Recursion

EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Recursion: Principle

- ***Recursion*** is useful in expressing solutions to problems that can be ***recursively*** defined:
  - ***Base* Cases:** Small problem instances immediately solvable.
  - ***Recursive* Cases:**
    - Large problem instances *not immediately solvable*.
    - Solve by reusing *solution(s) to strictly smaller problem instances*.
- Similar idea learnt in high school: [*mathematical induction*]
- Recursion can be easily expressed programmatically in Java:
  - In the body of a method m, there might be *a call or calls to m itself*.
  - Each such self-call is said to be a *recursive call*.
  - Inside the execution of $m(i)$, a recursive call $m(j)$ must be that $j < i$.

```
m (i) {
  ...
  m (j);/* recursive call with strictly smaller value */
  ...
}
```

Recursion: Factorial (1)

- Recall the formal definition of calculating the *n* factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

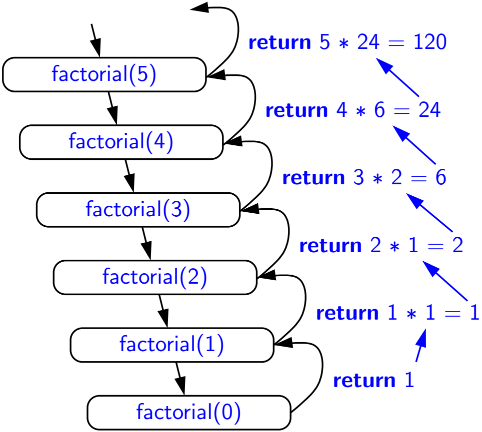
- How do you define the same problem *recursively*?

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

- To solve *n*!, we combine *n* and the solution to (*n - 1*)!.

```
int factorial (int n) {
  int result;
  if(n == 0) { /* base case */ result = 1; }
  else { /* recursive case */
    result = n * factorial (n - 1);
  }
  return result;
}
```

Recursion: Factorial (2)



Recursion: Factorial (3)

- When running *factorial(5)*, a *recursive call factorial(4)* is made. Call to *factorial(5)* suspended until *factorial(4)* returns a value.
- When running *factorial(4)*, a *recursive call factorial(3)* is made. Call to *factorial(4)* suspended until *factorial(3)* returns a value.

  . . .
- *factorial(0)* returns 1 back to *suspended call factorial(1)*.
- *factorial(1)* receives 1 from *factorial(0)*, multiplies 1 to it, and returns 1 back to the *suspended call factorial(2)*.
- *factorial(2)* receives 1 from *factorial(1)*, multiplies 2 to it, and returns 2 back to the *suspended call factorial(3)*.
- *factorial(3)* receives 2 from *factorial(1)*, multiplies 3 to it, and returns 6 back to the *suspended call factorial(4)*.
- *factorial(4)* receives 6 from *factorial(3)*, multiplies 4 to it, and returns 24 back to the *suspended call factorial(5)*.
- *factorial(5)* receives 24 from *factorial(4)*, multiplies 5 to it, and returns 120 as the result.

Recursion: Factorial (4)

- When the execution of a method (e.g., *factorial(5)*) leads to a nested method call (e.g., *factorial(4)*):
  - The execution of the current method (i.e., *factorial(5)*) is *suspended*, and a structure known as an *activation record* or *activation frame* is created to store information about the progress of that method (e.g., values of parameters and local variables).
  - The nested methods (e.g., *factorial(4)*) may call other nested methods (*factorial(3)*).
  - When all nested methods complete, the activation frame of the *latest suspended* method is re-activated, then continue its execution.
- What kind of data structure does this activation-suspension process correspond to? [LIFO Stack]

Tracing Recursion using a Stack

- When a method is called, it is ***activated*** (and becomes *active*) and *pushed* onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and *pushed* onto the stack.

