

# JAVA REVISITED

---

CSC 212: Data Structures

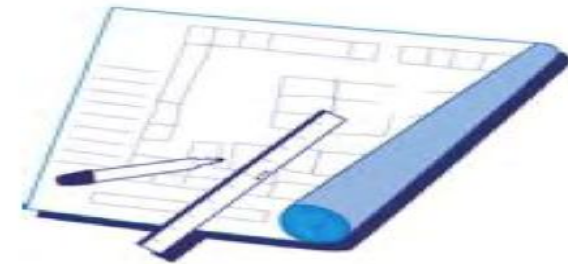
# Objective

- Object Oriented Programming (OOP): What, Why, How
- Analyzing and Designing OO Programs (Objects & Classes)
- Java Syntax, Java Program Skeleton
- Analyzing and Designing a Program
- Preparing Classes.

# Object-Oriented Design Principles

- **Abstraction**

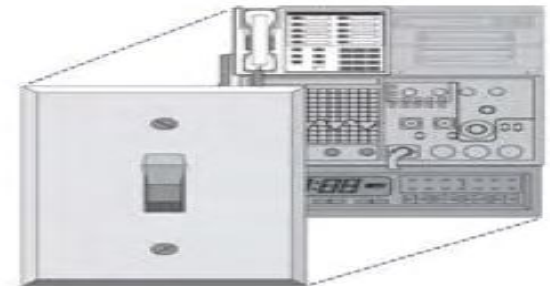
- To distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language



Abstraction

- **Encapsulation**

- Different components of a software system should not reveal the internal details of their respective implementations.



Encapsulation

# Object-Oriented Design Principles

- **Inheritance**

- Properties of a data type can be passed down to a subtype. We can build new types from old ones
- We can build class hierarchies with many levels of inheritance
- Allows the design of general classes that can be specialized to more particular classes, with the specialized classes reusing the code from the general class.

# Object-Oriented Design Principles

- **Polymorphism**

- Polymorphism is the capability of a method to do different things based on the object that it is acting upon.
- Following concepts demonstrate different types of polymorphism in java.
  - Method Overloading
  - Method Overriding

# Object-Oriented Design Principles

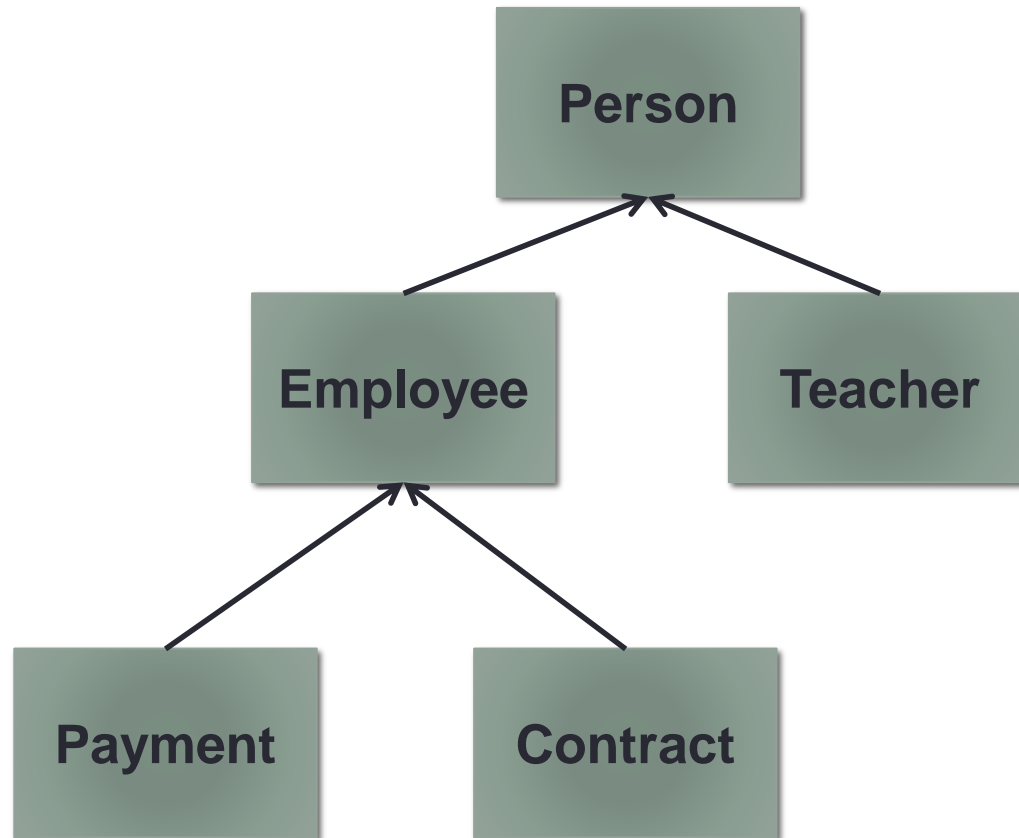
- **Overloading :**

- Allows the same method name to have multiple meanings or uses.
- Overloading of methods allows to use the same name to perform different actions depending on parameters.
- Overloaded methods are differentiated only on the number, type and order of parameters, not on the return type of the method.

- **Overriding:**

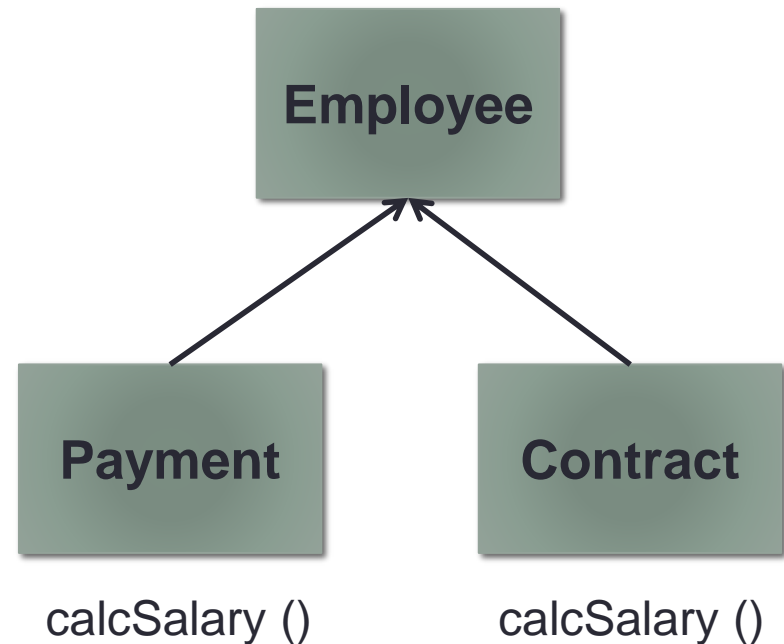
- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.

# Inheritance



# Polymorphism

- The derived classes override the definitions to provide unique behavior for each specific type of Employee
- ***Dynamic binding :***
  - Which method to call?
  - Binding occurs at run time, based on the type of object.





# Designing Object Oriented

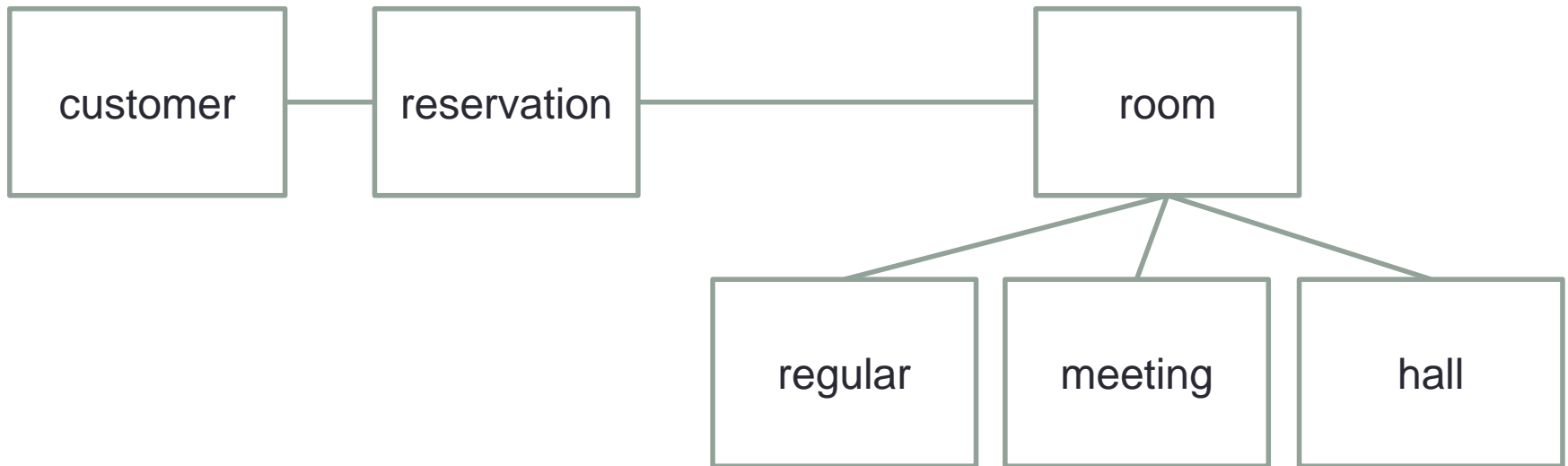
- It's a process of planning a computer program where objects will interact with each other to solve specific problems.

- **Example :**

A hotel contains a number of Residential rooms, Meeting rooms, Wedding halls where the customer can reserve any of them by choosing type of room and specify date and time. You are required to design reservation system for Hotel. where it can offer the available rooms or halls and allows the customer to reserve one of them.

# Designing Object Oriented

- Objects = nouns  
Functions to be encapsulated



# Class

- **Class:** a unit of abstraction in an object oriented (OO) program
- **Represents similar objects**
  - Its instances
- **A kind of software module:** Describes its instances' structure (properties). Contains methods to implement their behavior.

# Class Structure

- Two Main Sections
  - Variables: can be a simple data type or another class
    - Represent the State of the Class
    - Define Data represented in an Class
    - Associations
  - Operations :
    - A procedural abstraction used to implement the behavior of a class.

# Skelton of Class

```
class ClassName {  
  // Attributes  
  Type Name;  
  //Constructor  
  public ClassName (Type n,...) {  
  }  
  //Setter  
  public void setName(Type n)  
  {  
  }  
  //Getter  
  public Type getName()  
  {  
  }  
  //Operations  
}
```

# Java Rules

- The name of a class is the same as the name of the file (which has .java extension)
- All Java applications must have a main method in one of the classes
- Execution starts in the main method
- Body of a method is within { }
- All other statements end with semicolon ;

# Variables and data type

- `String name="Ali";`
- `name` is a variable of type `String`
- we have to declare variables before we use them
- Variables can be declared anywhere within a block
- Variable name
  - use meaningful names `numberOfBricks`
  - start with lower case
  - capitalize first letter of subsequent words
  - can not start with digit
  - can not use keyword

# Data types

- int 4 byte integer (whole number)
  - range -2147483648 to +2147483648
- float 4 byte floating point number
  - decimal points, numbers outside range of int
- double 8 byte floating point number
  - 15 decimal digits (float has 7) so bigger precision and range
- char 2 byte letter
- String string of letters
- boolean true or false (not 1 or 0)



# Output

- Java provides print methods in the class `System.out` (don't need to import)
- `println(name);`
  - prints out what is stored in name, then goes to a new line
- `print(name);`
  - prints out what is stored in name, but does not start a new line
- `print("My name is " + name);`
  - put text in quotes
  - use + to print more than one item

# Methods

- methods break down large problems into smaller ones
- your program may call the same method many times
  - saves writing and maintaining same code
- methods take parameters
  - information needed to do their job
- methods can return a value
  - must specify type of value returned

# Methods

Method Header :

**visibility** [static] returnType methodName(parameterList)

- **visibility:**
  - public  
accessible to other classes
  - protected  
accessible to classes which inherit from this one
  - private
- **static** keyword:
  - use when method belongs to class as whole not object of the class

# Methods

- **returnType:**
  - type of the value that the method calculates and returns (using return statement)
  - can return object
  - if nothing returned, use keyword void
- **methodName:** Java identifier; use meaningful name which describes what method does!
- formal parameter list:
  - information needed by method
  - pairs of type name
- examples:
  - addNums(int num1, int num2)
  - drawPerson(boolean isBald, String name, int
  - use empty brackets if method has no parameters  
printHeadings())

# Method Body

- use curly brackets to enclose method body  
all your code goes in here
- it should return a value of appropriate type
- - must match type in method header
  - nothing is executed after **return** statement
  - if method returns **void**, can omit **return** statement
- method will automatically return at closing }

# Calling Method

- methods will not run unless called from elsewhere
- ◦ a statement in `main()` method could call another method
- ◦ this method could call a third method .....
- ▶ class methods are called with the form:
- `ClassName.methodName(parameters);`
- ◦ omit `ClassName` if called in same class
- ▶ method name and parameters must match the
- ▶ if the method returns a value, it can be stored in a variable or passed to another method

# Passing parameters

```
public class Test{  
    public static void main(String args[]) {  
        int a =5;  
        System.out.println("Before Modify: " + a);  
        modify(a);  
        System.out.println("After Modify: " + a);  
    }  
    public static void modify(int x)  
    {  
        x++;  
    }  
}
```

# Passing parameters

```
class Student {  
    String name;  
    int id;  
    public Student(String name) {  
        this.name = name;  
    }  
    public String toString() {  
        return name;  
    }  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
}
```

```
public class Test{  
    public static void main(String args[]) {  
        Student a = new Student ("Ahmed");  
        System.out.println("Before Modify: " + a);  
        modify(a, "Ahmed Alli");  
        System.out.println("After Modify: " + a);  
    }  
    public static void modify(Student s,String n)  
    {  
        s.setName(n);  
    }  
}
```



# Constructors

- A ***constructor*** is a special kind of method that is used to initialize newly created objects.
- Java has a special way to declare the constructor and a special way to invoke the constructor.
- A class may contain more than one constructor.
- Constructor has the **same name as the class name**.
- invoked **“automatically”** at the time of object creation.

