# DSA LECTURE

February 2, 2019

## COURSE INFORMATION

## ESSENTIALS

- Data Structures and Algorithms (DSA)
- Course Website - atu-se.github.io/courses/dsa
- Primary Textbook - Introduction to Java Programming, 10th Edition (Liang)

## MEETING TIMES

- Sunday @ 10:30 a.m.
- Tuesday @ 8:30 a.m.
- Wednesday @ 10:30 a.m.

## HOW TO SUCCEED

- Attend all lectures
- Take notes (not all materials will be distributed as slides)
- Ready and study your textbook
- Do all assignments
- Ask questions

## ACADEMIC INTEGRITY

- Read the academic integrity policy in the syllabus
- You are encouraged to discuss the topics among yourselves.
- Unless otherwise noted, your work should be your own and you should not share your work with others
- Copying or cheating on homework, exams, etc. may

## LECTURE 1:
## RECURSION

## KEY TERMS

- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.
- A recursive method is one that invokes itself.

## RECURSION EXAMPLE: FACTORIAL

## EXAMPLE: FACTORIAL

$$0! = 1$$

$$n! = n \cdot (n - 1)! \qquad \text{when} \geq 1$$

## EXAMPLE: FACTORIAL

$$2! = 2 \times 1! = 2 \times 1 = 2$$

$$3! = 3 \times 2! = 3 \times 2 \times 1! = 3 \times 2 \times 1 = 6$$

## EXAMPLE: FACTORIAL

```
import java.util.Scanner;

public class ComputeFactorial {
    /** Main method */
    public static void main(String[] args) {
        // Create a Scanner

        Scanner input = new Scanner(System.in);
        System.out.print("Enter a non-negative integer:");
        int n = input.nextInt();
        // Display factorial
        System.out.println("Factorial of " + n + " is "
        + factorial(n));
    }
```

…

## EXAMPLE: FACTORIAL

...

```
  /** Return the factorial for a specified number */
  public static long factorial(int n) {
      if (n == 0) // Base case
          return 1;
      else
          return n * factorial(n - 1); // Recursive call
  }
}
```

## WHAT-IF

What if our factorial function was like this?

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

## EXAMPLE: FIBONACCI

# EXAMPLE: FIBONACCI

```java
import java.util.Scanner;
public class ComputeFibonacci {
  /** Main method */
  public static void main(String args[]) {

    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for the Fibonacci number:
    int index = input.nextInt();
    // Find and display the Fibonacci number
    System.out.println(
      "Fibonacci number at index " + index + " is " + fib(inde
  }
```

## EXAMPLE: FIBONACCI 2

```
/** The method for finding the Fibonacci number */
public static long fib(long index) {
  System.out.println("Called fib");
  if (index == 0) // Base case

    return 0;
  else if (index == 1) // Base case
    return 1;
  else  // Reduction and recursive calls
    return fib(index - 1) + fib(index - 2);
  }
}
```

## EXAMPLE:

## RECURSIVE PALINDROME

## EXAMPLE: RECURSIVE PALINDROME

```java
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  public static boolean isPalindrome(String s, int low, int hi
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }
```

# EXAMPLE: RECURSIVE PALINDROME 2

```
  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("
  }
}
```

## EXAMPLE:

## RECURSIVE SELECTION SORT

## EXAMPLE: SELECTION SORT

- Find the smallest element in the list and swap it with the first element.
- Ignore the first element and sort the remaining smaller list recursively.

## EXAMPLE: SELECTION SORT

```
public class RecursiveSelectionSort {
  public static void sort(double[] list) {
    sort(list, 0, list.length - 1); // Sort the entire list
   }
```

## EXAMPLE: SELECTION SORT

```
public static void sort(double[] list, int low, int high) {
  if (low < high) {
    // Find the smallest number and its index in list(low .. h
    int indexOfMin = low;
    double min = list[low];
    for (int i = low + 1; i <= high; i++) {
      if (list[i] < min) {
        min = list[i];
        indexOfMin = i;
      }
    }
```

## EXAMPLE: SELECTION SORT

```
    // Swap the smallest in list(low .. high) with list(low)
    list[indexOfMin] = list[low];

    list[low] = min;

    // Sort the remaining list(low+1 .. high)
    sort(list, low + 1, high);
  }
}
```

# EXAMPLE: SELECTION SORT

```java
  public static void main(String[] args) {
    double[] list = {2, 1, 3, 1, 2, 5, 2, -1, 0};

    sort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```

## PRACTICE EXERCISES

## PRACTICE EXERCISE 18.4

Write a recursive mathematical definition for computing $2^n$ for a positive integer n.

# PRACTICE EXERCISE 18.4

*power_of_two*(0) = 1

*power_of_two*(*n*) = 2 * *power_of_two*(*n* – 1)        when > 0

# PRACTICE EXERCISE 18.7A

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);

  }
  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n + " ");
      xMethod(n - 1); }
    }
}
```