  ⇒ The stack contains activation records of all *active* methods.
  - *Top* of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

  ⇒ The current point of execution is returned to the new *top* of stack (which was ***suspended*** and just became ***active***).
- Execution terminates when the stack becomes *empty*.

Recursion: Fibonacci (1)

Recall the formal definition of calculating the n_{th} number in a Fibonacci series (denoted as F_n), which is already itself recursive:

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

```
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;
}
```

Recursion: Fibonacci (2)

fib(5)

= {fib(5) = fib(4) + fib(3); push(fib(5)); *suspended*: ⟨fib(5)⟩; *active*: fib(4)}

fib(4) + fib(3)

= {fib(4) = fib(3) + fib(2); *suspended*: ⟨fib(4), fib(5)⟩; *active*: fib(3)}

(fib(3) + fib(2)) + fib(3)

= {fib(3) = fib(2) + fib(1); *suspended*: ⟨fib(3), fib(4), fib(5)⟩; *active*: fib(2)}

((fib(2) + fib(1)) + fib(2)) + fib(3)

= {fib(2) returns 1; *suspended*: ⟨fib(3), fib(4), fib(5)⟩; *active*: fib(1)}

((1 + fib(1)) + fib(2)) + fib(3)

= {fib(1) returns 1; *suspended*: ⟨fib(3), fib(4), fib(5)⟩; *active*: fib(3)}

((1 + 1) + fib(2)) + fib(3)

= {fib(3) returns 1 + 1; pop(); *suspended*: ⟨fib(4), fib(5)⟩; *active*: fib(2)}

(2 + fib(2)) + fib(3)

= {fib(2) returns 1; *suspended*: ⟨fib(4), fib(5)⟩; *active*: fib(4)}

(2 + 1) + fib(3)

= {fib(4) returns 2 + 1; pop(); *suspended*: ⟨fib(5)⟩; *active*: fib(3)}

3 + fib(3)

= {fib(3) = fib(2) + fib(1); *suspended*: ⟨fib(3),fib(5)⟩; *active*: fib(2)}

3 + (fib(2) + fib(1))

= {fib(2) returns 1; *suspended*: ⟨fib(3), fib(5)⟩; *active*: fib(1)}

3 + (1 + fib(1))

= {fib(1) returns 1; *suspended*: ⟨fib(3), fib(5)⟩; *active*: fib(3)}

3 + (1 + 1)

= {fib(3) returns 1 + 1; pop() ; *suspended*: ⟨fib(5)⟩; *active*: fib(5)}

3 + 2

= {fib(5) returns 3 + 2; *suspended*: ⟨⟩}

5

Java Library: String

```
public class StringTester {
  public static void main(String[] args) {
    String s = "abcd";
    System.out.println(s.isEmpty()); /* false */
    /* Characters in index range [0, 0) */
    String t0 = s.substring(0, 0);
    System.out.println(t0); /* "" */
    /* Characters in index range [0, 4) */
    String t1 = s.substring(0, 4);
    System.out.println(t1); /* "abcd" */
    /* Characters in index range [1, 3) */
    String t2 = s.substring(1, 3);
    System.out.println(t2); /* "bc" */
    String t3 = s.substring(0, 2) + s.substring(2, 4);
    System.out.println(s.equals(t3)); /* true */
    for(int i = 0; i < s.length(); i ++) {
      System.out.print(s.charAt(i));
    }
    System.out.println();
  }
}
```

Recursion: Palindrome (1)

Problem: A palindrome is a word that reads the same forwards and backwards. Write a method that takes a string and determines whether or not it is a palindrome.

```
System.out.println(isPalindrome(""));  true
System.out.println(isPalindrome("a"));  true
System.out.println(isPalindrome("madam"));  true
System.out.println(isPalindrome("racecar"));  true
System.out.println(isPalindrome("man"));  false
```

Base Case 1: Empty string $\longrightarrow$ Return *true* immediately.