# Object

- a new object is created from a defined class by using the **new** operator.
- The **new** operator creates a new object from a specified class and returns a *reference* to that object.
- Example

Student S1=**new** Student ("Ahmed ",29211212);

- In order to access attributes of a given object:
  - use the dot (.) operator with the object reference (instance variable) to have access to attributes' values of a specific object.
  - *ObjectName.VariableName*
  - *ObjectName.MethodName(parameter-list)*

-

# Casting

- Taking an Object of one particular type and “turning it into” another Object type.
- Primitive Type Casting:

```
public class MainClass{  
    public static void main(String[] argv){  
        int a = 100;  
        float f=100.00f;  
        long b = a; // Implicit cast, an int value always fits in a  
        long  
        int c = (int)f; // Explicit cast, the float could lose inf  
    }  
}
```

# Casting

Object Type Casting:

```
interface Vehicle {
}
class Car implements
Vehicle {
void carMethod()
{}
}
class Ford extends Car {
void fordMethod () {}
}
```

```
Car c = new Car();
Ford f = new Ford();
c=f;
```

```
c.carMethod();
```

//How can I access fordMethod ()

```
if (c instanceof Ford)
((Ford) c).fordMethod ();
```

# Extending Class

- Inheritance in Java is implemented by **extending** a class

```
public class subClass extends superClass {
```

- We then continue the definition of `subClass` as normal
- However, implicitly `subClass` contains all the data and operations associated with `superClass`. Even though we don't see them in the definition

# Interfaces

- **Interface** is a collection of method declarations with no bodies.

```
public interface NameOfInterface
```

```
{  
    //Any number of final, static fields  
    //Any number of abstract method declarations\  
}
```

- Methods of an interface are always empty (that is, they are simply method signatures).
- When a class implements an interface, it **must** implement **all** of the methods declared in the interface.

# Example

```
public interface Shape
{
    public double calculate_Area();
}
public class Rectangle implements Shape
{
    double w,h;
    public double calculate_Area()
    {
        return w*h;
    }
}
```

# To Do

- ► Read Chapter 1 & 2 of the Textbook.
- ► Install eclipse or any java editor you fancy.



# JAVA REVISION

---

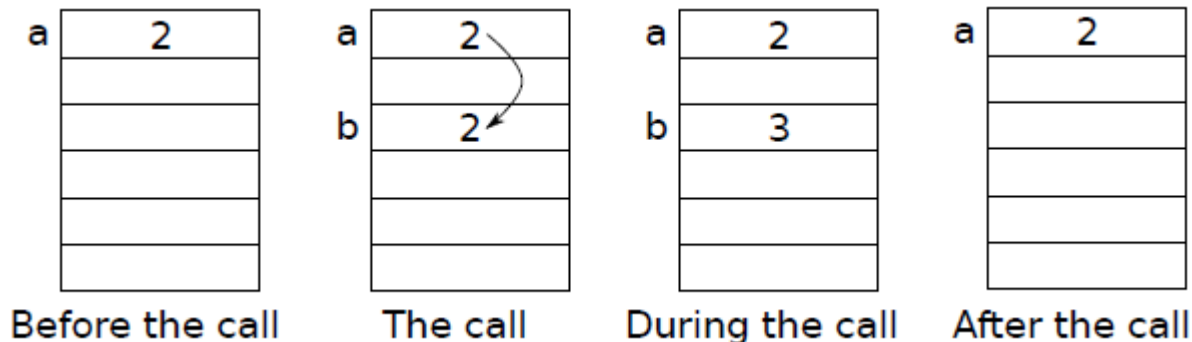
Wrappers

# Argument passing

- All arguments in Java are passed by **value**.
- The method creates a copy of the argument, and manipulate the copy (not the argument itself).
- It is therefore impossible to change the argument from within the method.
  - This means that it is impossible to change the value of an argument of a **primitive type** (int, double, char, ...).
  - **Reference variables** cannot be changed inside methods as well, but their content can be changed.

# Argument passing: primitive type variables

```
public void myF(int b) {  
    b++;  
}  
  
public void test() {  
    int a = 2;  
    myF(a);  
    System.out.println(a);  
}
```

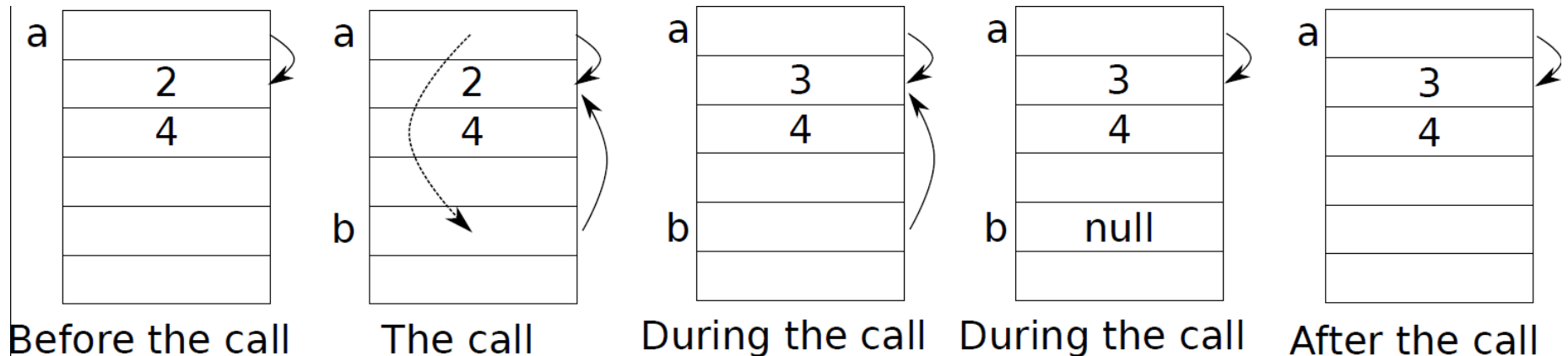


# Argument passing: reference variables

```

public void myF(int [] b) {
    b[0]++; b = null;
}
public void test() {
    int [] a = {2,4};
    myF(a);
    System.out.println(a[0]);
}

```



# Object class

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.

# Wrappers

- The Java primitive types are NOT classes, and thus cannot be used as easy as objects
  - If I make an array of Object or any other class, primitive types cannot be stored in it.
- **Wrapper classes** allow to get around this problem
  - Wrappers are classes that “wrap” objects around primitive values, thus making them compatible with other Java classes
    - We cannot store an int in an array of Object, but we can store an Integer

# Java wrappers

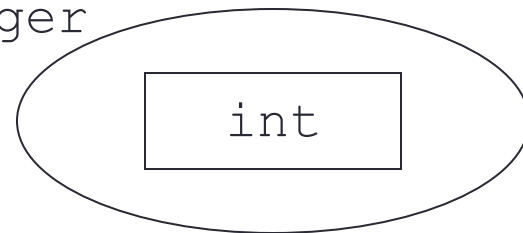
- Java offers a set of standard wrapper classes for all primitive data types.
  - Integer i, j, k;  
    i = new Integer(20); // or i=20; is also correct  
    j = new Integer(40);
- Java wrappers are immutable, which means that their content cannot be changed once created. Hence they cannot be used for parameter passing.

Primitive type	Standard wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

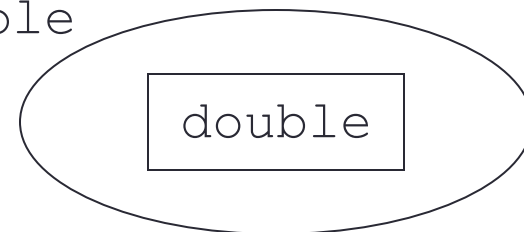
# Java wrappers

- The wrapper classes also provide extra useful functionality for these types
  - Ex: `Integer.parseInt(String s)` is a static method that enables us to convert from a `String` into an `int`
  - Ex: `Character.isLetter(char c)` is a static method that tests if a letter is a character or not

Integer



Double





# Java wrappers

- J2SE 5.0 introduced the feature of auto-boxing and auto-unboxing of standard primitive wrappers. This means that primitive types are automatically converted to their wrappers, and vice-versa, as necessary.

```
Integer a = 3; // 3 is auto-boxed to the object a
int b = a + 2; // a is auto-unboxed to its primitive value
Double x = b + 1; // The result is auto-boxed to a wrapper of double
Double y = x + a; // x and a are unboxed, their sum is computed and boxed to
y
```

# Wrappers

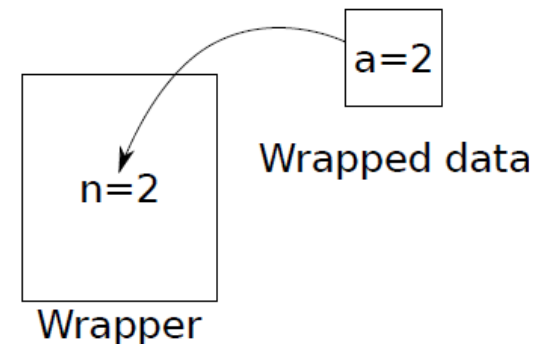
- Since all arguments are passed by value in Java, how can we solve problems like the following:
- **Example:** we want to write a method `checkModify` that takes as input an integer parameter `n` and performs the following:
  - If `n` is odd, the method returns `false`.
  - If `n` is even, the method returns `true` and changes `n` to `n/2`.
- Since the return statement returns a boolean, we cannot return the value `n/2`.
- One possible solution is to use wrappers (not Java wrappers).

# Creation and use of a wrapper

- A wrapper is a class that is used to store an object of a certain type and hence allows modifying the object.

We can create a simple wrapper for type int as follows:

```
public class IntWrapper {  
    private int n;  
    public IntWrapper(int n) {  
        this.n = n; }  
    public void setN(int n) {  
        this.n = n; }  
    public int getN() {  
        return n; }  
}
```



# Creation and use of a wrapper

Wrappers are generally used for two purposes:

1. To store primitive types in classes that require objects.
2. For passing parameters that can be changed inside a method. This use involves usually 5 steps:
  1. Calling method: create a wrapper object and put the data inside it, then pass it as a parameter.
  2. Called method: take the data from the wrapper. (use getters)
  3. Called method: modify the data.
  4. Called method: update the data of the wrapper. (use setters)
  5. Calling method: take the updated value from the wrapper.

Here we use the class *IntWrapper* to write the method *checkModify*:

```
public void testCheckModify() {
    int n = 8;
    IntWrapper w = new IntWrapper(n);
        // Step 1
    checkModify(w);
    n = w.getN(); // Step 5
}
```

```
public void checkModify(IntWrapper w
    ) {
    int n = w.getN(); // Step 2
    if (n % 2 == 0) {
        w.setN(n/2); // Step 3, 4
        return true;
    } else
        return false;
}
```

# JAVA GENERICS

---

CS212:Data Structures

# Outline

- Problem definition
- Solution using the casting
- Solution using Generics

# Problem

- Data structures are used in various applications to store virtually any type of data.
- How can we adapt the data structure code to new data types?
- A solution is to rewrite the code each time we need to store a new data type (unrealistic).
  - Example: list of integers, list of doubles, list of characters.
- The problem with this solution is : it is time and effort consuming and also error-prone.
- In the rest, we will see two solutions to this problem,
  - one based on casting
  - the second on generic classes.

## First solution: casting

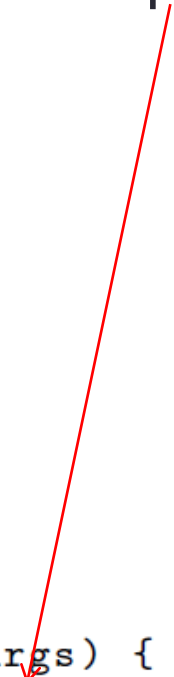
- The first solution takes advantage from the fact that **all classes in Java derive from the class Object**.
- Instead of rewriting the code for every new class, we will write a single box class for data of type Object and use it to store all class data types.
- Putting the data inside the data structure is simple and does not require any special treatment.
- Retrieving the data from the data structure requires casting from Object to the original data type.



# First solution: casting

- This solution does not handle primitive data types as these are not classes. You need to use wrappers.

```
public class Box {  
    private Object object;  
    public Box (Object object)  
    {this.object = object;  
    }  
  
    public Object get()  
    {return object;  
    }  
}  
  
public class BoxTest1 {  
    public static void main(String[] args) {  
        Box integerBox = new Box(new Integer(10));  
        Integer someInteger=(Integer)integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```



# First solution: casting

- If we pass a string as a parameter to the method, we need to change the **casting** type otherwise we get runtime exception (the compiler does not detect it).

