

# Recursion



EECS2030 E: Advanced  
Object Oriented Programming  
Summer 2026

CHEN-WEI WANG

## Learning Outcomes



This module is designed to help you learn about:

1. How to solve problems **recursively**
2. Example **recursions** on string and arrays
3. Some more advanced example (if time permitted)

## Beyond this lecture ...



- 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/>

## 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:

```
m(i) {  
    if(i == ...) { /* base case: do something directly */ }  
    else {  
        m(j); /* recursive call with strictly smaller value */  
    }  
}
```

- 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  $j < i$ .

## Tracing Method Calls via 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**.

5 of 37

## Common Errors of Recursive Methods



- Missing Base Case(s).

```
int factorial(int n) {  
    return n * factorial(n - 1);  
}
```

**Base case(s)** are meant as points of stopping growing the runtime stack.

- Recursive Calls on Non-Smaller Problem Instances.

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

Recursive calls on **strictly smaller** problem instances are meant for moving gradually towards the base case(s).

- In both cases, a `StackOverflowException` will be thrown.

7 of 37

## 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**?

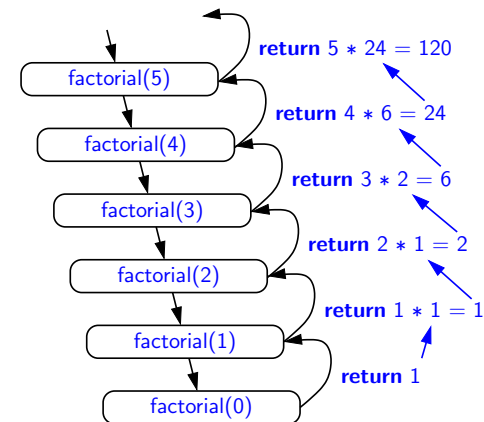
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

- 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;  
}
```

6 of 37

## Recursion: Factorial (2)



8 of 37

## 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(2)*, 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.

9 of 37

## 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 ]

10 of 37

## Recursion: Fibonacci Sequence (1)

- Can you identify the pattern of a Fibonacci sequence?

$F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$

- Here is the formal, *recursive* definition of calculating the  $n_{th}$  number in a Fibonacci sequence (denoted as  $F_n$ ):

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

```
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;
}
```

11 of 37

## Recursion: Fibonacci Sequence (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: {} }
```

12 of 37

## 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();
    }
}
```

13 of 37

## 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);
    }
}
```

15 of 37

## 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 → Return *true* immediately.

**Base Case 2:** String of length 1 → Return *true* immediately.

**Recursive Case:** String of length  $\geq 2$  →

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

14 of 37

## 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 → Return *empty string*.

**Base Case 2:** String of length 1 → Return *that string*.

**Recursive Case:** String of length  $\geq 2$  →

- 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 2) and 1).

16 of 37

## 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;
    }
}
```

17 of 37

## 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);
        }
    }
}
```

19 of 37

## 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).

18 of 37

## 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) {
    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) } }
}
```

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

```
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) } }
}
```

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

20 of 37

## 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 somePositive(int[] a)`

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

21 of 37

## 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 → Return **true** immediately.

The base case is **true** ∴ 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 →

- 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 order**.

23 of 37

## 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);
    }
}
```

22 of 37

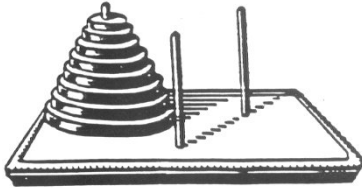
## Recursion: Is an Array Sorted? (2)

```
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);
    }
}
```

24 of 37

## 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.

25 of 37

## Tower of Hanoi: A Recursive Solution

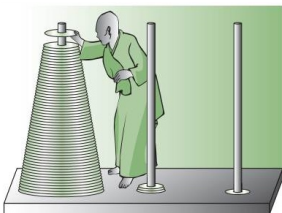


The general, a recursive solution requires 3 steps:

1. Transfer the **n - 1 smallest** disks to a **second** peg.
2. Move the **largest** disk to the **third** peg (free of disks).
3. Transfer the **n - 1 smallest** disks back onto the **largest** disk.

27 of 37

## Tower of Hanoi: Legend



*Brahmins at a temple in Benares, India have been carrying out movement of "Sacred Tower of Brahma", consisting of **sixty-four** golden disks, according to the same rules as in the Tower of Hanoi game, and that the completion of the tower would lead to the end of the world.*

