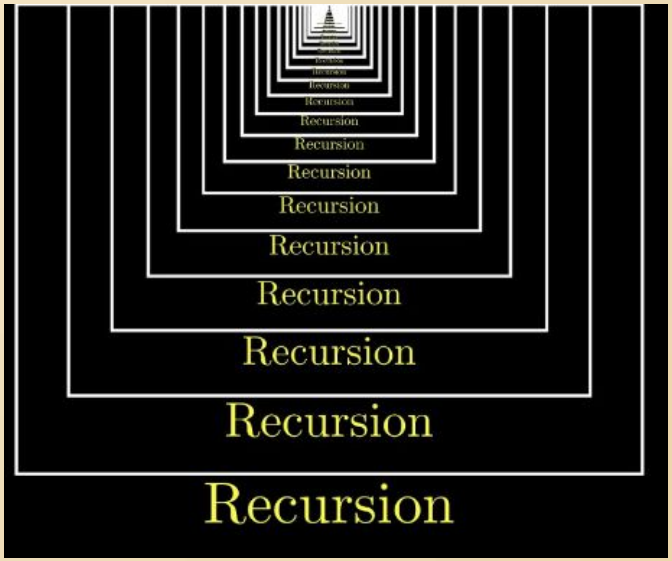
Recitation 9

Overview

- Recursion
  - Array
- Memoization
- StringBuilder
- OOP
  - Example 1: Polynomial
  - Hands-On session: Lamp

Recitation 9

Recursion



Question 1 - Recursion - Arrays

- Now Let's see usage of recursion with arrays.
- Usually when we want to go over the array we must use a helper function
- Let's find the maximum value within an array using recursion. You may assume that it is notempty array

- maxValue({-1,2,3,-5,2,2,1,3}); // 3
- maxValue({1,2,3,5,2}); // 5

Question 1 - Solution

```
public static int maxValue(int [] arr) {
     int maxIndex = maxValueHelper(arr, 0, 0);
     return arr[maxIndex];
}

private static int maxValueHelper(int [] arr, int curMaxIndex, int curIndex) {
     if (curIndex == arr.length) {
     return curMaxIndex;
     }
     if (arr[curMaxIndex] < arr[curIndex]){
     return maxValueHelper(arr, curIndex, curIndex + 1);
     }
     return maxValueHelper(arr, curMaxIndex, curIndex + 1);
}
```

Question 2 – Merge arrays

- mergeArrays (or merge) is a function which takes two given **sorted** int arrays and returns one array which contains all elements from the two arrays, sorted.
  - mergeArrays({1, 3, 5}, {2, 4, 6}); // {1, 2, 3, 4, 5, 6}
  - mergeArrays({1}, {1, 2, 3}); // {1, 1, 2, 3}
- Implement a recursive version of this function

Question 2 – Solution

```
public static int[] mergeArrays(int[] arr1, int[] arr2) {
        // create a new array to store the merged elements
        int[] merged = new int[arr1.length + arr2.length];

        // merge the arrays
        mergeArrays(arr1, arr2, 0, 0, merged);

        // return the merged array
        return merged;
 }
```

Question 2 – Solution (continue)

```
private static int[] mergeArrays(int[] arr1, int[] arr2, int index1, int index2, int[] merged) {
        // if one of the arrays is empty, copy the rest of the other array into merged
        if (index1 >= arr1.length) {
            copyArray(arr2, merged, index2, index1 + index2);
            return merged;
        }
        if (index2 >= arr2.length) {
            copyArray(arr1, merged, index1, index1 + index2);
            return merged;
        }
        // add the smaller element to the merged array
        if (arr1[index1] < arr2[index2]) {
            merged[index1 + index2] = arr1[index1];
            index1++;
        } else {
            merged[index1 + index2] = arr2[index2];
            index2++;
        }
        // recursively merge the rest of the arrays
        return mergeArrays(arr1, arr2, index1, index2, merged);
    }
```