Base Case 2: String of length 1 $\longrightarrow$ Return *true* immediately.

Recursive Case: String of length ≥ 2 $\longrightarrow$

- 1st and last characters match, **and**
- *the rest (i.e., middle) of the string* *is a palindrome*.

Recursion: Palindrome (2)

```
boolean isPalindrome (String word) {
  if(word.length() == 0 || word.length() == 1) {
    /* base case */
    return true;
  }
  else {
    /* recursive case */
    char firstChar = word.charAt(0);
    char lastChar = word.charAt(word.length() - 1);
    String middle = word.substring(1, word.length() - 1);
    return
        firstChar == lastChar
        /* See the API of java.lang.String.substring. */
        && isPalindrome (middle);
  }
}
```

Recursion: Reverse of String (1)

Problem: The reverse of a string is written backwards. Write a method that takes a string and returns its reverse.

```
System.out.println(reverseOf(""));  /* "" */
System.out.println(reverseOf("a"));  "a"
System.out.println(reverseOf("ab"));  "ba"
System.out.println(reverseOf("abc"));  "cba"
System.out.println(reverseof("abcd"));  "dcba"
```

Base Case 1: Empty string $\longrightarrow$ Return *empty string*.

Base Case 2: String of length 1 $\longrightarrow$ Return *that string*.

Recursive Case: String of length ≥ 2 $\longrightarrow$

1) Head of string (i.e., first character)
2) Reverse of the tail of string (i.e., all but the first character)

Return the concatenation of **1)** and **2)**.

Recursion: Reverse of a String (2)

```
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;
  }
}
```

Recursion: Number of Occurrences (1)

Problem: Write a method that takes a string `s` and a character `c`, then count the number of occurrences of `c` in `s`.

```
System.out.println(occurrencesOf("", 'a'));  /* 0 */
System.out.println(occurrencesOf("a", 'a'));  /* 1 */
System.out.println(occurrencesOf("b", 'a'));  /* 0 */
System.out.println(occurrencesOf("baaba", 'a'));  /* 3 */
System.out.println(occurrencesOf("baaba", 'b'));  /* 2 */
System.out.println(occurrencesOf("baaba", 'c'));  /* 0 */
```

Base Case: Empty string ⟶ Return *0*.

Recursive Case: String of length ≥ 1 ⟶

1) Head of `s` (i.e., first character)
2) Number of occurrences of `c` in the tail of `s` (i.e., all but the first character)

If head is equal to `c`, return 1 + **2)**.
If head is not equal to `c`, return 0 + **2)**.

Recursion: Number of Occurrences (2)

```
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: All Positive (1)

Problem: Determine if an array of integers are all positive.

```
System.out.println(allPositive({})); /* true */
System.out.println(allPositive({1, 2, 3, 4, 5})); /* true */
System.out.println(allPositive({1, 2, -3, 4, 5})); /* false */
```

Base Case: Empty array ⟶ Return *true* immediately.

The base case is *true* ∵ we can *not* find a counter-example (i.e., a number *not* positive) from an empty array.

Recursive Case: Non-Empty array ⟶

- 1st element positive, **and**
- *the rest of the array* *is all positive*.

Exercise: Write a method `boolean somePostive(int[] a)` which *recursively* returns *true* if there is some positive number in `a`, and *false* if there are no positive numbers in `a`.

Hint: What to return in the base case of an empty array? [*false*]

∵ No witness (i.e., a positive number) from an empty array

Making Recursive Calls on an Array

- Recursive calls denote solutions to *smaller* sub-problems.
- *Naively*, explicitly create a new, smaller array:

```
void m(int[] a) {
  int[] subArray = new int[a.length - 1];
  for(int i = 1; i < a.length; i ++) { subArray[0] = a[i - 1]; }
  m(subArray) }
```

- For *efficiency*, we pass the same array **by reference** and specify the *range of indices* to be considered:

```
void m(int[] a, int from, int to) {
  if(from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

  - `m(a, 0, a.length - 1)` [Initial call; entire array]
  - `m(a, 1, a.length - 1)` [1st r.c. on array of size *a.length* – 1]
  - `m(a, 2, a.length - 1)` [2nd r.c. on array of size *a.length* – 2]

  ...