```
public class BoxTest2 {  
    public static void main(String[] args) {  
        Box integerBox = new Box(new String("10"));  
        Integer someInteger=(Integer)integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```

Exception in thread "main" java.lang.ClassCastException:  
java.lang.String cannot be cast to java.lang.Integer at  
BoxTest2.main(BoxTest2.java:4)

# First solution: casting

- Advantages:
  - This solution is simple as it requires the code only once.
  - It handles class data types as well as primitive data types through the use of wrappers.
- Disadvantages:
  - It allows for mixing data types, which is a bad practice.
  - It may lead to casting errors, when the data type inserted is different from the one we want to retrieve.
  - Casting errors are difficult, since they are runtime errors which are far harder to detect than compilation errors.

# A Better Solution: Generics

- It is possible to “parameterize” a class with the type of the object we expect it to contain using the `<>` syntax
  - Declares an attribute of type `T` instead of `Object`.
  - Compiler can now ensure only objects of the specific type are used.

# Writing a generic class

```
public class Box<T> {  
    private T t;    // T stands for Type  
    public Box(T t)  
    {this.t = t;  
    }  
    public T get()  
    {return t;  
    }  
}
```

- Box<T> is a generic class.
- Use the <> syntax in the class definition,.
- This is similar to declaring parameters in a method
  - Called **Formal Type Parameters**
- The <T> declares that a type must be used when an instance is created
  - The type is then replaced everywhere where the 'T' is used in the class definition

# Using a generic class

```
public class BoxTest3 {  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>(new Integer(10));  
        Integer someInteger = integerBox.get(); // no casting  
        System.out.println(someInteger);  
        Box<String> stringBox = new Box<String>("ABC");  
        String someString = stringBox.get();  
    }  
}
```

- Remark: no need to use the casting

# Using a generic class

```
public class BoxTest4 {  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>(new String( 10 ));  
        Integer someInteger = integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```

BoxTest4.java:3: Box(java.lang.Integer) in  
Box<java.lang.Integer> cannot be applied to (java.lang.String)

# A Better Solution: Generics

- **1) Type-safety** : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- **2) Type casting is not required**: There is no need to typecast the object.
- **3) Compile-Time Checking**: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.



# ToDo

- Read the text book: section 2.5

# RECURSION

---

CSC212: Data Structures

# Recursion

- ❖ Sometimes, certain statements in an algorithm are repeated on different sizes of an input instance.
- ❖ Repetition can be achieved in two different ways.
  - **Iteration:** uses **for** and **while** loops
  - **Recursion:** function calls itself

## Example -1:

- Factorial Function
- Factorial function of any integer  $n$  is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n.(n-1).(n-2) \dots 3.2.1 & \text{if } n \geq 1 \end{cases}$$

- $4! = 4 * 3 * 2 * 1$
- $4! = 4 * 3!$

# Recursion

## Example -1:

– Factorial function of any integer  $n$  is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 & \leftarrow \text{Base Case} \\ n(n-1)! & \text{if } n \geq 1 & \leftarrow \text{Recursion Case} \end{cases}$$

- This is recursive definition. It consists of two parts:
  - i. **Base case**
  - ii. **Recursive case**

## Important:

Every recursion must have at least one base case, at which the recursion does not recur

# Recursion

## Example -1 (Continue) :

- It can be written as:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 & \leftarrow \text{Base Case} \\ n \cdot \text{fact}(n-1) & \text{if } n \geq 1 & \leftarrow \text{Recursion Case} \end{cases}$$

where  $\text{fact}(n)$  is the function that calculates  $n!$ .

# Recursion

## Example -1 (Continue) :

- Implementation

### Recursive:

```
public static int recursiveFact(int n)
{
    if(n==0) return 1;
    else
        return n*recursiveFact(n-1);
}
```

### Iterative:

```
public static int iterativeFact(int n)
{
    int fact = 1;
    for(i = 1; i <= n; i++)
        fact=fact*i;
    return fact;
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

- Recursive trace for `recursiveFact(4)`

```
public static int recursiveFact(int n)
{
    if(n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

- Recursive trace for `recursiveFact(4)`

`recursiveFact(4)`

$4 * ? = ?$

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```



# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

- Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?



recursiveFact(3)

3\*?=?

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?

recursiveFact(3)

3\*?=?

recursiveFact(2)

2\*?=?

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?

recursiveFact(3)

3\*?=?

recursiveFact(2)

2\*?=?

recursiveFact(1)

1\*?=?

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?

recursiveFact(3)

3\*?=?

recursiveFact(2)

2\*?=?

recursiveFact(1)

1\*?=?

recursiveFact(0)

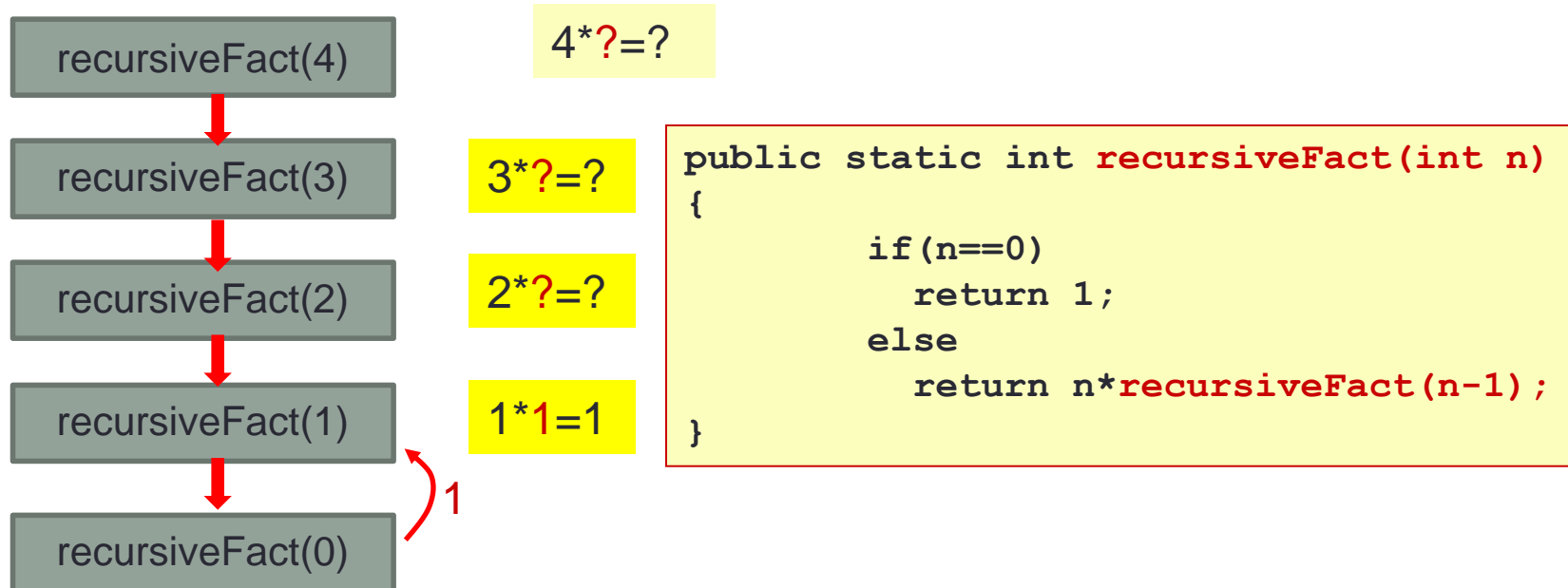
```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

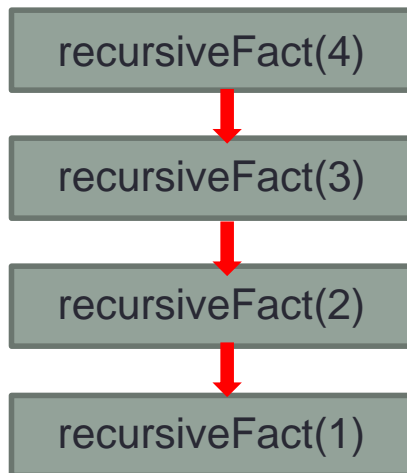


# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`



4\*?=?

3\*?=?

2\*?=?

1\*1=1

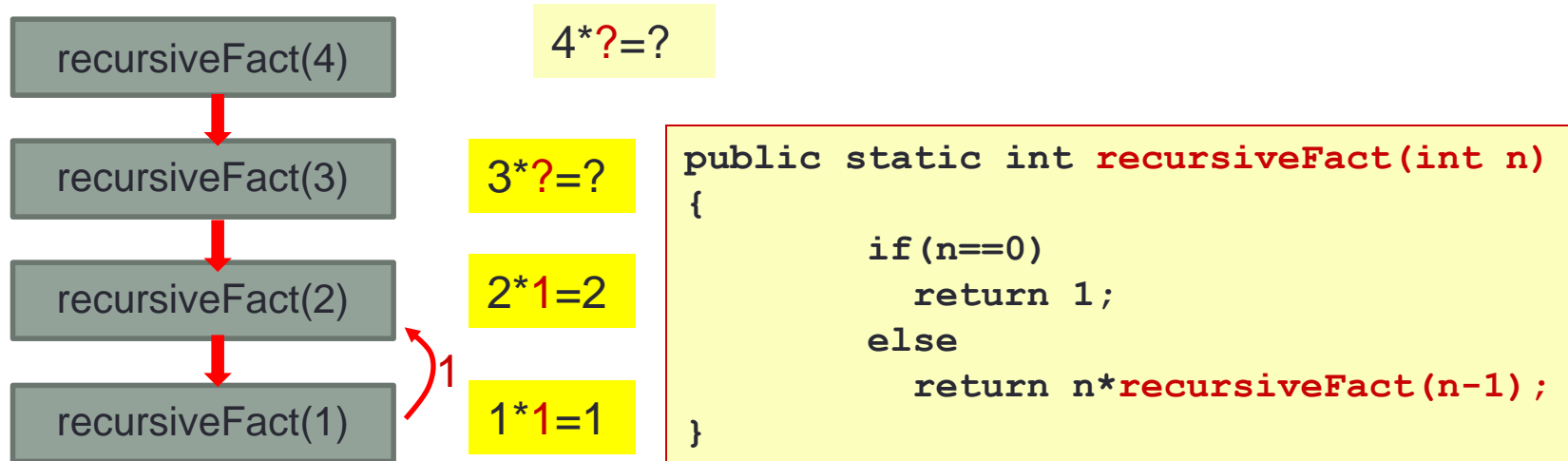
```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`



# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?

recursiveFact(3)

3\*?=?

recursiveFact(2)

2\*1=2

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```



# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`

recursiveFact(4)

4\*?=?

recursiveFact(3)

3\*2=6

recursiveFact(2)

2\*1=2

2

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

- Recursive trace for `recursiveFact(4)`

recursiveFact(4)



recursiveFact(3)

4\*?=?

3\*2=6

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

– Recursive trace for `recursiveFact(4)`



```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace

- ❖ A graphical representation of recursive calls.
- ❖ It is used to analyze the algorithm.

## Example -2:

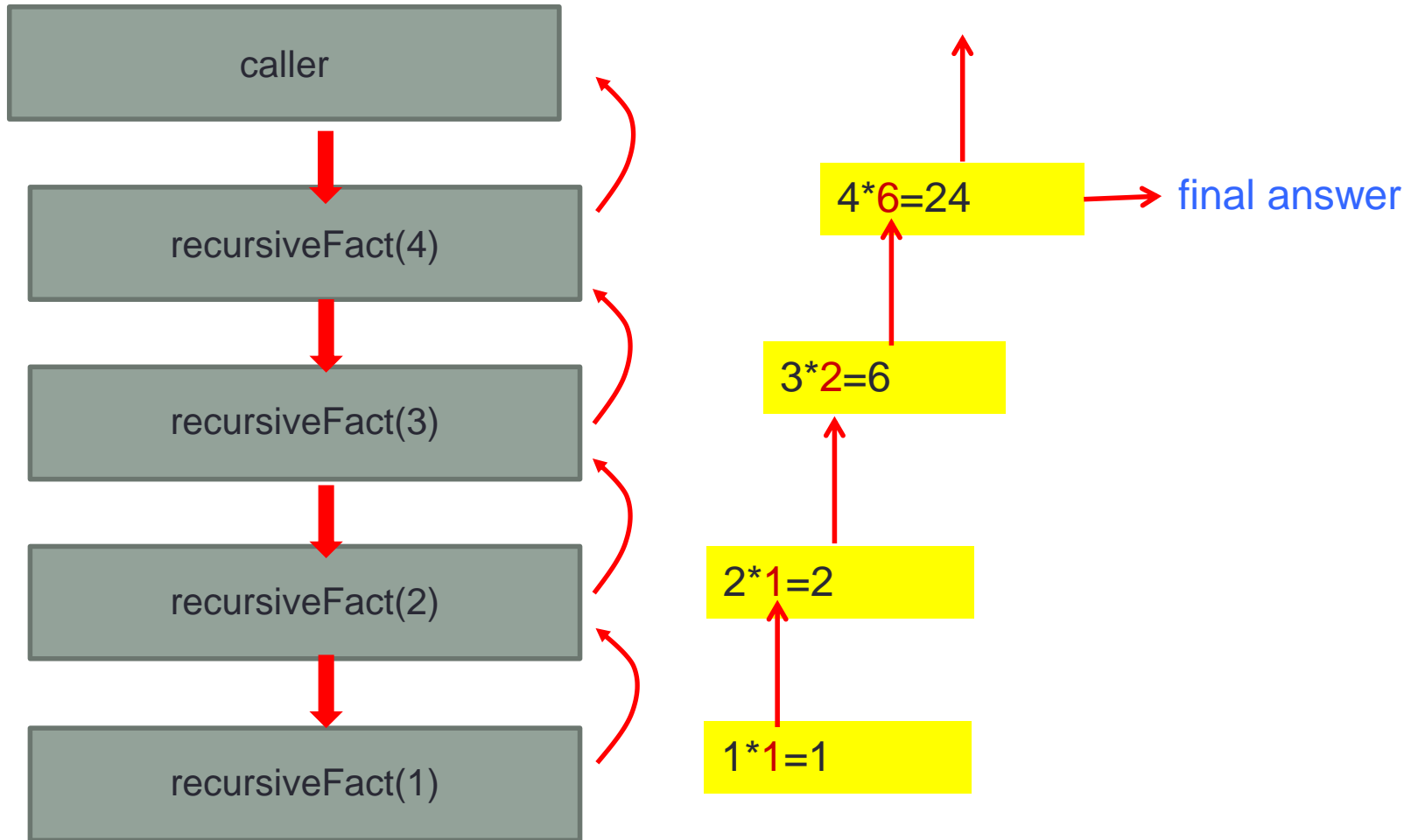
- Recursive trace for `recursiveFact(4)`