26 of 37

## 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 ori, int des) {  
    if (from > to) { }  
    else if (from == to) {  
        print("move " + disks[to] + " from " + ori + " to " + des);  
    }  
    else {  
        int intermediate = 6 - ori - des;  
        tohHelper(disks, from, to - 1, ori, intermediate);  
        print("move " + disks[to] + " from " + ori + " to " + des);  
        tohHelper(disks, from, to - 1, intermediate, des);  
    }  
}
```

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

28 of 37

## Tower of Hanoi in Java (2)



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

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

29 of 37

## Running Time: Tower of Hanoi (1)



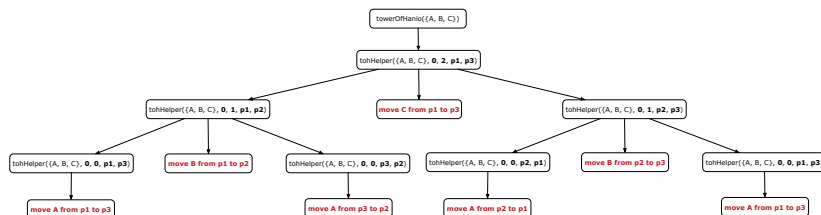
- Generalize the problem by considering  $n$  disks.
- Let  $T(n)$  denote the number of moves required to transfer  $n$  disks from one to another under the rules.
- Recall the general solution pattern:
  - Transfer the  $n - 1$  **smallest** disks to a **second** peg.
  - Move the **largest** disk to the **third** peg (free of disks).
  - Transfer the  $n - 1$  **smallest** disks back onto the **largest** disk.
- We end up with the following recurrence relation that allows us to compute  $T(n)$  for any  $n$  we like:

$$\begin{cases} T(1) = 1 \\ T(n) = 2 \cdot T(n-1) + 1 \quad \text{where } n > 0 \end{cases}$$

- To solve this recurrence relation, we study the pattern of  $T(n)$  and observe how it reaches the **base case(s)**.

31 of 37

## Tower of Hanoi in Java (3)



30 of 37

## Running Time: Tower of Hanoi (2)

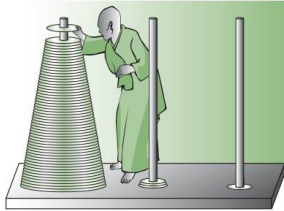


$$\begin{aligned} T(n) &= \underbrace{2}_{1 \text{ term}} \times T(n-1) + \underbrace{1}_{1 \text{ term}} \\ &= \underbrace{2 \times (2 \times T(n-2) + 1)}_{2 \text{ terms}} + \underbrace{1}_{1 \text{ term}} \\ &= \underbrace{2 \times (2 \times (2 \times T(n-3) + 1) + 1)}_{3 \text{ terms}} + \underbrace{1}_{1 \text{ term}} \\ &= \dots \\ &= \underbrace{2 \times (2 \times (2 \times (\dots \times (2 \times T(1) + 1) + 1) + 1) + 1)}_{n-1 \text{ terms}} + \underbrace{1}_{n-1 \text{ terms}} \\ &= 2^{n-1} + (n-1) \end{aligned}$$

$\therefore T(n)$  is  $O(2^n)$

32 of 37

## Tower of Hanoi: Legend



*Brahmins at a temple in Benares, India have been carrying out movement of "Sacred Tower of Brahma", consisting of **sixty-four** golden disks, according to the same rules as in the Tower of Hanoi game, and that the completion of the tower would lead to the end of the world.*

Say one disk can be moved in one second.

**Q.** How long does it take to finish moving 64 disks ( $n = 64$ )?

**A.**  $2^{64}$  seconds  $\approx$  585 billion years ( $>>$  5 billion centuries)!

33 of 37

## Beyond this lecture ...



- Recursions on Arrays: Lab Exercise from EECS2030-F19
- Notes on Recursion:  
[http://www.eecs.yorku.ca/~jackie/teaching/lectures/2026/S/EECS2030/notes/EECS2030\\_S26\\_Notes\\_Recursion.pdf](http://www.eecs.yorku.ca/~jackie/teaching/lectures/2026/S/EECS2030/notes/EECS2030_S26_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/>

34 of 37

## Index (1)



**Learning Outcomes**

**Beyond this lecture ...**

**Recursion: Principle**

**Tracing Method Calls via a Stack**

**Recursion: Factorial (1)**

**Common Errors of Recursive Methods**

**Recursion: Factorial (2)**

**Recursion: Factorial (3)**

**Recursion: Factorial (4)**

**Recursion: Fibonacci Sequence (1)**

**Recursion: Fibonacci Sequence (2)**

35 of 37

## Index (2)



**Java Library: String**

**Recursion: Palindrome (1)**

**Recursion: Palindrome (2)**

**Recursion: Reverse of a String (1)**

**Recursion: Reverse of a String (2)**

**Recursion: Number of Occurrences (1)**

**Recursion: Number of Occurrences (2)**

**Making Recursive Calls on an Array**

**Recursion: All Positive (1)**

**Recursion: All Positive (2)**

**Recursion: Is an Array Sorted? (1)**

36 of 37

## Index (3)

[Recursion: Is an Array Sorted? \(2\)](#)

[Tower of Hanoi: Specification](#)

[Tower of Hanoi: Legend](#)

[Tower of Hanoi: A Recursive Solution](#)

[Tower of Hanoi in Java \(1\)](#)

[Tower of Hanoi in Java \(2\)](#)

[Tower of Hanoi in Java \(3\)](#)

[Running Time: Tower of Hanoi \(1\)](#)

[Running Time: Tower of Hanoi \(2\)](#)

[Tower of Hanoi: Legend](#)

[Beyond this lecture ...](#)