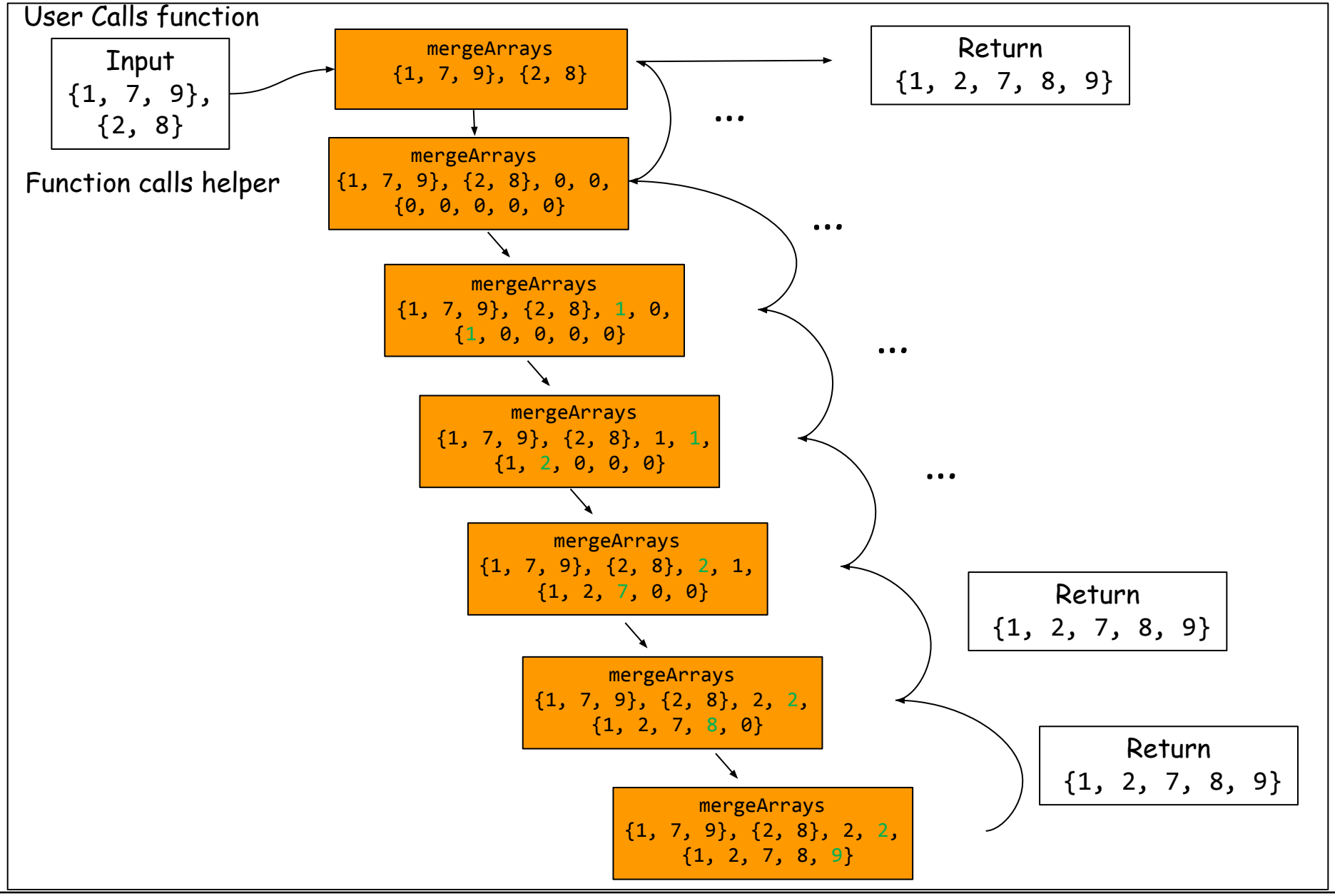
Question 2 – Solution (continue)

```
private static void copyArray(int[] arr1, int[] arr2, int index1, int index2) {
    // if finished copying, return
    if (index1 == arr1.length || index2 == arr2.length) {
        return;
    }

    // copy the value at the current index
    arr2[index2] = arr1[index1];

    // recursively copy the rest of the array
    copyArray(arr1, arr2, index1 + 1, index2 + 1);
}
```

Question 2 – Visual Example



Question 3 – filterByLength

- The function filterByLength receives a String array, and a non negative int 'minLen' which represent a certain length of a String, and returns a new String array which holds the all the strings which has length equal or bigger than 'minLen'.

```
String [] arr = {"hello","hi","word"};

String [] arrFiltered = filterByLength(arr,4);

printArray(arrFiltered); // {"hello","word"}

printArray(arr);     // {"hello","hi","word"}
```