  - `m(a, a.length-1, a.length-1)` [Last r.c. on array of size 1]

Recursion: All Positive (2)

```
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);
  }
}
```

Recursion: Is an Array Sorted? (1)

Problem: Determine if an array of integers are sorted in a non-descending order.

```
System.out.println(isSorted({}));  true

System.out.println(isSorted({1, 2, 2, 3, 4}));  true

System.out.println(isSorted({1, 2, 2, 1, 3}));  false
```

Base Case: Empty array $\longrightarrow$ Return *true* immediately.

The base case is *true* $\because$ we can *not* find a counter-example (i.e., a pair of adjacent numbers that are *not* sorted in a non-descending order) from an empty array.

Recursive Case: Non-Empty array $\longrightarrow$

- 1st and 2nd elements are sorted in a non-descending order, **and**
- *the rest of the array*, starting from the 2nd element, *are sorted in a non-descending positive* .

Recursion: Is an Array Sorted? (2)

```java
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);
  }
}
```

Recursion: Sorting an Array (1)

Problem: Sort an array into a non-descending order, using the *selection-sort* strategy.

Base Case: Arrays of size 0 or 1 $\longrightarrow$ Completed immediately.

Recursive Case: Non-Empty array `a` $\longrightarrow$

Run an iteration from indices $i = 0$ to $a.length - 1$.
In each iteration:

- In index range $[i, a.length - 1]$, *recursively* compute the *minimum* element $\boxed{e}$.
- Swap $\boxed{a[i]}$ and $\boxed{e}$ if $e < a[i]$.

Recursion: Sorting an Array (2)

```
public static int getMinIndex (int[] a, int from, int to) {
  if(from == to) { return from; }
  else {
    int minIndexOfTail = getMinIndex(a, from + 1, to);
    if(a[from] < a[minIndexOfTail]) { return from; }
    else { return minIndexOfTail; }
  }
}
public static void selectionSort(int[] a) {
  if(a.length == 0 || a.length == 1) { /* sorted, do nothing */ }
  else {
    for(int i = 0; i < a.length; i ++) {
      int minIndex = getMinIndex (a, i, a.length - 1);
      /* swap a[i] and a[minIndex] */
      int temp = a[i];
      a[i] = a[minIndex];
      a[minIndex] = temp;
    }
  }
}
```

Recursion: Binary Search (1)

- **Searching Problem**

  Input: A number a and a *sorted* list of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

  Output: Whether or not a exists in the input list

- **An Efficient Recursive Solution**

  Base Case: Empty list $\longrightarrow$ *False*.

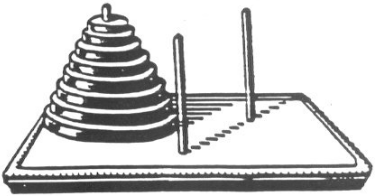
  Recursive Case: List of size ≥ 1 $\longrightarrow$

  - *Compare* the *middle* element against a.
    - All elements to the left of *middle* are $\leq a$
    - All elements to the right of *middle* are $\geq a$
  - If the *middle* element *is* equal to a $\longrightarrow$ *True*.
  - If the *middle* element *is not* equal to a:
    - If $a <$ *middle*, recursively find a on the left half.
    - If $a >$ *middle*, recursively find a on the right half.