# PRACTICE EXERCISE 18.7B

```java
public class Test {
  public static void main(String[] args) {
    xMethod(5);

  }
  public static void xMethod(int n) {
    if (n > 0) {
      xMethod(n - 1);
      System.out.print(n + " "); }
    }
}
```

# PRACTICE EXERCISE 18.8A

```
public class Test {
  public static void main(String[] args) {
      xMethod(1234567);

  }
  public static void xMethod(double n) {
    if (n != 0) {
      System.out.println(n);
      xMethod(n / 10); }
    }
}
```

## PRACTICE EXERCISE 18.8B

```java
public class Test {
  public static void main(String[] args) {
      Test test = new Test();

      System.out.println(test.toString());
    }
    public Test() {
      Test test = new Test();
    }
}
```

## LECTURE 1B

## PROBLEM SOLVING USING RECURSION

All recursive methods have the following characteristics:

- The method is implemented using an if-else or a switch statement that leads to different cases.
- One or more base cases (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

## PROBLEM SOLVING USING RECURSION

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem is the same as the original problem but smaller in size. You can apply the same approach to each subproblem to solve it recursively.

# RECURSIVE COFFEE DRINKING

```java
public static void drinkCoffee(Cup cup) {
  if (!cup.isEmpty()) {
        cup.takeOneSip(); // Take one sip
        drinkCoffee(cup);
      }
}
```

# PRINTING N TIMES

```java
public static void nPrintln(String message, int times)
{
  if (times >= 1) {
      System.out.println(message);
      nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

# RECURSIVE BINARY SEARCH (BS)

## BS ALGORITHM

Given an array that is sorted in increasing order, imagine searching for a single element (the key).

Begin by comparing the key to the middle element of the array. There are three possibilities.

## BS ALGORITHM

- Case 1: If the key is less than the middle element, recursively search for the key in the first half of the array.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, recursively search for the key in the second half of

## BS IMPLEMENTATION

```
public class RecursiveBinarySearch {
  public static int recursiveBinarySearch(int[] list, int key)
    int low = 0;
    int high = list.length - 1;
    return recursiveBinarySearch(list, key, low, high);
  }
```

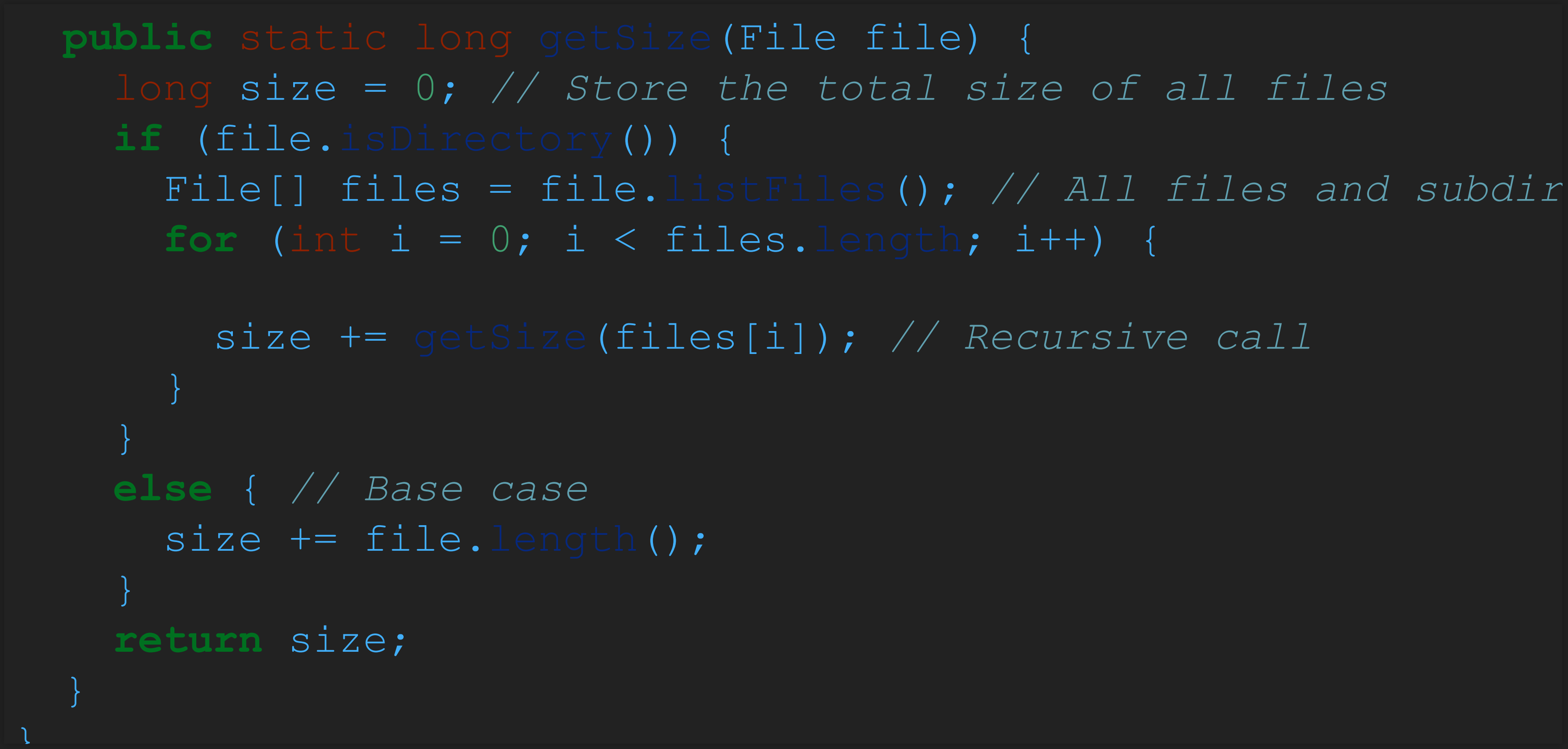
# BS IMPLEMENTATION