- Implement a recursive version of this function

Question 3 – Solution

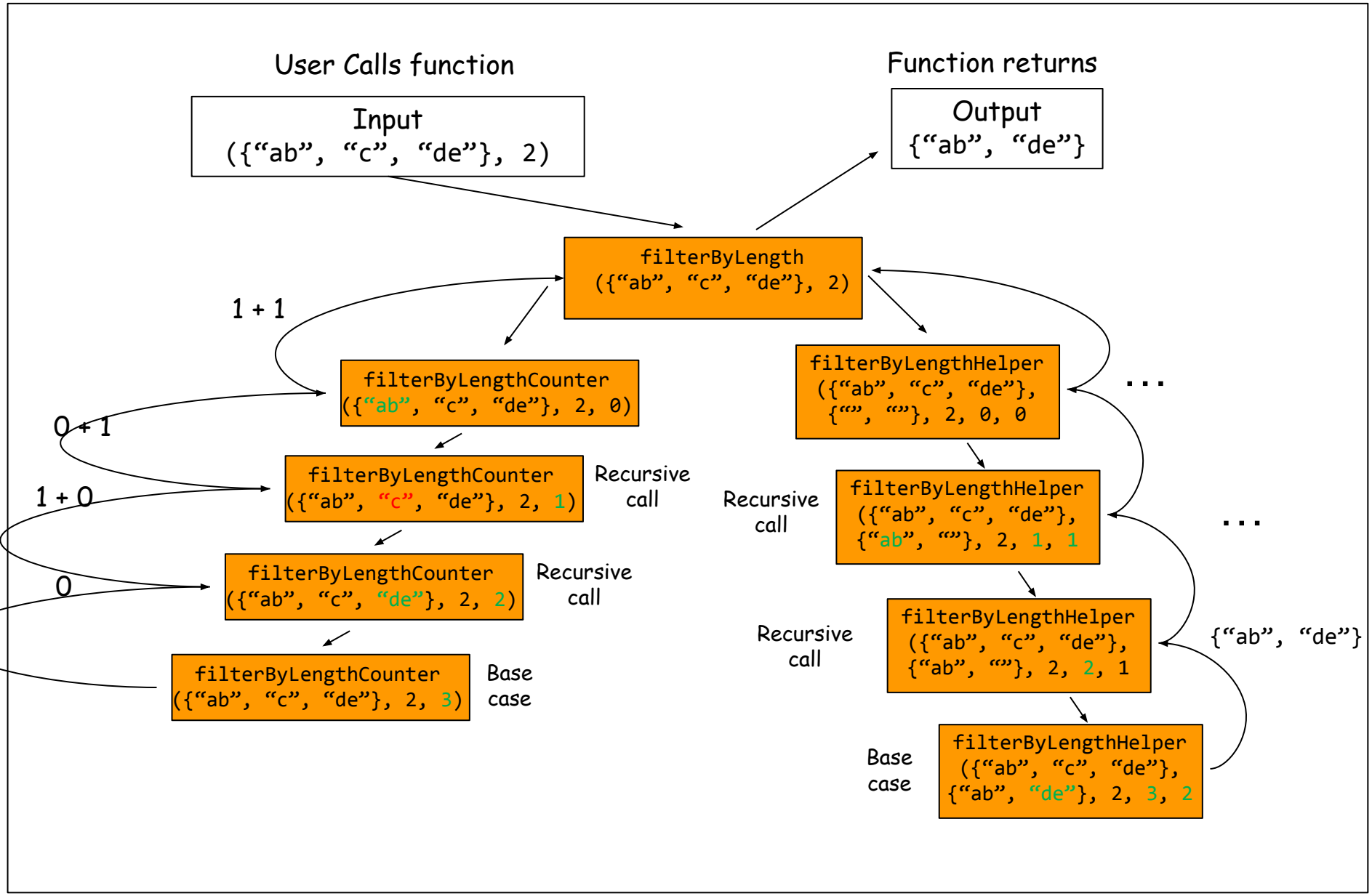
```
public static String[] filterByLength(String[] arr, int minLen) {
        int count = filterByLengthCounter(arr, minLen, 0);
        String[] filteredArray = new String[count];
        filteredArray = filterByLengthHelper(arr, filteredArray, minLen, 0, 0);
        return filteredArray;
}
private static int filterByLengthCounter(String[] arr, int minLen, int index) {
        if (arr.length == index) {
            return 0;
        }
        int isInLength = (arr[index].length() >= minLen) ? 1 : 0;
        return isInLength + filterByLengthCounter(arr, minLen, index + 1);
}
```

Question 3 – Solution (continued)

```
private static String[] filterByLengthHelper(String[] arr, String[] filtered,
                                             int minLen, int originalArrIndex, int newArrIndex) {
        if (arr.length == originalArrIndex) {
          return filtered;
        }

        if (arr[originalArrIndex].length() >= minLen){
          filtered[newArrIndex] = arr[originalArrIndex];
          newArrIndex++;
        }
        return filterByLengthHelper(arr, filtered, minLen, originalArrIndex + 1, newArrIndex);
    }
```

Question 3 – Visual Example



Recitation 9

Memoization

Reminder – Recitation 7, Question 2

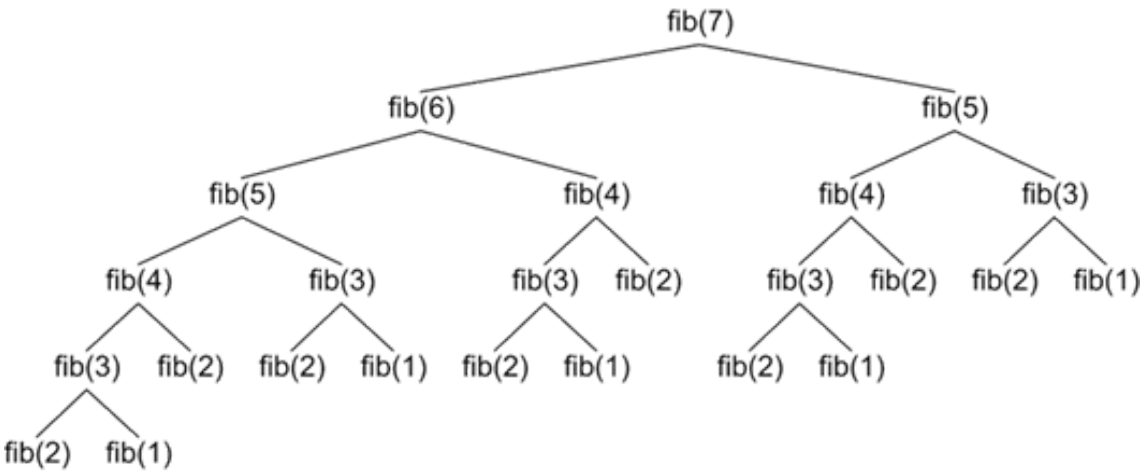
- Let's reconsider the following code:

```
public static int fibo2(int n){
    if (n == 0 || n == 1){
        return n;
    }
    return fibo2(n - 1) + fibo2(n - 2);
}
```