`recursiveFact(4)`

$4 * 6 = 24$

```
public static int recursiveFact(int n)
{
    if (n==0)
        return 1;
    else
        return n*recursiveFact(n-1);
}
```

# Recursive Trace



# Recursive Exercise

Calculate  $x^n$  using both iteration and recursion.  
(Assume  $x > 0$  and  $n \geq 0$ )

```
if (n==1)
    return x;
else
    return x * pow(x, n-1);
```

# Main Types of Recursion

- ❖ Linear Recursion

- ❖ Binary Recursion

# Linear Recursion

In this case a recursive method makes at most one recursive call each time it is invoked.

## Example – 3:

- **Problem:** Given an array  $A$  of  $n$  integers, find the sum of first  $n$  integers.
- **Observation:** Sum can be defined recursively as follows:

$$\text{Sum}(n) = \begin{cases} A[0] & \text{if } n = 1 \\ \text{Sum}(n - 1) + A[n - 1] & \text{if } n > 1 \end{cases}$$

→ Base Case  
→ Recursive Case



# Linear Recursion

## Example – 3 (Continued):

### – Algorithm

**Sum(A, n)**

**Input:** An integer array **A** and an integer  $n \geq 1$ , such that **A** has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in **A**.

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

return  $\text{Sum}(A, n-1) + A[n-1]$ ; → recursive case.

### Note:

- Base case should be defined so that every possible chain of recursive calls eventually reach a base case.
- Algorithm must start by testing a set of base cases.
- After testing for base cases perform a single recursive call.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$

A	
0	4
1	3
2	6
3	2
4	5

### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

    return  $A[0]$ ;                       $\rightarrow$  base case

else

    return  $\text{Sum}(A, n-1) + A[n-1]$ ;                       $\rightarrow$  recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$

**Sum(A,5)**

??+A[4]=??+5=?

A	
0	4
1	3
2	6
3	2
4	5

### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

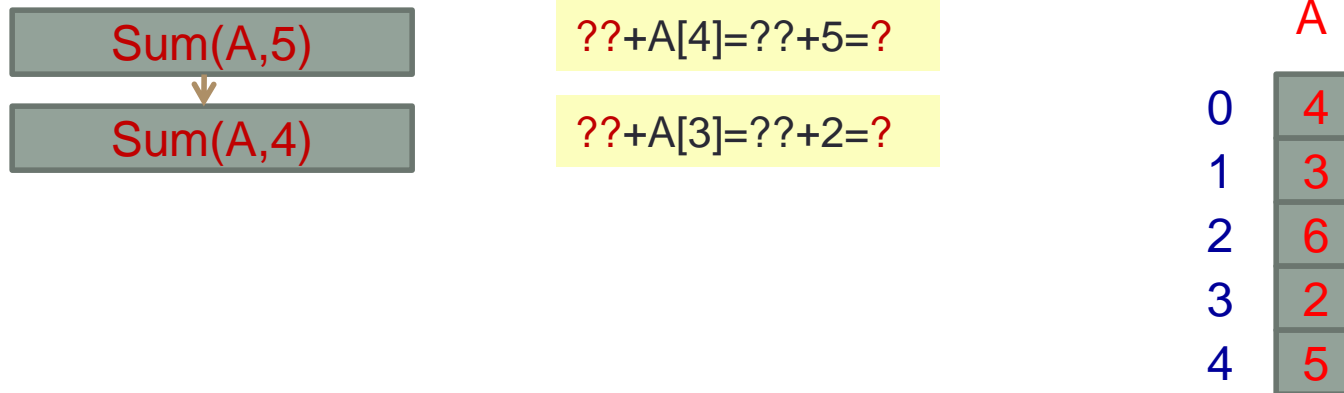
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

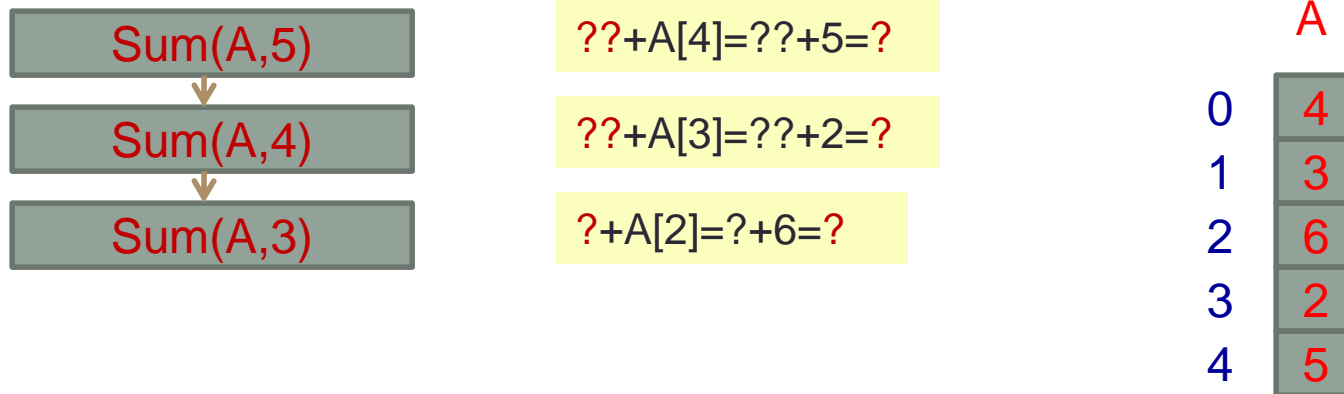
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### **Sum(A, n)**

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;  $\rightarrow$  base case

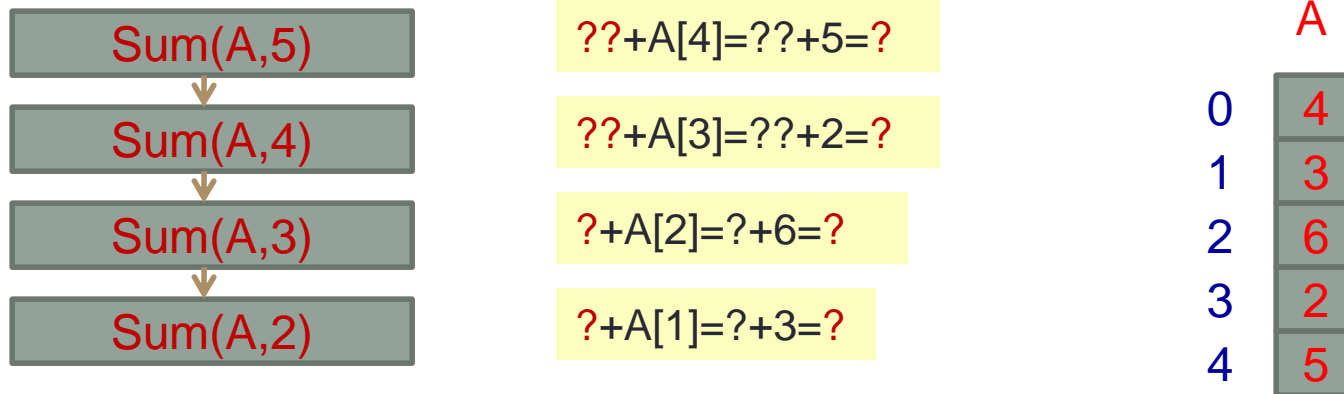
else

return  $\text{Sum}(A, n-1) + A[n-1]$ ;  $\rightarrow$  recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

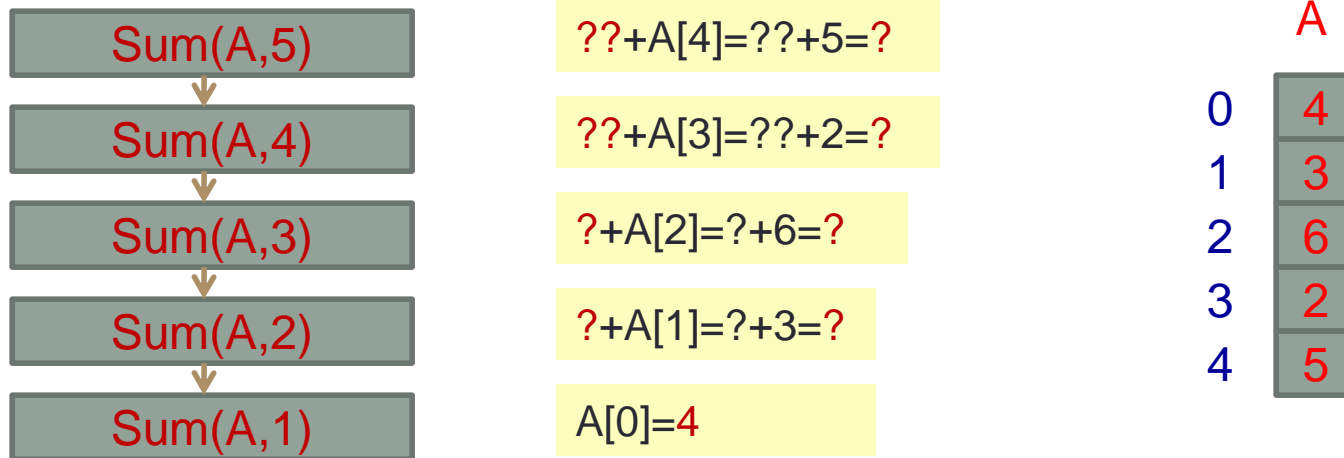
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

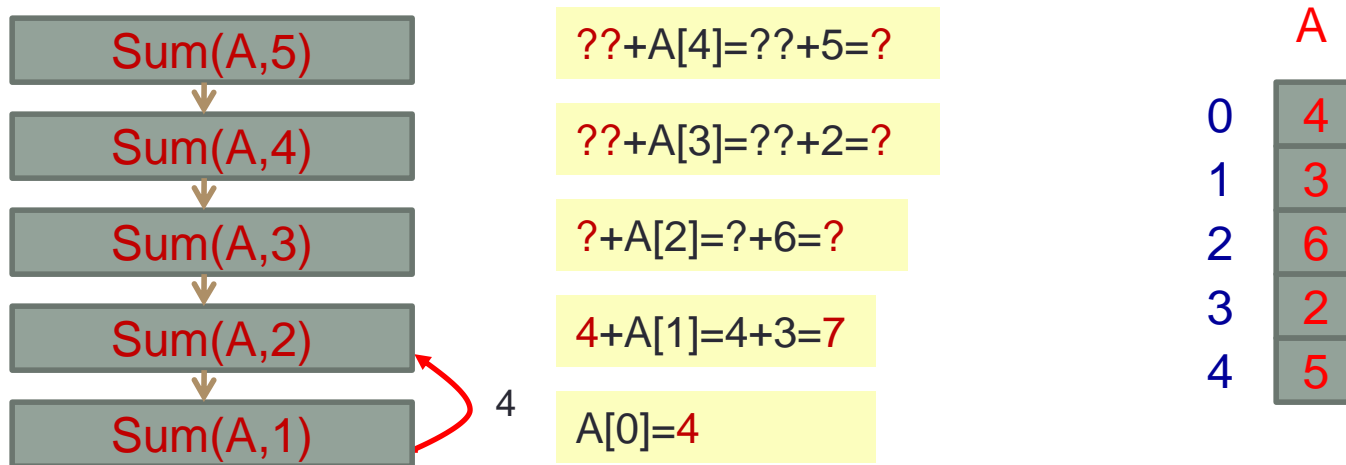
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

return  $\text{Sum}(A, n-1) + A[n-1]$ ;

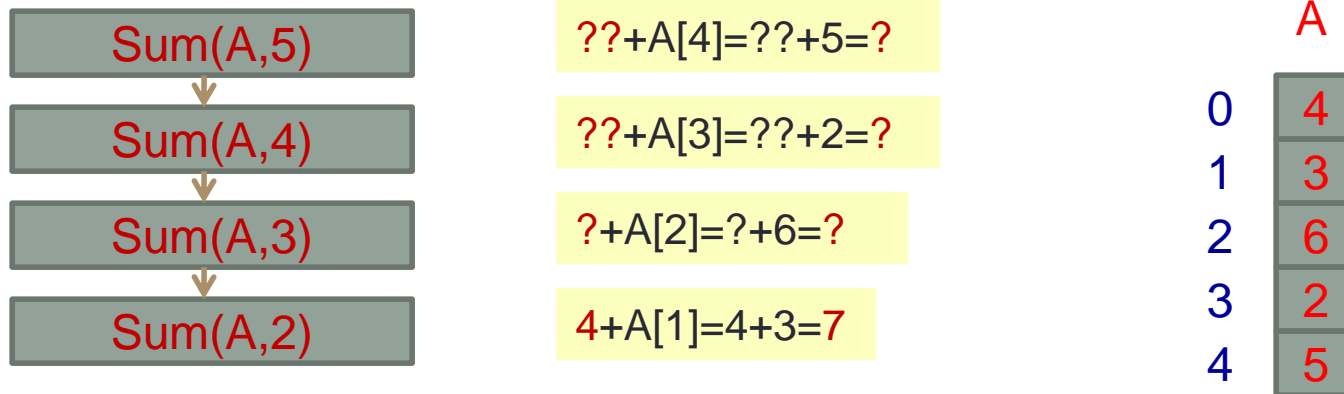
→ recursive case.



# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### **Sum(A, n)**

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

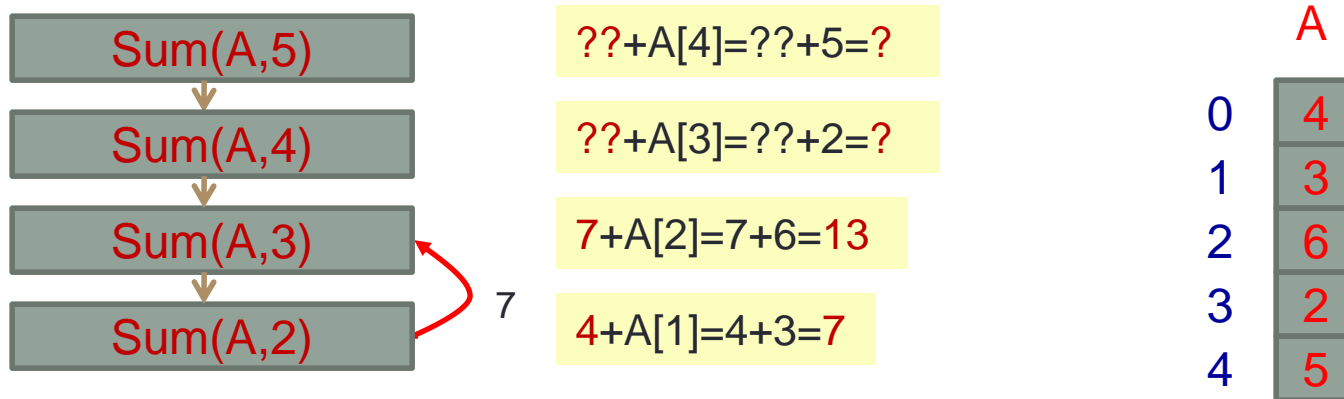
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

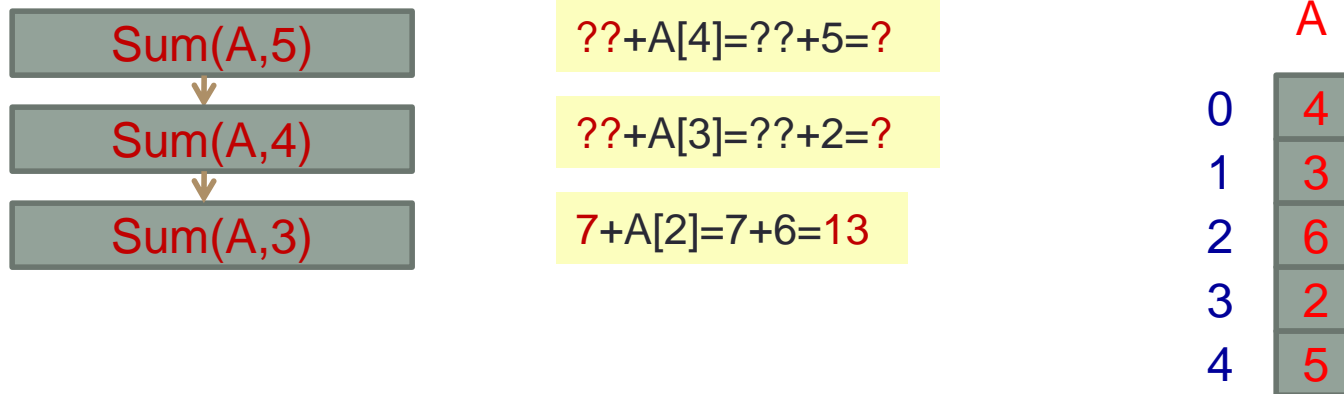
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### **Sum(A, n)**

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ; → base case

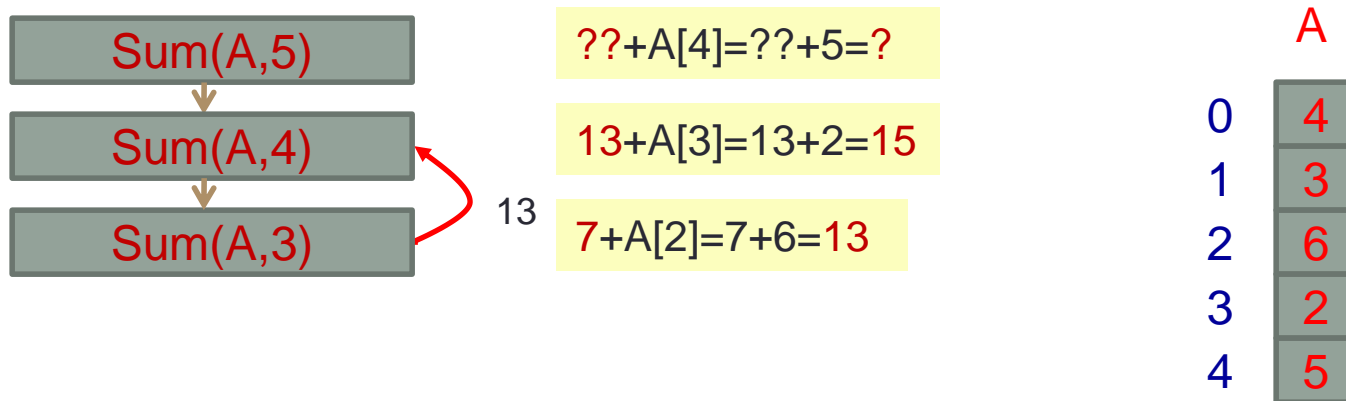
else

return  $\text{Sum}(A, n-1) + A[n-1]$ ; → recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### **Sum(A, n)**

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

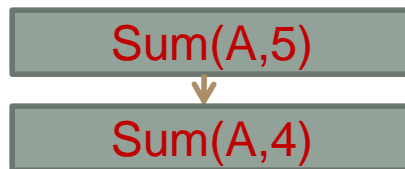
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



??+A[4]=??+5=?

13+A[3]=13+2=15

A	
0	4
1	3
2	6
3	2
4	5

### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

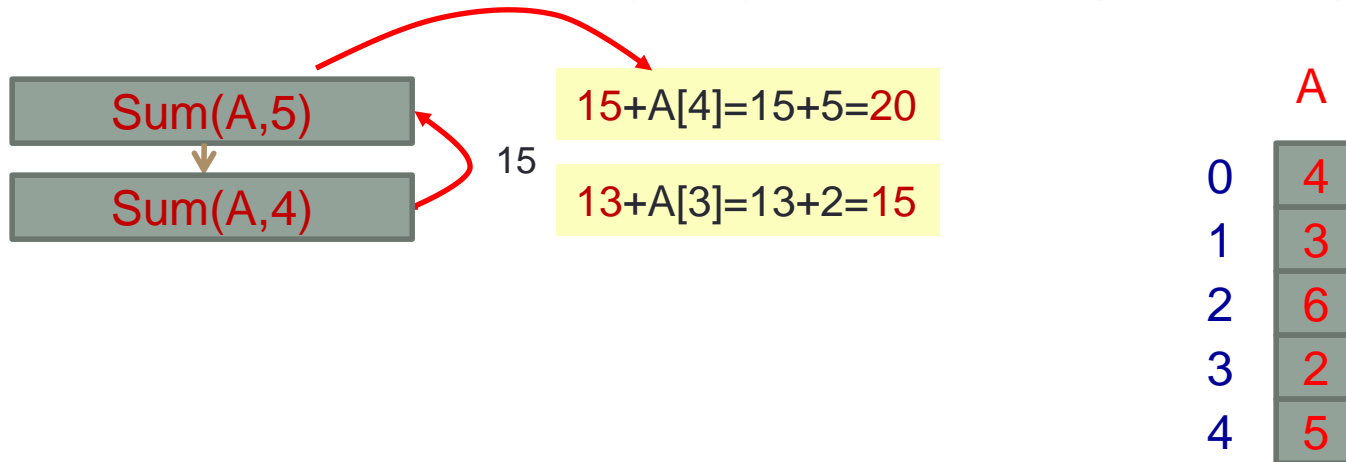
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### **Sum(A, n)**

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

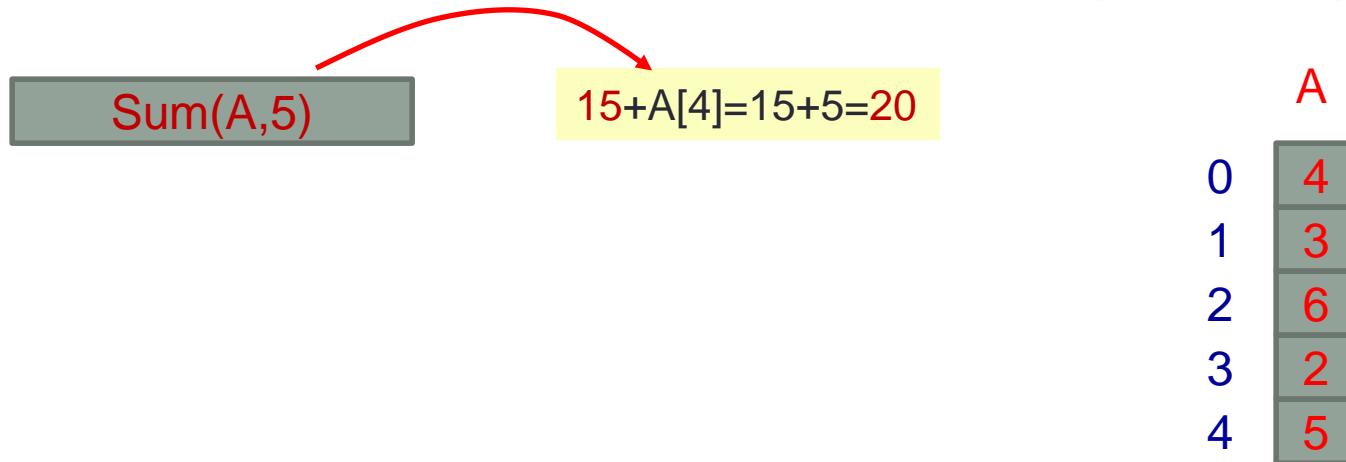
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Sum(A, n)

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$ ;

return  $A[0]$ ;

→ base case

else

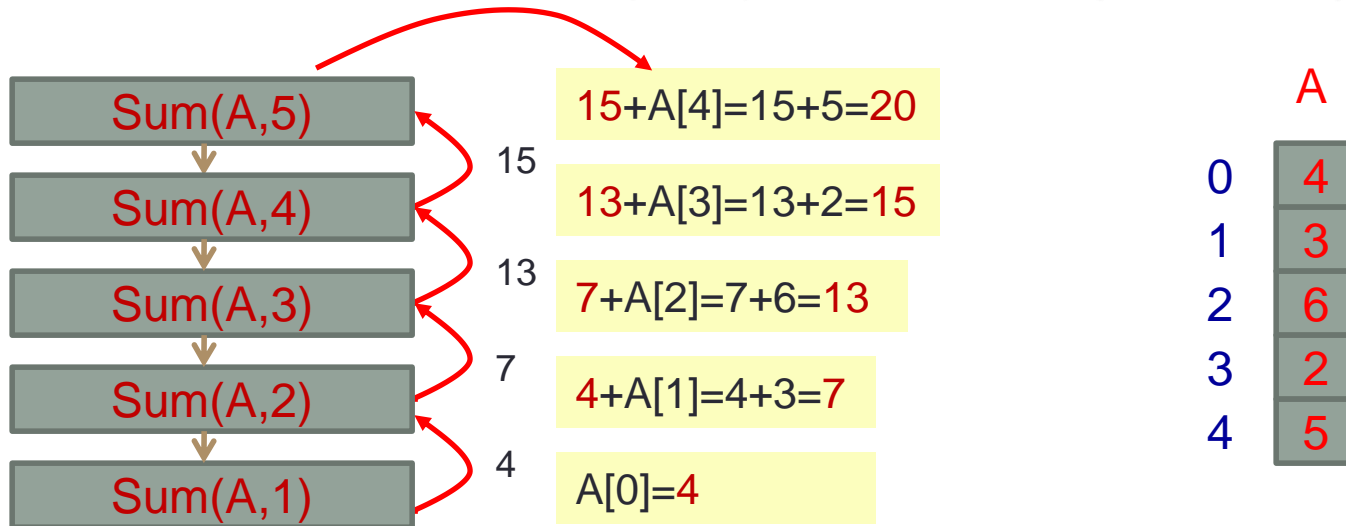
return  $\text{Sum}(A, n-1) + A[n-1]$ ;

→ recursive case.