```java
  public static int recursiveBinarySearch(int[] list, int key,
      int low, int high) {
    if (low > high)  // The list has been exhausted without a
      return -low - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      return recursiveBinarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
      return mid;
    else
      return recursiveBinarySearch(list, key, mid + 1, high);
  }
}
```

# DIRECTORY SIZE

```java
import java.io.File;
import java.util.Scanner;
public class DirectorySize {
  public static void main(String[] args) {

    // Prompt the user to enter a directory or a file
    System.out.print("Enter a directory or a file: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes"
  }
```

# DIRECTORY SIZE



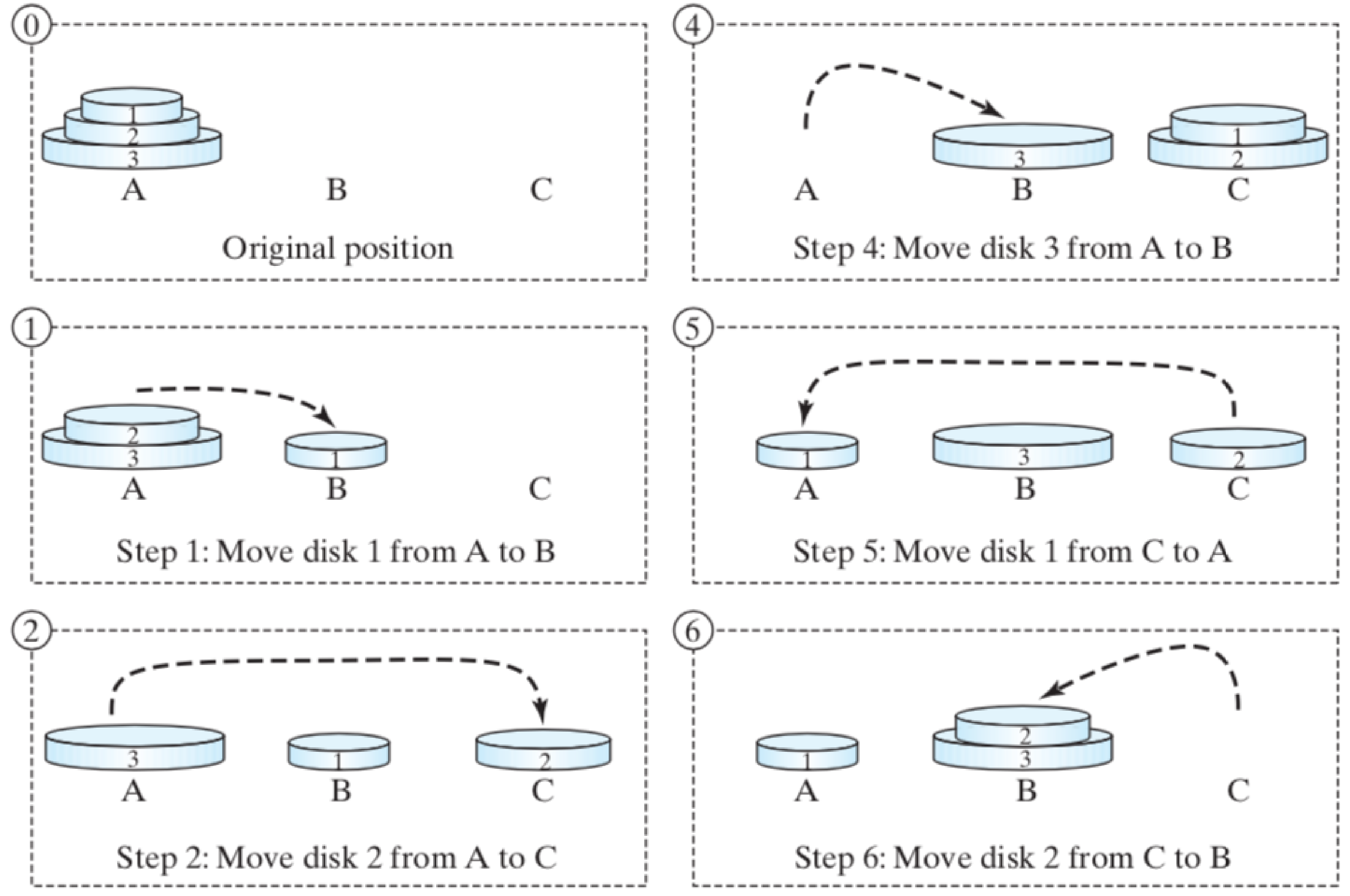
```java
  public static long getSize(File file) {
    long size = 0; // Store the total size of all files
    if (file.isDirectory()) {
      File[] files = file.listFiles(); // All files and subdir
      for (int i = 0; i < files.length; i++) {