- Natural questions to ask are:
- How many recursive calls are made to calculate one value of *n*?
- Aren't we calculating the same values repeatedly?

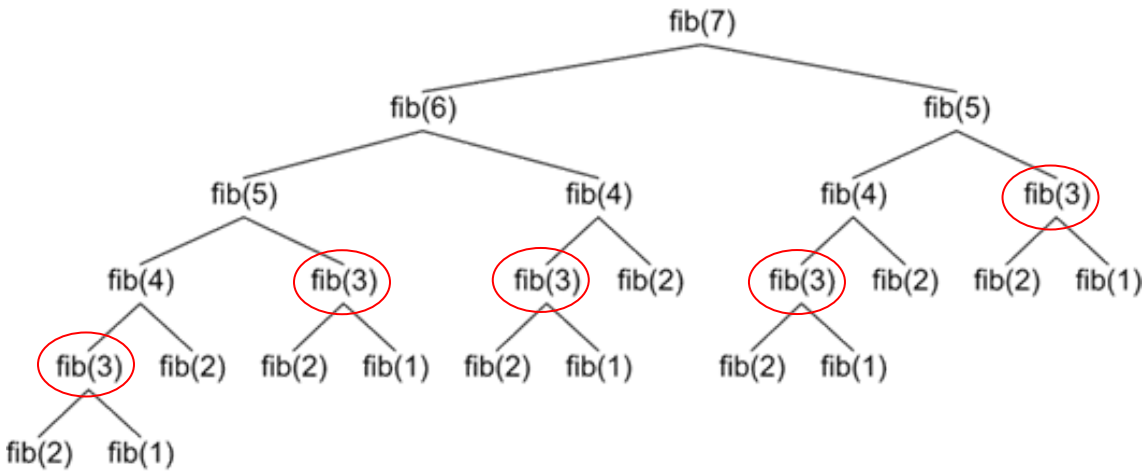
Fibonacci

- Example for *n* = 7:



- Mathematical analysis shows that for any *n*, the total number of recursive calls is somewhere between $\sqrt{2}^n$ and 2^n.
- For *n* = 100, we will need more than 10,000,000,000,000,000 recursive calls.
- Let's see how we can increase the efficiency.