# Linear Recursion

## Example – 3 (Continued):

- Recursive trace for  $\text{sum}(A, n)$ , where  $A = \{4, 3, 6, 2, 5\}$ ,  $n=5$



### Note:

- For an array of size  $n$ ,  $\text{Sum}(A, n)$  makes  $n$  calls.
- Each spends a constant amount of time.
- So time complexity is  $O(n)$ .



# Binary Recursion

In this case, a recursive algorithm makes two recursive calls.

## Example – 4

- **Problem:** Find the sum of  $n$  elements of an integer array  $A$ .
- **Algorithm:**
  - Recursively find the sum of elements in the **first half** of  $A$ .
  - Recursively find the sum of elements in the **second half** of  $A$ .
  - Add these two values

A	
0	6
1	5
2	3
3	2
4	8

### BinarySum( $A, i, n$ )

**Input:** An integer array  $A$  and an integer  $n \geq 1$ , such that  $A$  has at least  $n$  elements

**Output:** The sum of the first  $n$  integers in  $A$ .

**Processing:**

if  $n = 1$

return  $A[i]$ ;

Else

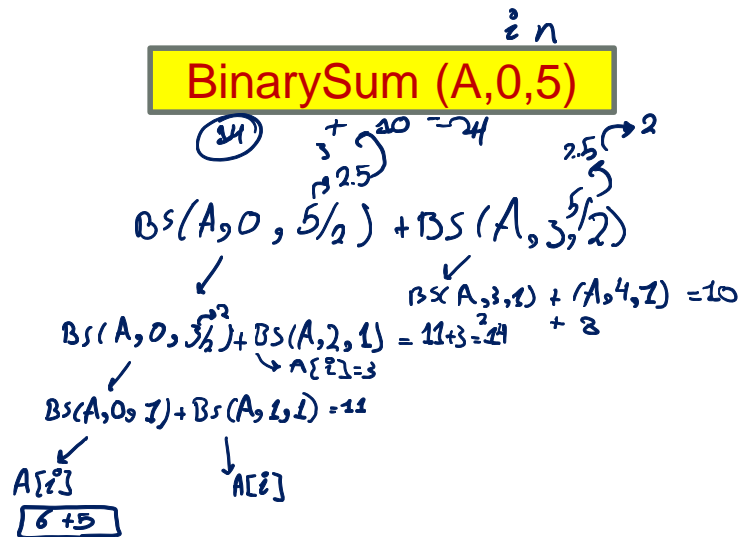
return BinarySum( $A, i, \lceil n/2 \rceil$ ) + BinarySum( $A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor$ );

# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

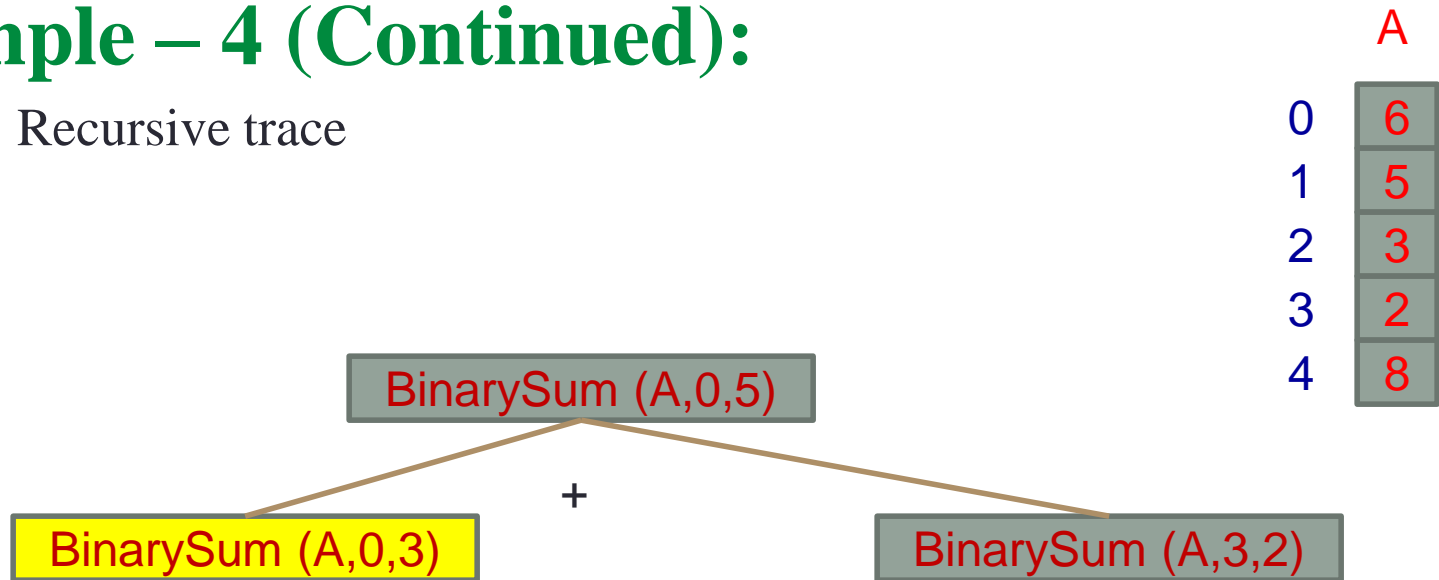
	A
0	6
1	5
2	3
3	2
4	8



# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

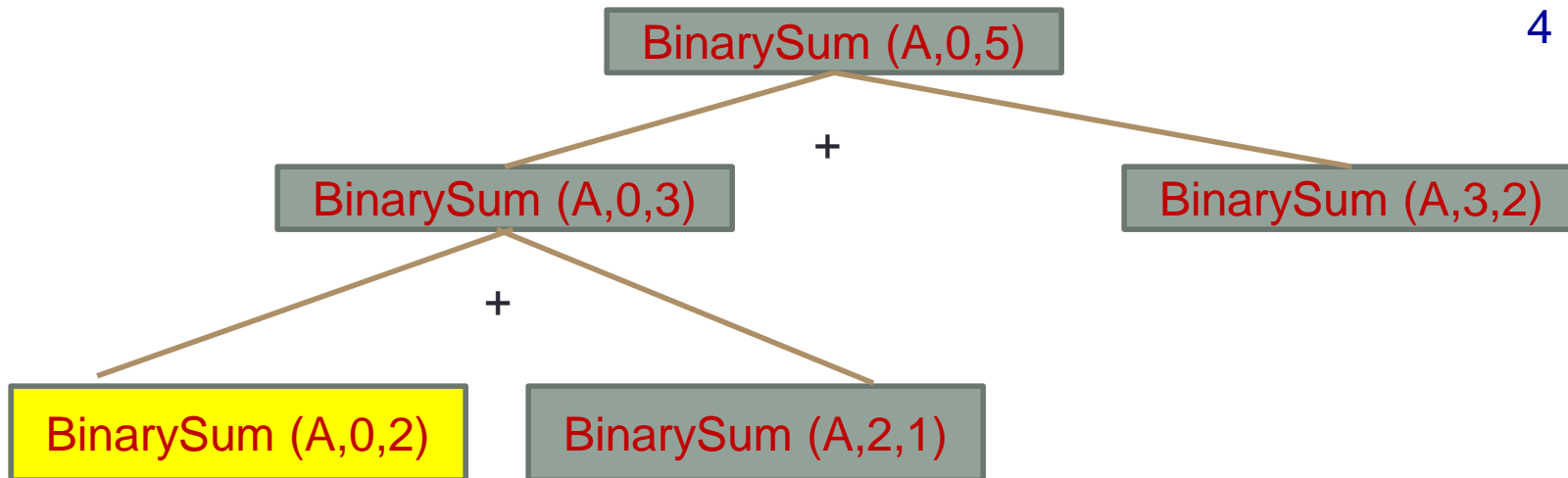


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

A	
0	6
1	5
2	3
3	2
4	8

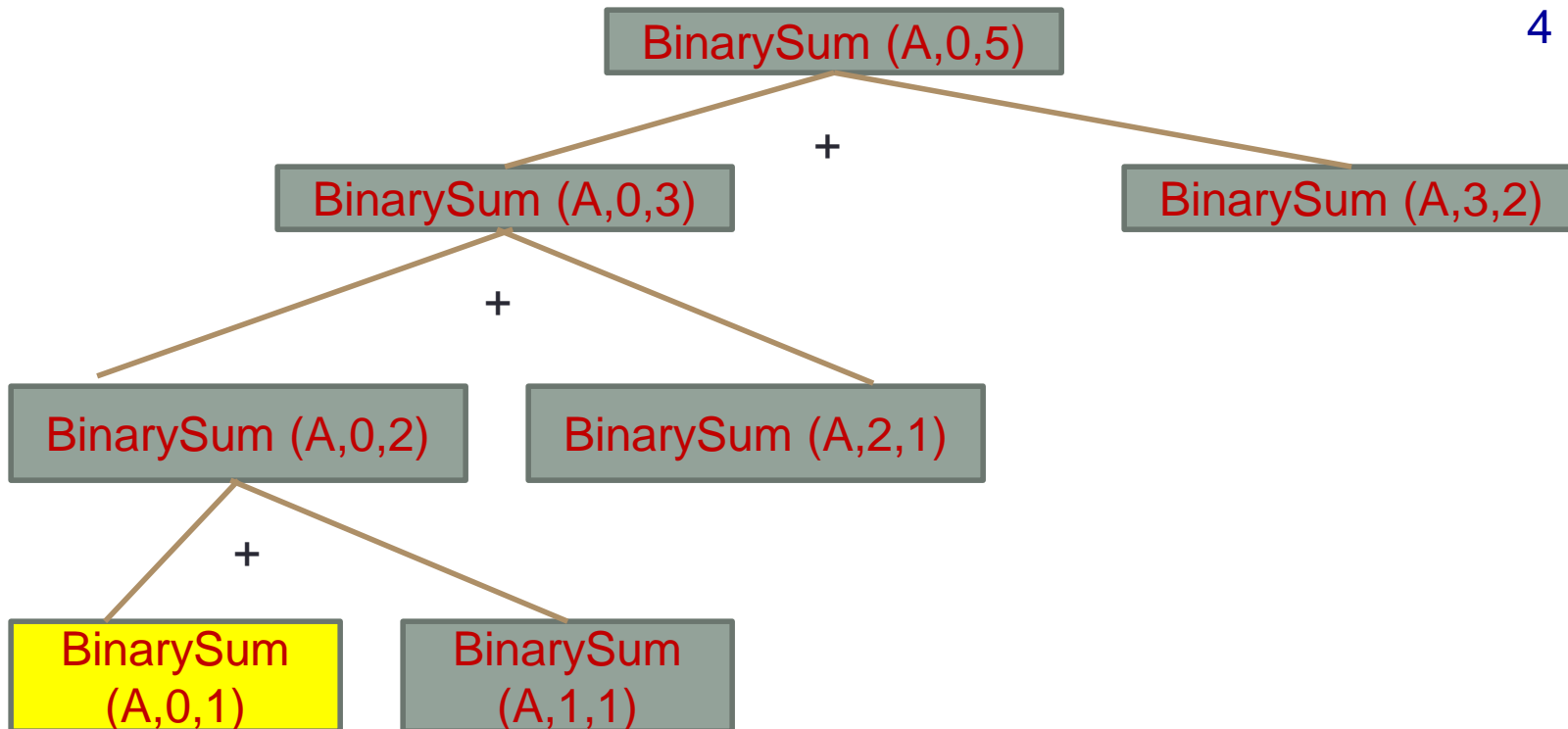


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

A	
0	6
1	5
2	3
3	2
4	8

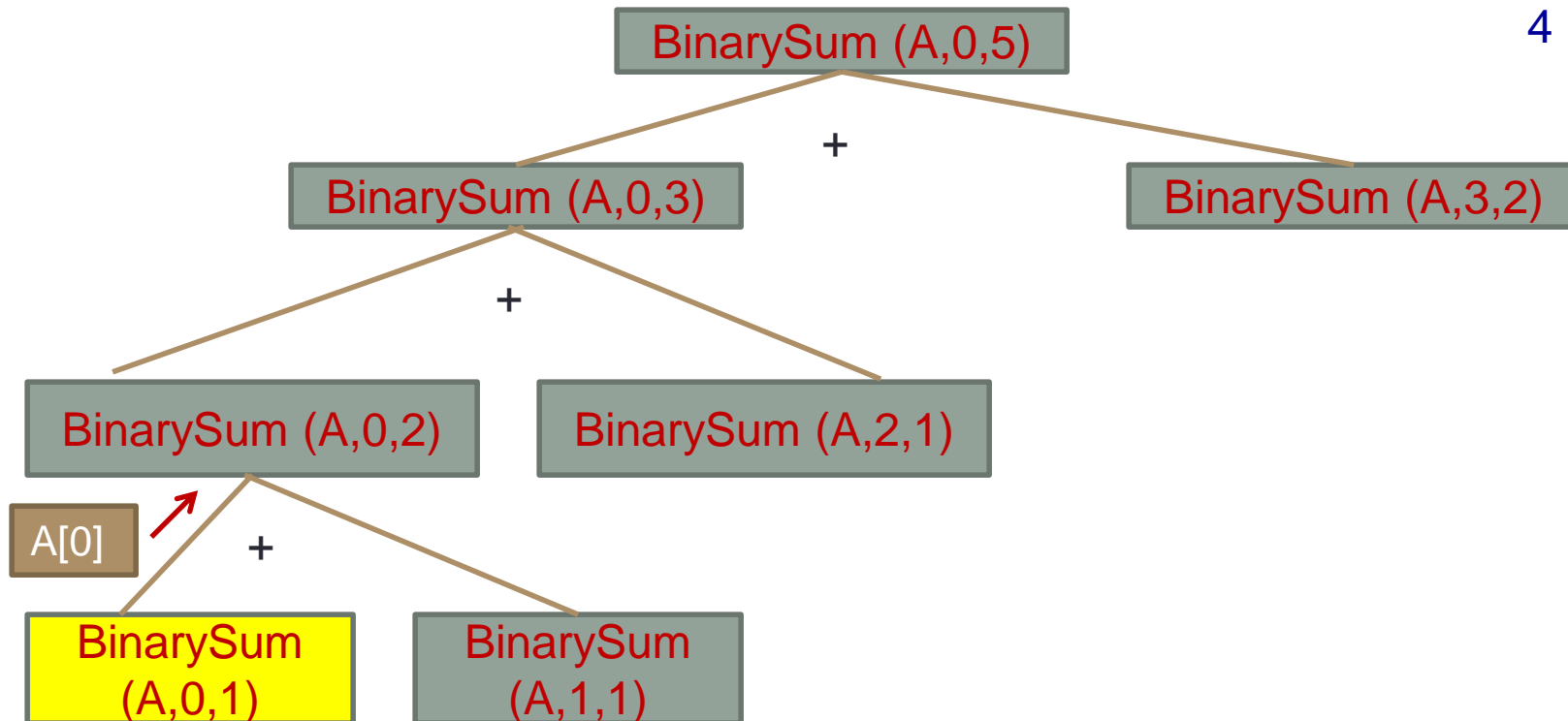


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

	A
0	6
1	5
2	3
3	2
4	8

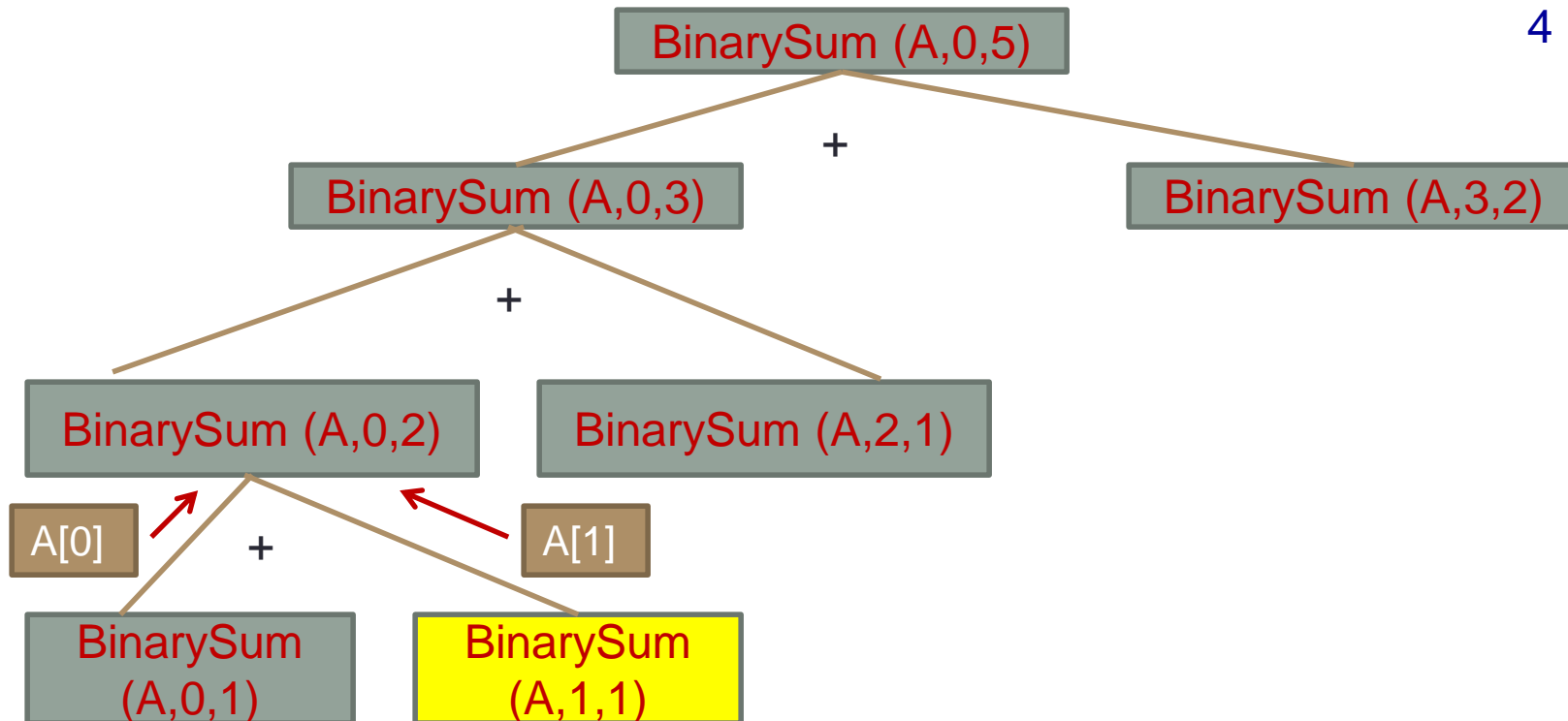


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

	A
0	6
1	5
2	3
3	2
4	8

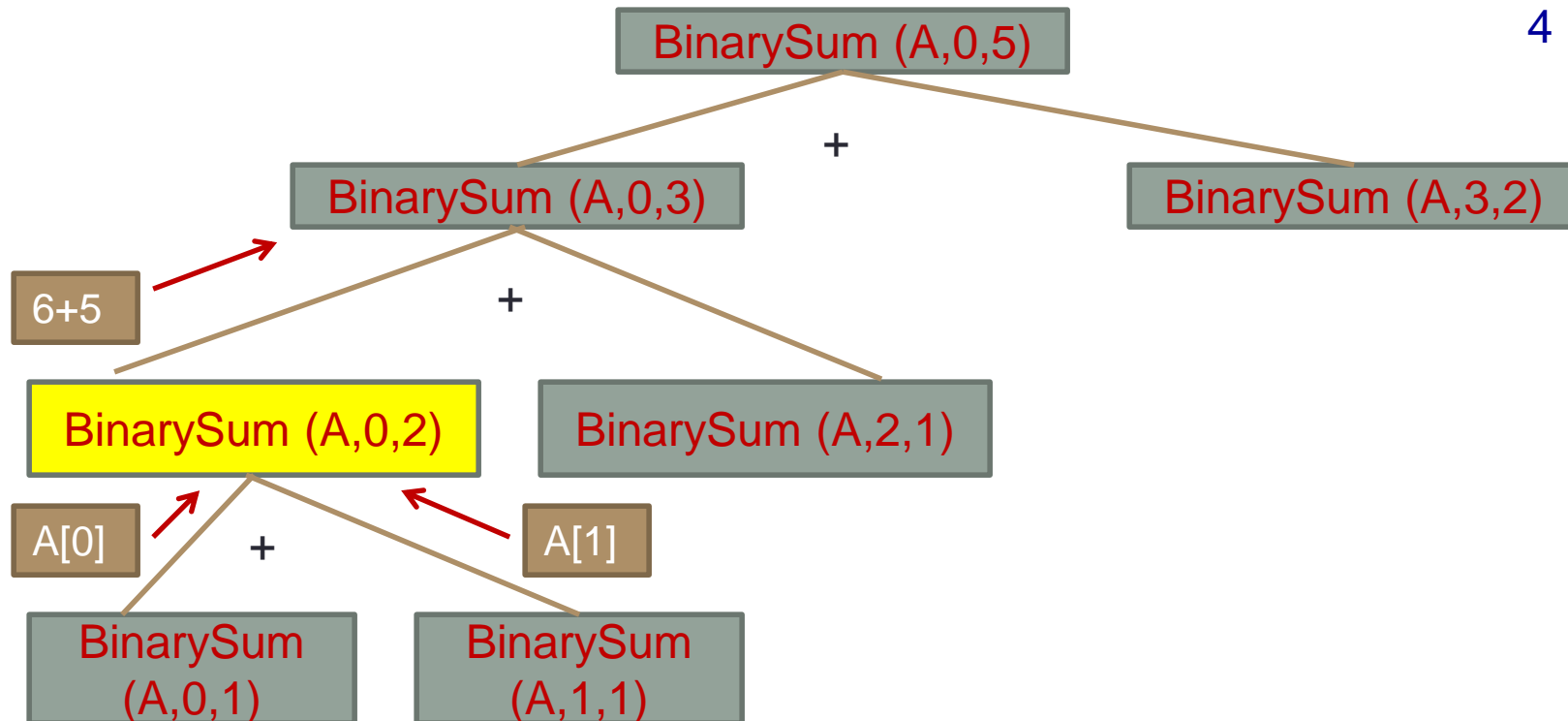