        size += getSize(files[i]); // Recursive call
      }
    }
    else { // Base case
      size += file.length();
    }
    return size;
  }
}
```

## TOWER OF HANOI

## TOWER OF HANOI

The Tower of Hanoi problem is a classic problem that can be solved easily using recursion, but it is difficult to solve otherwise.

# TOWER OF HANOI


0
1
2
3
A
B
C
Original position
1
2
3
A
B
C
Step 1: Move disk 1 from A to B
3
1
2
A
B
C
Step 2: Move disk 2 from A to C
4
3
1
2
A
B
C
Step 4: Move disk 3 from A to B
5
1
3
2
A
B
C
Step 5: Move disk 1 from C to A
6
1
2
3
A
B
C
Step 6: Move disk 2 from C to B

## TOWER OF HANOI

There are n disks labeled 1, 2, 3, . . . , n and three towers labeled A, B, and C.

- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the smallest disk on a tower.

Demonstration

# TOWER OF HANOI

```java
import java.util.Scanner;
public class TowersOfHanoi {
  /** Main method */
  public static void main(String[] args) {

    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter number of disks: ");
    int n = input.nextInt();
    // Find the solution recursively
    System.out.println("The moves are:");
    moveDisks(n, 'A', 'B', 'C');
  }
```

# TOWER OF HANOI

```java
/** The method for finding the solution to move n disks
    from fromTower to toTower with auxTower */
  public static void moveDisks(int n, char fromTower,
      char toTower, char auxTower) {
    if (n == 1) // Stopping condition
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
    else {
      moveDisks(n - 1, fromTower, auxTower, toTower);
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
      moveDisks(n - 1, auxTower, toTower, fromTower);
    }
  }
}
```

## RECURSION VS. ITERATION

## RECURSION VS. ITERATION

- Recursion is an alternative form of program control. It is essentially repetition without a loop.
- Any problem that can be solved recursively can be solved nonrecursively with iterations.
- Recursion has some negative aspects: it uses up too much time and too much memory.

## WHY RECURSION?

- In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain.
- The rule of thumb is to use which- ever approach can best develop an intuitive solution that naturally mirrors the problem

## TAIL RECURSION

## TAIL RECURSION

- A tail recursive method is efficient for reducing stack size.
- A recursive method is said to be tail recursive if there are no pending operations to be performed on return from a recursive call

# TAIL RECURSION

## NON-TAIL RECURSION

```
...
...

recursive_method_call()
...
```

## TAIL RECURSION

```
...
...
...
recursive_method_call()
```

## NONTAIL-RECURSIVE FACTORIAL

```java
/** Return the factorial for a specified number */
public static long factorial(int n) {

  if (n == 0) // Base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call
}
```

## TAIL RECURSIVE FACTORIAL

```java
/** Auxiliary tail-recursive method for factorial */
private static long factorial(int n, int result) {

  if (n == 0)
    return result;
  else
    return factorial(n - 1, n * result); // Recursive call
}
```

## CHAPTER KEY POINTS

## CHAPTER KEY POINTS

1. A recursive method is one that directly or indirectly invokes itself. For a recursive method to terminate, there must be one or more base cases.
2. Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise

## CHAPTER KEY POINTS

3. Sometimes the original method needs to be modified to receive additional parameters in order to be invoked recursively. A recursive helper method can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method’s local variables and

## CHAPTER KEY POINTS

5. A recursive method is said to be tail recursive if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack size.