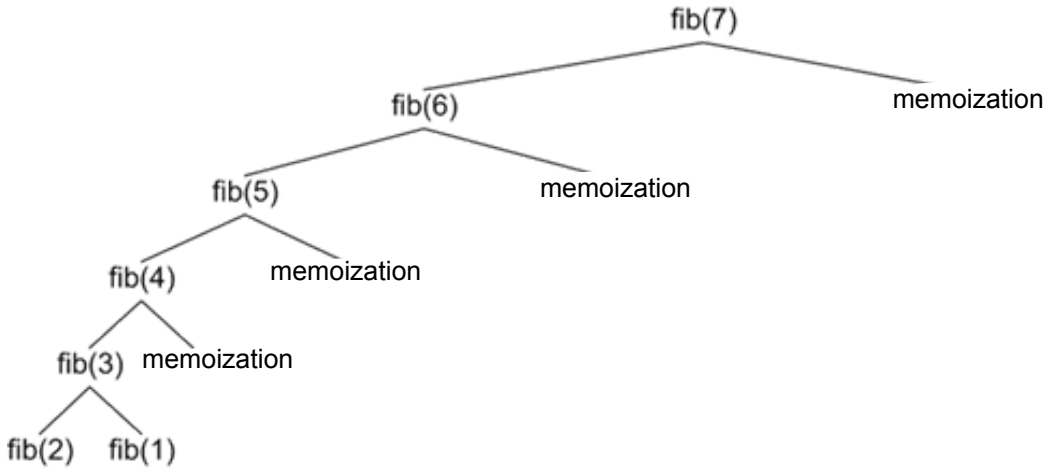
Fibonacci – The Memoization Method



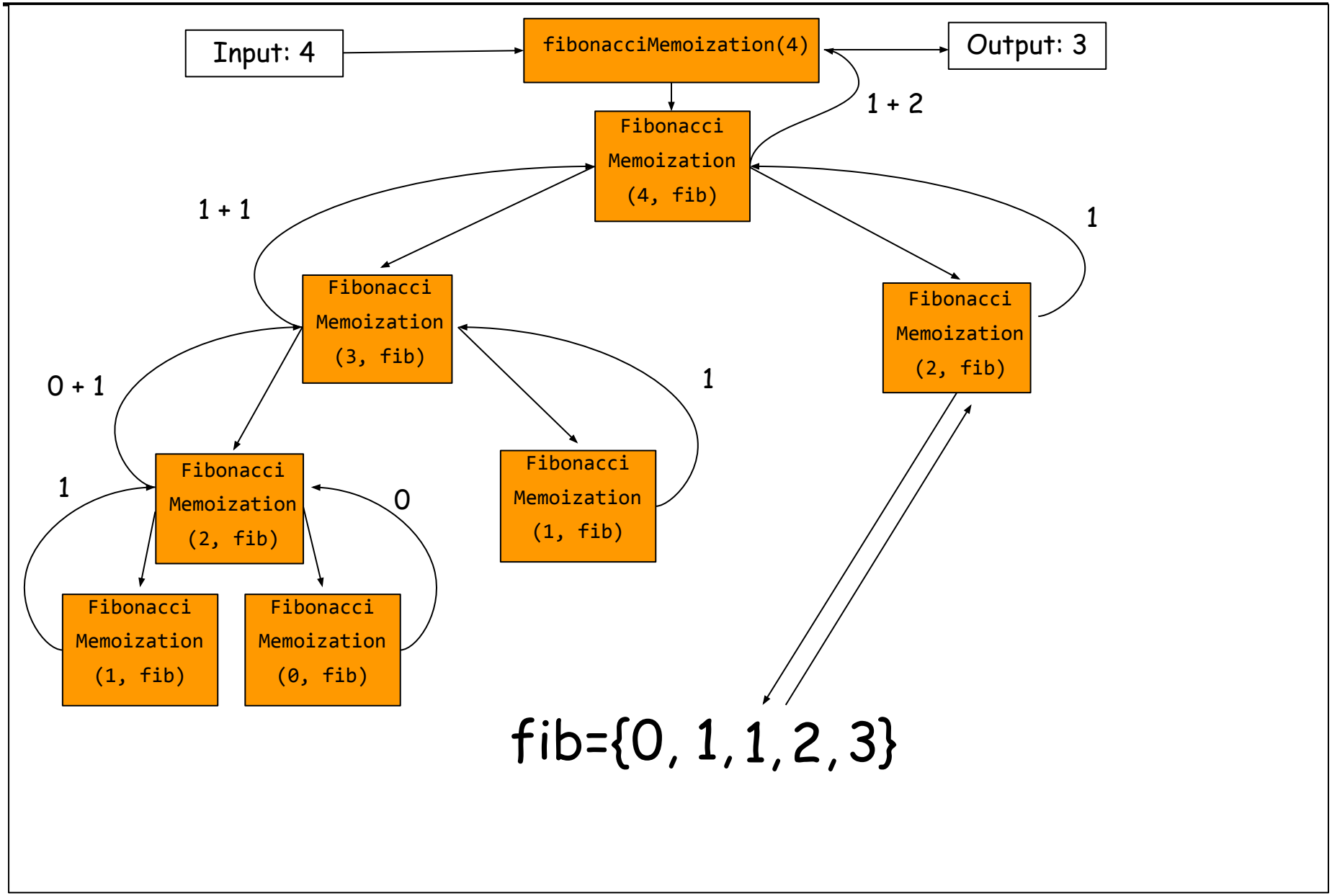
- Notice that even though we have many recursive calls, a great fraction of them is identical.
- Calculating fib(3) so many times is redundant.
- If we could reuse past calculations we would lower the number of recursive calls.

Fibonacci – The Memoization Method

- A memoization table does exactly that.
- In our case, we will use an array to hold all the Fibonacci numbers which were calculated.
- Every time a new value is calculated we will keep the value inside the memoization table.
- Every time a recursive call is made we will check if that value was already calculated.
- If it was, we can simply return the value present in the memoization table.
- Otherwise, calculate as normal.



Question 4 - Visual Example



Question 4 - Memoization - Solution

```
public static long fibonacciMemoization(int n){
        return fibonacciMemoization(n, new long[n + 1]);
}
public static long fibonacciMemoization(int n, long[] fib) {
        // if n is 0 or 1, return n
        if (n == 0 || n == 1) {
            return n;
        }
        // if the Fibonacci number has already been calculated, return it
        if (fib[n] != 0) {
            return fib[n];
        }

        // the Fibonacci number at index n is the sum of the previous two numbers
        fib[n] = fibonacciMemoization(n - 1, fib) + fibonacciMemoization(n - 2, fib);
        return fib[n];
    }
```