Recursion: Binary Search (2)

```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchHelper (sorted, 0, sorted.length - 1, key);
}
boolean binarySearchHelper (int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element */
    return sorted[from] == key; }
  else {
    int middle = (from + to) / 2;
    int middleValue = sorted[middle];
    if(key < middleValue) {
      return binarySearchHelper (sorted, from, middle - 1, key);
    }
    else if (key > middleValue) {
      return binarySearchHelper (sorted, middle + 1, to, key);
    }
    else  { return true; }
  }
}
```

Tower of Hanoi: Specification



- *Given*: A tower of 8 disks, initially stacked in decreasing size on one of 3 pegs
- *Rules*:
  - Move only one disk at a time
  - Never move a larger disk onto a smaller one
- *Problem*: Transfer the entire tower to one of the other pegs.

Tower of Hanoi: Strategy

- Generalize the problem by considering n disks.
- Introduce appropriate notation:
  - T_n denotes the *minimum* number of moves required to to transfer n disks from one to another under the rules.
- General patterns are easier to perceive when the extreme or base cases are well understood.
  - Look at small cases first:
    - $T_1 = 1$
    - $T_2 = 3$
    - How about T_3? Does it help us perceive the general case of n?

Tower of Hanoi: A General Solution Pattern

A possible (yet to be proved as *optimal*) solution requires 3 steps:

1. Transfer the *n - 1* smallest disks to a different peg.
2. Move the largest to the remaining free peg.
3. Transfer the *n - 1* disks back onto the largest disk.

How many moves are required from the above 3 steps?

$$(2 \times T_{n-1}) + 1$$

However, we have only proved that the # moves required by this solution are *sufficient*:

$$T_n \leq (2 \times T_{n-1}) + 1$$

But are the above steps all *necessary*? Can you justify?

$$T_n \geq (2 \times T_{n-1}) + 1$$

Tower of Hanoi: Recurrence Relation for T_n

We end up with the following recurrence relation that allows us to compute T_n for any n we like:

$$\begin{aligned} T_0 &= 0 \\ T_n &= (2 \times T_{n-1}) + 1 \quad \text{for } n > 0 \end{aligned}$$

However, the above relation only gives us *indirect* information:

To calculate T_n, first calculate T_{n-1}, which requires the calculation of T_{n-2}, and so on.

Instead, we need a *closed-form solution* to the above recurrence relation, which allows us to *directly* calculate the value of T_n.

Tower of Hanoi: A Hypothesized Closed Form Solution to T_n

$$
\begin{array}{lclcl}
T_0 & & & = & 0 \\
T_1 & = & 2 \times T_0 + 1 & = & 1 \\
T_2 & = & 2 \times T_1 + 1 & = & 3 \\
T_3 & = & 2 \times T_2 + 1 & = & 7 \\
T_4 & = & 2 \times T_3 + 1 & = & 15 \\
T_5 & = & 2 \times T_4 + 1 & = & 31 \\
T_6 & = & 2 \times T_5 + 1 & = & 63 \\
\ldots & & & &
\end{array}
$$

Guess:

$$T_n = 2^n - 1 \qquad \text{for } n \geq 0$$

Prove by *mathematical induction*.

Tower of Hanoi: Prove by Mathematical Induction

Basis:

$$T_0 = 2^0 - 1 = 0$$

Induction:

Assume that

$$T_{n-1} = 2^{n-1} - 1$$

then

$$\begin{array}{cl} & T_n \\ = & \{\texttt{Recurrence relation for } T_n\} \\ & (2 \times T_{n-1}) + 1 \\ = & \{\texttt{Inductive assumption}\} \\ & (2 \times (2^{n-1} - 1)) + 1 \\ = & \{\texttt{Arithmetic}\} \\ & 2^n - 1 \end{array}$$

Revisiting the Tower of Hanoi

Given: A tower of 8 disks, initially stacked in decreasing size on one of 3 pegs.

This shall require

$$T_8 = 2^8 - 1 = 255$$

moves to complete.

