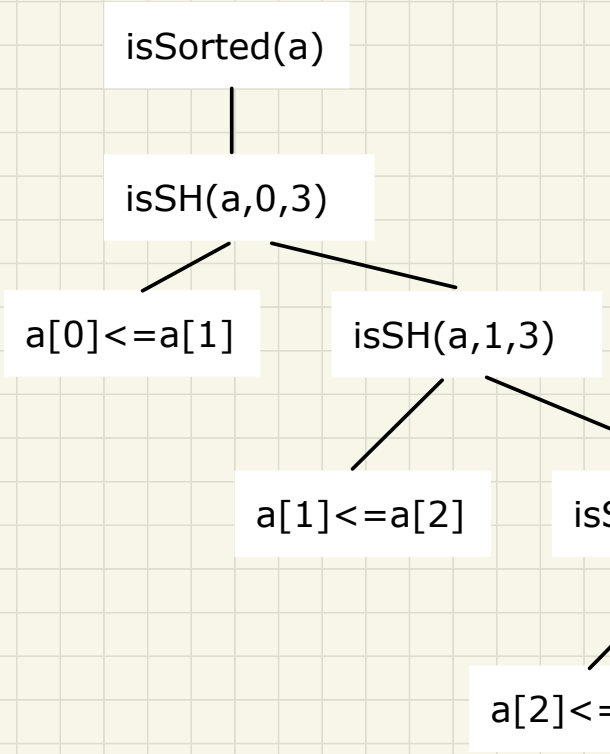

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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