Question 5 - Memoization – maxSubArray

- The function maxSubArray receives an int array, and returns an int.
- The array holds both negative and positive, unsorted.
- The function calculates the largest (in terms of numric value) sum of continuous sub array and return its value.
- Examples:
  - maxSubArray({1,2,3});
    - // value : 6, subarray:{1,2,3}
  - maxSubArray({-2, 1, -3, 4, -1, 2, 1, -5, 4});
    - // value : 6, subarray:{4,-1,2,1}
  - maxSubArray({5, -1, -2, 7, -6, 3, -2, -10});
    - // value : 9, subarray:{5,-1,-2,7}
  - maxSubArray({3, 1, -7, -6, 3, -2, 2, 1, -3, 14, 5});
    - // value : 20, subarray: {2, 1, -3, 14, 5}
- Build the function maxSubArray using recursion and memoization

Question 5 - Solution

```
public static int maxSubArray(int[] nums) {
        // create new array for memo'
        int [] maxSub = new int [nums.length];
        // create fill the data with Integer.MIN_VALUE to not tamper with calculation
        fillArray(maxSub, 0, Integer.MIN_VALUE);
        // base case
        maxSub[0] = nums[0];
        // fill rest of memo array
        maxSubArrayHelper(nums, maxSub, 1);
        // find max in the memo array, Question 1
        return findMax(maxSub);
    }

 private static void maxSubArrayHelper(int[] nums, int [] maxSub, int index) {
        // if the finished running, return
        if (nums.length == index) {
            return;
        }
        // calculating according to formula
        maxSub[index] = Math.max(maxSub[index - 1] + nums[index], nums[index]);

        // recursively calculating the rest of the array
        maxSubArrayHelper(nums, maxSub, index + 1);
    }
```

Question 5 – Solution (continue)

```
private static void fillArray(int[] arr, int index, int value) {
        // if the finished running, return
        if (index == arr.length) {
            return;
        }
        // set the value at the current index
        arr[index] = value;

        // recursively fill the rest of the array
        fillArray(arr, index + 1, value);
}
```

Recitation 9

Exceptions

Runtime Errors

An event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions:

- Trying to read a file that doesn't exist
- Trying to divide an integer by zero
- Calling a method with improper arguments. Say, calling the CharAt(int i) method of a String object with a negative integer.
- Etc.

In these cases, the instruction would fail. We say that a run-time error had occurred.

Exceptions

- In Java, run-time errors are indicated by exceptions.
- If a method wants to signal that something went wrong during its execution it throws an exception.
- Throwing an exception involves:
  - Creating an **exception object** that encloses information about the problem that occurred. (with optional message and more info)
  - Use reserved word **throw** to notify Java that an exception has occurred to trigger a stop to the program.

```
public char charAt(int i){
     if (i < 0 || i >= this.length()){
        throw new IndexOutOfBoundsException(i + " is out of bounds");
     }
}
```

Examples of Exception Types

- Several types of exceptions are defined in Java:
  - IllegalArgumentException - passing an illegal argument to a method. Example: Math.pow(0,"0");
  - ArithmeticException – Example: division by zero.
  - NullPointerException - trying to refer to an object through a variable whose value is null
  - And more.
- You can define more on your own.

Documenting Exceptions

- JavaDoc helps us document possible exceptions via the @throws tag.

```
/** Constructs a fraction.
 * Convention 1: the newly constructed fraction is always reduced.
 *               For example, given 6 and 9, constructs the fraction 2/3.
 * Convention 2: if the denominator is negative,
 *               switches the signs of both the numerator and the denominator.
 *               For example, 2/-3 becomes -2/3, and -2/-3 becomes 2/3.
 *
 * @param numerator    can be signed
 * @param denominator  can be signed
 * @throws ArithemticException if the denominator is 0.
 */

public Fraction (int numerator, int denominator) {
```

Handling Exceptions

- As you already (painfully) know, runtime errors crash your programs.
- However, this is not always preferable.
- For example, imagine we are trying to read an integer through the standard input.
- When a user enters "five" or "5," instead of "5", our program will crash.
- Think about more complicated programs (Flight Control, Security Systems, etc.).
- It doesn't seem reasonable for a program to crash when a user makes a typo.

Try … Catch

- The idea is to try and execute some statement.
  - If whatever you tried succeeded, all is well. The program will continue, and the catch statement is ignored.
  - If an exception is thrown by any of the statements within the try block, the rest of the statements are skipped, and the corresponding catch block is executed.
- Every catch clause has an associated exception type.
- When an exception occurs, processing continues at the first catch clause matching the exception type.

```
try {
        \\something
} catch (ExceptionType e){
        \\handle it
}
```

Try … Catch - Examples

```
public class ExceptionExamples {

  /**
   * Tries to read an integer through the
standard input
   * If an illegal argument is entered, returns
0
   */
  public static int readInt() {
    In in = new In ();
    int ans = 0;
    StdOut.println("Please enter a number.");
    try {
      String s = in.readString();
      ans = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      ans = 0;
    }
    return ans;
  }
```

```
public class ExceptionExamples {

  /**
   * Tries to read an integer through the standard input
   * Keeps prompting the user to enter a valid argument
   */
  public static int readInt2() {
    int ans = 0;
    boolean legal = false;
    StdOut.println("Please enter a number.");
    In in = new In ();
    while (!legal) {
      try {
        String s = in.readString();
        ans = Integer.parseInt(s);
        legal = true;
      } catch (NumberFormatException e) {
        StdOut.println("Invalid argument, try again.");
      }
    }
    return ans;
  }
```