Tower of Hanoi in Java (1)

```
void towerOfHanoi(String[] disks) {
  tohHelper (disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int p1, int p2) {
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + p1 + " to " + p2);
  }
  else {
    int intermediate = 6 - p1 - p2;
    tohHelper (disks, from, to - 1, p1, intermediate);
    print("move " + disks[to] + " from " + p1 + " to " + p2);
    tohHelper (disks, from, to - 1, intermediate, p2);
  }
}
```

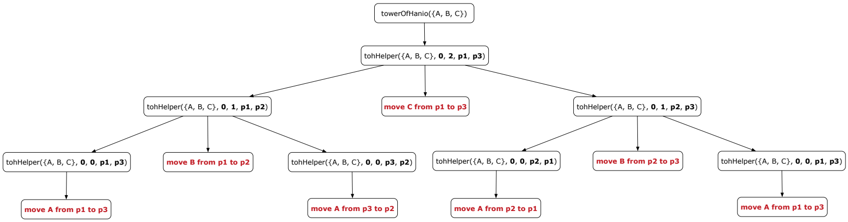
- `tohHelper(disks, from, to, p1, p2)` moves disks $\{disks[from], disks[from+1], \ldots, disks[to]\}$ from peg $p1$ to peg $p2$.
- Peg id's are 1, 2, and 3 $\Rightarrow$ The intermediate one is $6 - p1 - p2$.

Tower of Hanoi in Java (2)

Say ds (disks) is $\{A, B, C\}$, where $A < B < C$.

$$
tohH(ds, \underbrace{0,2}_{\{A,B,C\}}, p1, p3) =
\begin{cases}
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p1, p2) =
\begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \begin{cases} \texttt{Move A: p1 to p3} \end{cases} \\
\texttt{Move B: p1 to p2} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p3, p2) = \begin{cases} \texttt{Move A: p3 to p2} \end{cases}
\end{cases} \\
\texttt{Move C: p1 to p3} \\
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p2, p3) =
\begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p2, p1) = \begin{cases} \texttt{Move A: p2 to p1} \end{cases} \\
\texttt{Move B: p2 to p3} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \begin{cases} \texttt{Move A: p1 to p3} \end{cases}
\end{cases}
\end{cases}
$$

Tower of Hanoi in Java (3)



Recursive Methods: Correctness Proofs

```
1 boolean allPositive(int[] a) { return allPosH (a, 0, a.length - 1); }
2 boolean allPosH (int[] a, int from, int to) {
3   if (from > to) { return true; }
4   else if(from == to) { return a[from] > 0; }
5   else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

- Via mathematical induction, prove that `allPosH` is correct:

  Base Cases
  - In an empty array, there is no non-positive number $\therefore$ result is ***true***. **[L3]**
  - In an array of size 1, the only one elements determines the result. **[L4]**

  Inductive Cases
  - **Inductive Hypothesis**: `allPosH(a, from + 1, to)` returns ***true*** if a[from + 1], a[from + 2], . . . , a[to] are all positive; ***false*** otherwise.
  - `allPosH(a, from, to)` should return ***true*** if: **1)** a[from] is positive; and **2)** a[from + 1], a[from + 2], . . . , a[to] are all positive.
  - By *I.H.*, result is $a[from] > 0 \land$ `allPosH(a, from + 1, to)`. **[L5]**
- `allPositive(a)` is correct by invoking `allPosH(a, 0, a.length - 1)`, examining the entire array. **[L1]**

Beyond this lecture . . .

- Notes on Recursion:
  `http://www.eecs.yorku.ca/~jackie/teaching/lectures/2017/F/EECS2030/slides/EECS2030_F17_Notes_Recursion.pdf`
- API for `String`:
  `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`
- Fantastic resources for sharpening your recursive skills for the exam:
  `http://codingbat.com/java/Recursion-1`
  `http://codingbat.com/java/Recursion-2`
- The *best* approach to learning about recursion is via a functional programming language:
  Haskell Tutorial: `https://www.haskell.org/tutorial/`

Index (1)

Index (2)

Index (3)