Try … Catch … Finally

- After the try-catch structure, there's the finally block. The finally block contains code that will be executed whether an exception is thrown or not.
  - It's useful for cleanup operations, such as closing a file
  - It's optional

```
try {
        \\do something
} catch (ExceptionType e){
        \\handle it
} finally {
        \\do something regardless of whether an
exception occurred or not
}
```

Try … Catch … Finally - Example

```
public class ExceptionExamples {


  /**
   * Tries to read an integer through the standard input
   * If an illegal argument is entered, returns 0
   */
  public static int readInt3() {
    int ans = 0;
    StdOut.println("Please enter a number.");
    In in = new In();
    try {
      String s = in.readString();
      ans = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      ans = 0;
    } finally {
        // will execute always.
        return ans;
     }
}
```

Recitation 9

StringBuilder

StringBuilder

- A class which builds strings.
- https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html
- useful when we want to write a toString() method to a class.
- Holds a 'mutable' string - efficient
- Has useful methods for string building:
- append(), insert(), replace()

Recitation 9

Object Oriented Programming

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

(Edsger Dijkstra)

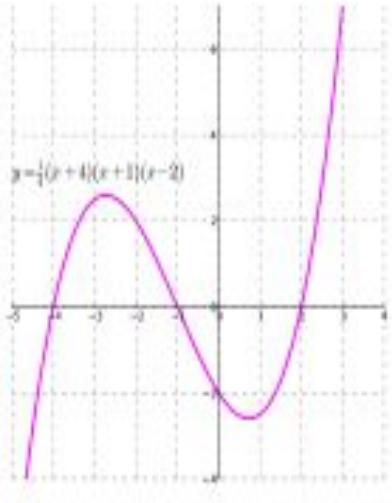
izquotes.com

Polynomials

- A real polynomial on the variable 'x' is a formal sum of the form.

$$\sum_{i=0}^{n} a_i x^i$$

- Where all a_i are real numbers, and are known as the coefficients of the polynomial.
- n is denoted as the polynomial's degree.
- Polynomials form one of the most important classes of function in any applicable science.
- We will begin today's discussion by creating a class to abstract and represent a polynomial.



Polynomials - Discussion

- Before beginning, lets ask ourselves a few questions:
  - What defines a polynomial?
  - What fields must a polynomial class contain?
  - What kind of operations can we do with polynomials?
  - What methods will a polynomial class have?
  - How will we implement them?

The Polynomial Class

```
public class Polynomial {

    // The fields of a Polynomial instance
    private double[] coefficients;

    /** Default constructor*/
    public Polynomial();


    /** Standard constructor*/
    public Polynomial(double[] coefficients);


    /** copy constructor*/
    public Polynomial(Polynomial original);
```

The Polynomial Class - API

```
    /** Returns a copy of the coefficients. */
    public double[] getCoefficients() {}

    /** Returns the degree of the polynomial. */
    public int degree () {}

    /** Evaluates the polynomial at point x. */
    public double evaluate(double x) {}

    /** adds two polynomial. */
    public Polynomial addition(Polynomial other) {}

    /** Multiplies two polynomials. */
    public Polynomial multiplication(Polynomial other) {}

    /** subtracts two polynomials. */
    public Polynomial subtraction(Polynomial other) {}

    /** Derives a polynomial*/
    public Polynomial derivative() {}

    /** Checks if two polynomials are equal*/
    public boolean equals(Polynomial other) {}

    /** Returns a textual representation of this polynomial. */
    public String toString() {}
}
```

Polynomials

```
public class Polynomial {

    // The polynomial's coefficients. Has a non-zero leading term.
    private double[] coefficients;

    /**
     * Default constructor, creates the 0 polynomial.
     */
    public Polynomial() {
        this.coefficients = new double[1];
        this.coefficients[0] = 0;
    }

...

}
```

Polynomials

```
public class Polynomial {
    /**
     * A standard constructor. Ensures that the highest term of the polynomial is non-zero.
     * @param coefficients- a list of the polynomial's coefficients
     * @throws RuntimeException in the case no coefficients are entered.
     */
    public Polynomial(double[] coefficients) {
        if (coefficients == null || coefficients.length == 0) {
            throw new InvalidArgumentException("not a valid polynomial");
        }
        int leadingIndex = coefficients.length - 1;
        while (leadingIndex >= 0 && coefficients[leading] == 0) {
            leadingIndex--;
        }
        if (leadingIndex < 0) {
            this.coefficients = new double[1];
            this.coefficients[0] = 0;
        } else {
            this.coefficients = new double[leading + 1];
            for (int i = 0; i < leadingIndex + 1; i++) {
                this.coefficients[i] = coefficients[i];
            }
        }
    }
```

Polynomials

```
public class Polynomial {

    /**
     * a copy constructor. Deep copies the other polynomial's coefficients.
     * @param other - the polynomial to be copied
     */
    public Polynomial(Polynomial original) {
        this.coefficients = new double[original.degree() + 1];
        for (int i = 0; i < this.coefficients.length; i++) {
            this.coefficients[i] = original.getCoefficients()[i];
        }
    }

    /**
     * Returns a copy of the polynomial's coefficients.
     * @return a copy of the polynomial's coefficients.
     */
    public double[] getCoefficients() {
        double[] ans = new double[this.coefficients.length];
        for (int i = 0; i < ans.length; i++) {
            ans[i] = this.coefficients[i];
        }
        return ans;
    }
```

Polynomials

```
public class Polynomial {

     /**
     * Returns the degree of the polynomial. The 0 polynomial is considered to have degree 0.
     * @return the degree of the polynomial.
     */
    public int degree() {
        return this.coefficients.length - 1;
    }

    /**
     * evaluates the polynomial at a given point.
     * @param x - the point to be evaluated.
     * @return this evaluated at x.
     */
    public double evaluate(double x) {
        double ans = 0;
        for (int i = 0; i <= this.degree(); i++) {
            // sum all parts of the polynomial according to the variable's value
            ans = ans + (this.coefficients[i] * (Math.pow(x, i)));
        }
        return ans;
    }
```

Polynomials

```
public class Polynomial {

     /**
     * Adds two polynomials.
     * @param other - the polynomial to be added to this
     * @return the addition of this polynomial and the other polynomial.
     */
    public Polynomial addition(Polynomial other) {
        double[] sums = new double[Math.max(this.degree() + 1, other.degree() + 1)];
        for (int i = 0; i <= this.degree(); i++) {
            sums[i] += this.coefficients[i];
        }
        for (int i = 0; i <= other.degree(); i++) {
            sums[i] += other.getCoefficients()[i];
        }
        return new Polynomial(sums);
    }
```

Polynomials

```
public class Polynomial {

     /**
     * multiplies two polynomials.
     * @param other - the polynomial to be multiplied with this
     * @return the product of this polynomial and the other polynomial.
     */
    public Polynomial multiplication(Polynomial other) {
        double[] products = new double[this.degree() + other.degree() + 1];
        for (int i = 0; i < products.length; i++) {
            for (int j = 0; j <= i; j++) {
                if (j <= this.degree() && (i - j) <= other.degree()) {
                    products[i] += this.coefficients[j] * other.getCoefficients()[i - j];
                }
            }
        }
        return new Polynomial(products);
    }
```

Polynomials

```
public class Polynomial {
     /**
     * Subtracts two polynomials.
     * @param other - the polynomial to be subtracted from this
     * @return the subtraction of the other polynomial from this polynomial.
     */
    public Polynomial subtraction(Polynomial other) {
        double[] d = { -1 };
        Polynomial temp = new Polynomial(d);
        return this.addition(other.multiplication(temp));
    }

    /**
     * Derives a polynomial
     * @return the derivative of this polynomial.
     */
    public Polynomial derivative() {
        if (this.degree() == 0) {
            return new Polynomial();
        }
        double[] diff = new double[this.degree()];
        for (int i = 0; i < diff.length; i++) {
            diff[i] = this.coefficients[i + 1] * (i + 1);
        }
        return new Polynomial(diff);
    }
```

Polynomials

```
public class Polynomial {

    /**
     * Compares two polynomial.
     * @param other
     * @return true if and only if other is a polynomial whose coefficients are equal to this polynomial's.
     */
    public boolean equals(Polynomial other) {
        if (this.degree() != other.degree()) {
            return false;
        }
        for (int i = 0; i <= this.degree(); i++) {
            if (this.coefficients[i] != other.getCoefficients()[i]) {
                return false;
            }
        }
        return true;
    }
```

Polynomials

```
public class Polynomial {
    /**
     * Returns the string representation of this polynomial
     * @return string representation of this polynomial.
     */
    public String toString() {
        StringBuilder s = new StringBuilder();
        for (int i = this.coefficients.length - 1; i >= 0; i--) {
            s.append(this.coefficients[i] + "*x^" + i);
            if (i > 0) {
                s.append(" + ");
            }
        }
        return s.toString();
    }
    public static void main(String[] args) {
        double[] d = { 1, 1, 1 };
        Polynomial p = new Polynomial(d);
        Polynomial q = new Polynomial(p);
        System.out.println(p.equals(q)); // true
        System.out.println(p == q); // false
        System.out.println(p.evaluate(1)); // 3.0
    }
}
```

Lamp

- For us to truly understand the power of OOP let us define a lamp together.
- What are the characteristics of a Lamp?
- How would you define a Lamp?
- What operation a Lamp will have?
- How to compare between Lamps?
- How to represent Lamps?
- What values I would want to info about?
- What values I would want my user to get access to?