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

	A
0	6
1	5
2	3
3	2
4	8



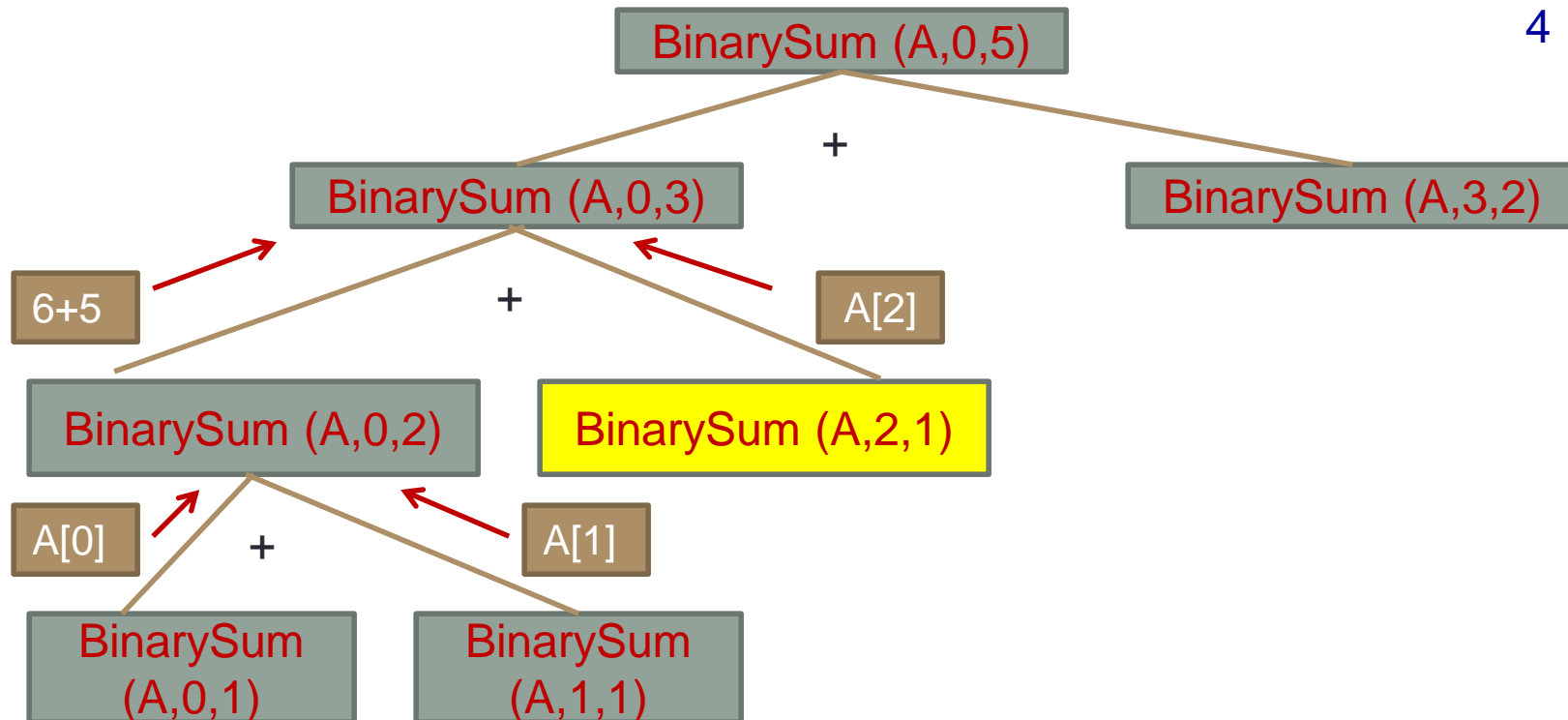


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

A	
0	6
1	5
2	3
3	2
4	8

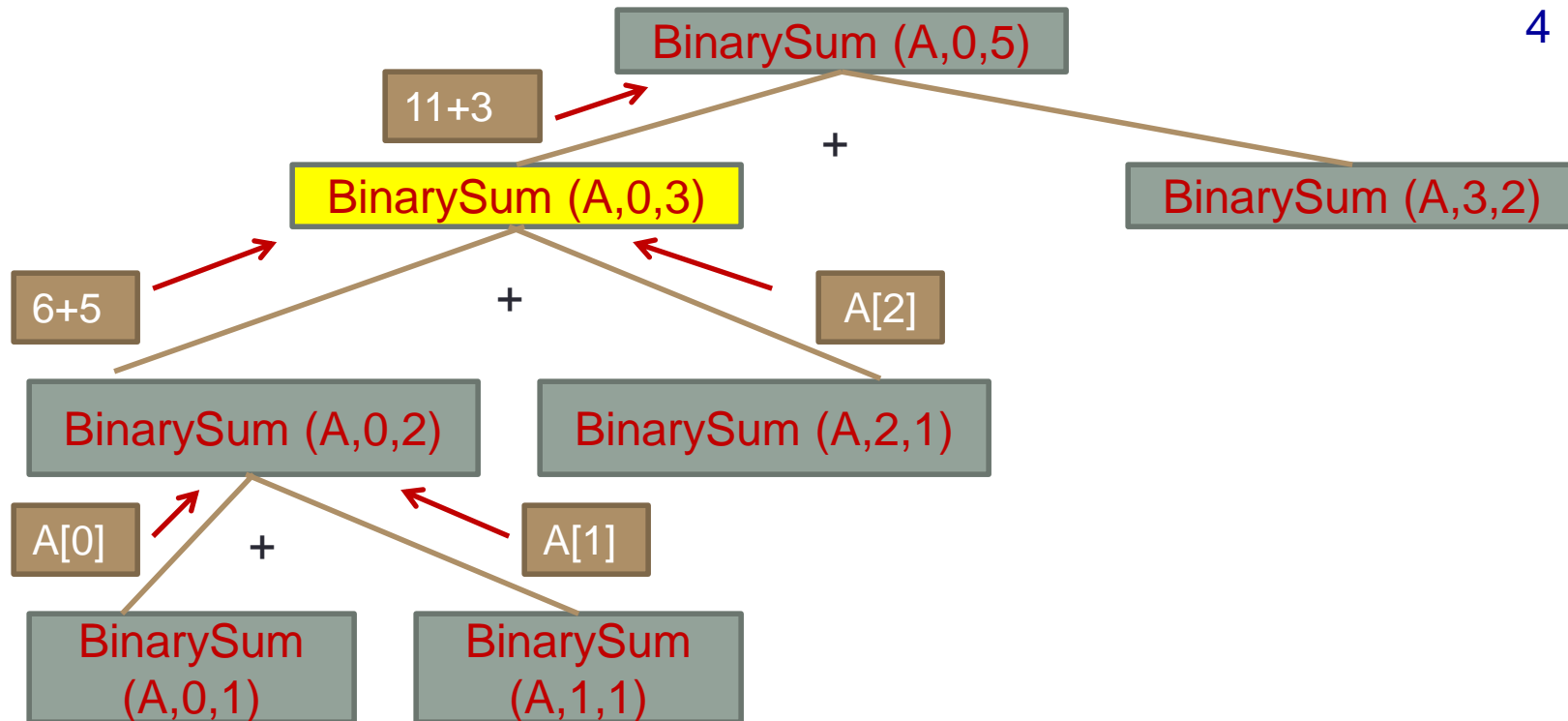


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

A	
0	6
1	5
2	3
3	2
4	8

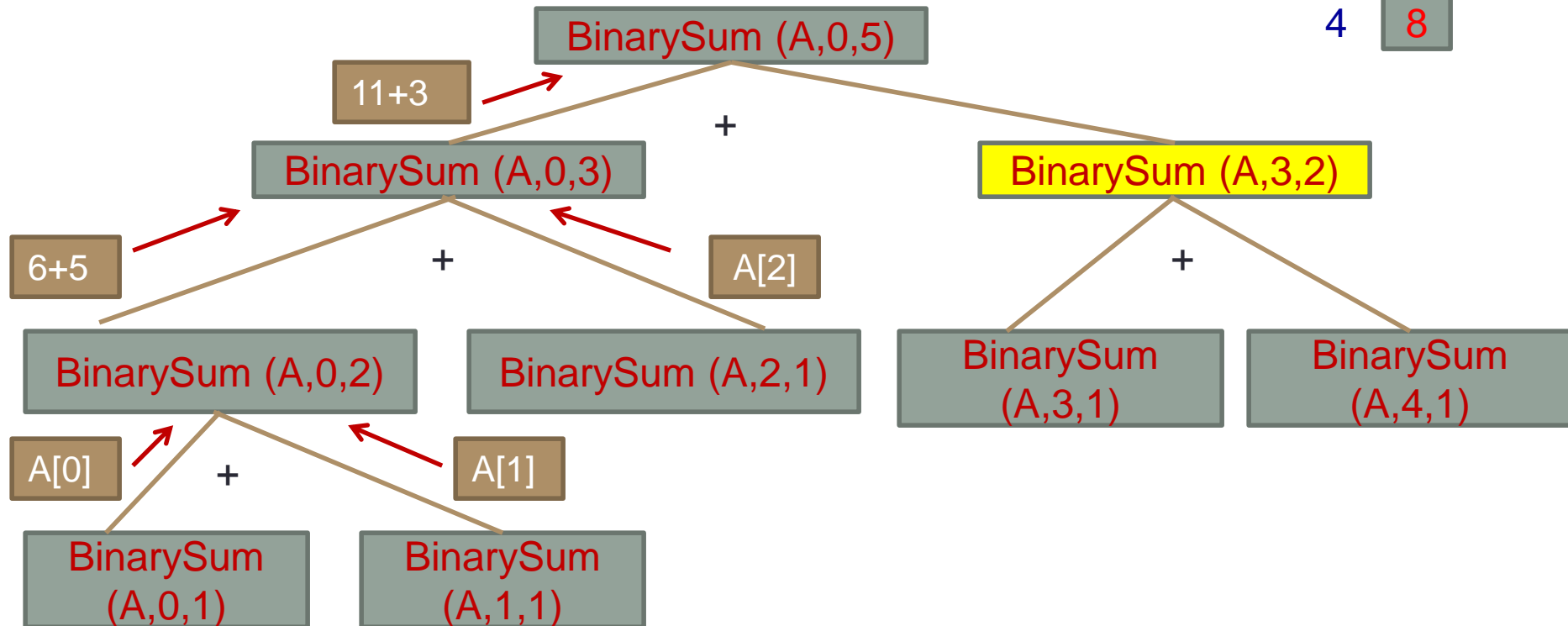


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

	A
0	6
1	5
2	3
3	2
4	8

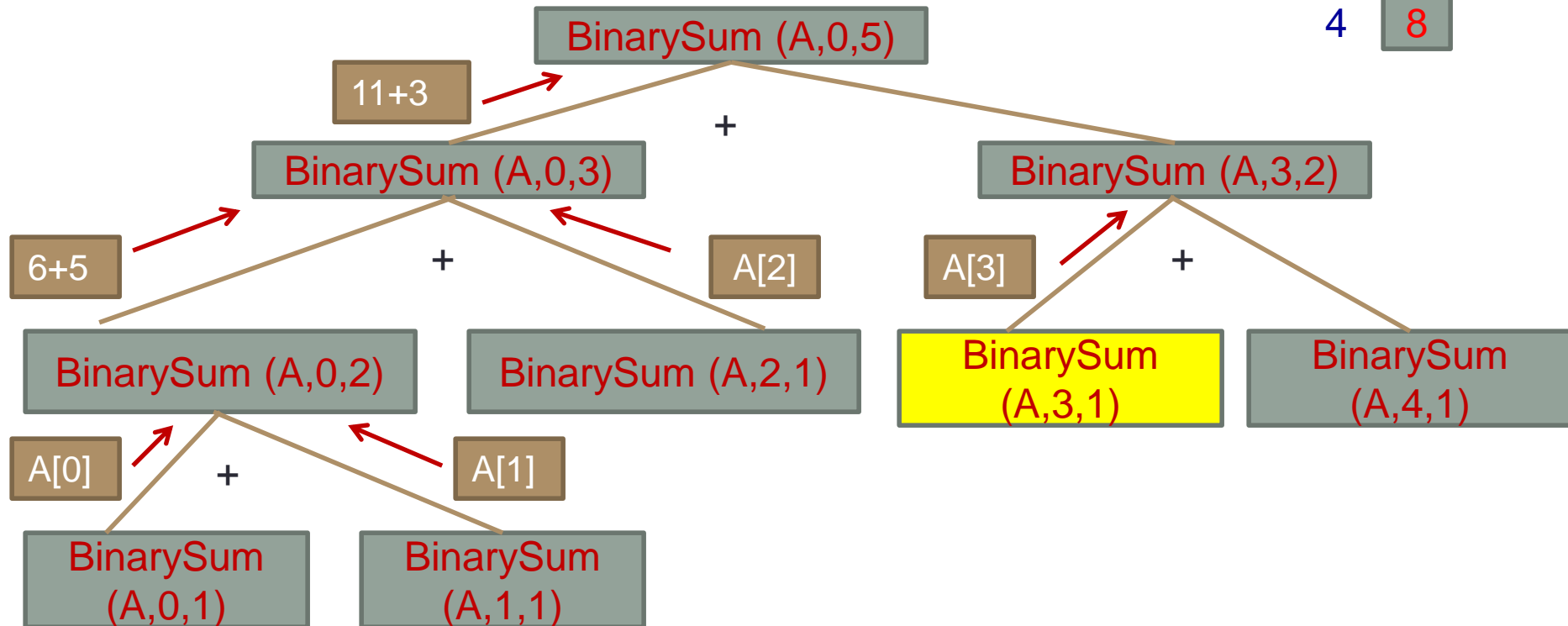


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

	A
0	6
1	5
2	3
3	2
4	8

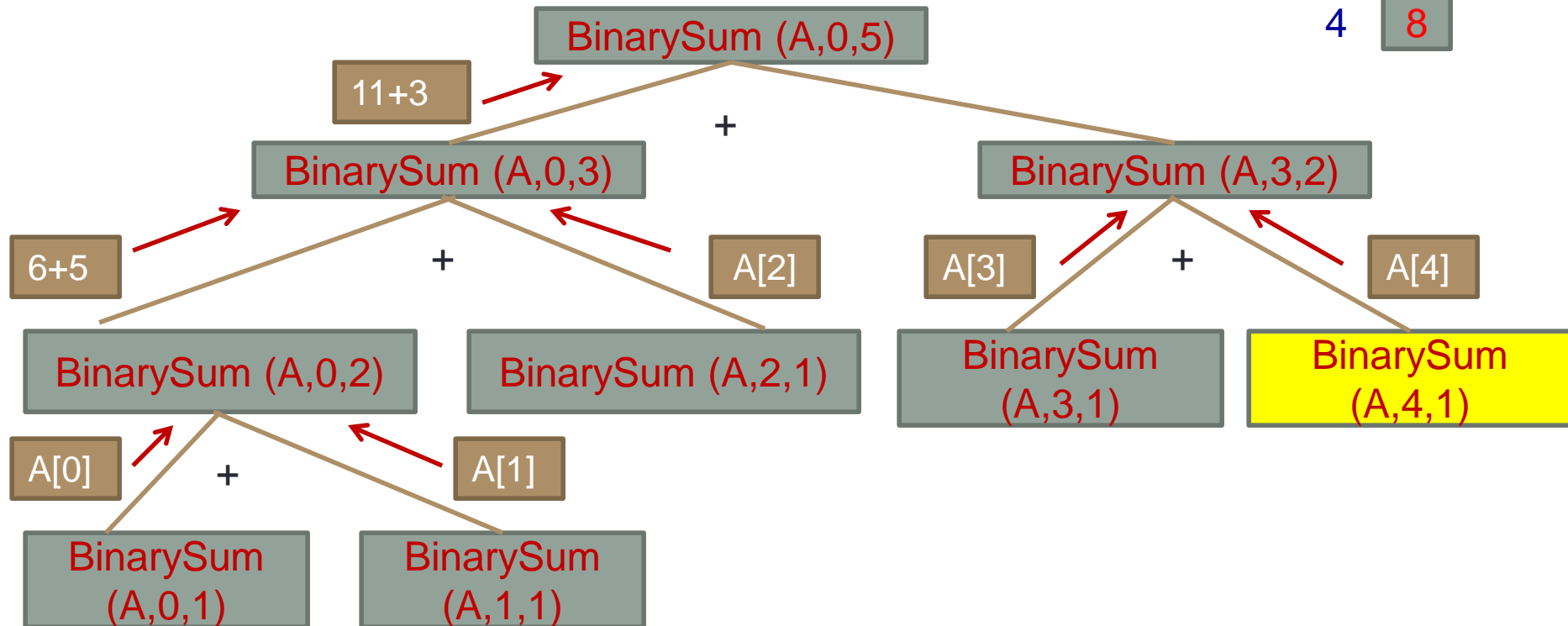


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

A	
0	6
1	5
2	3
3	2
4	8

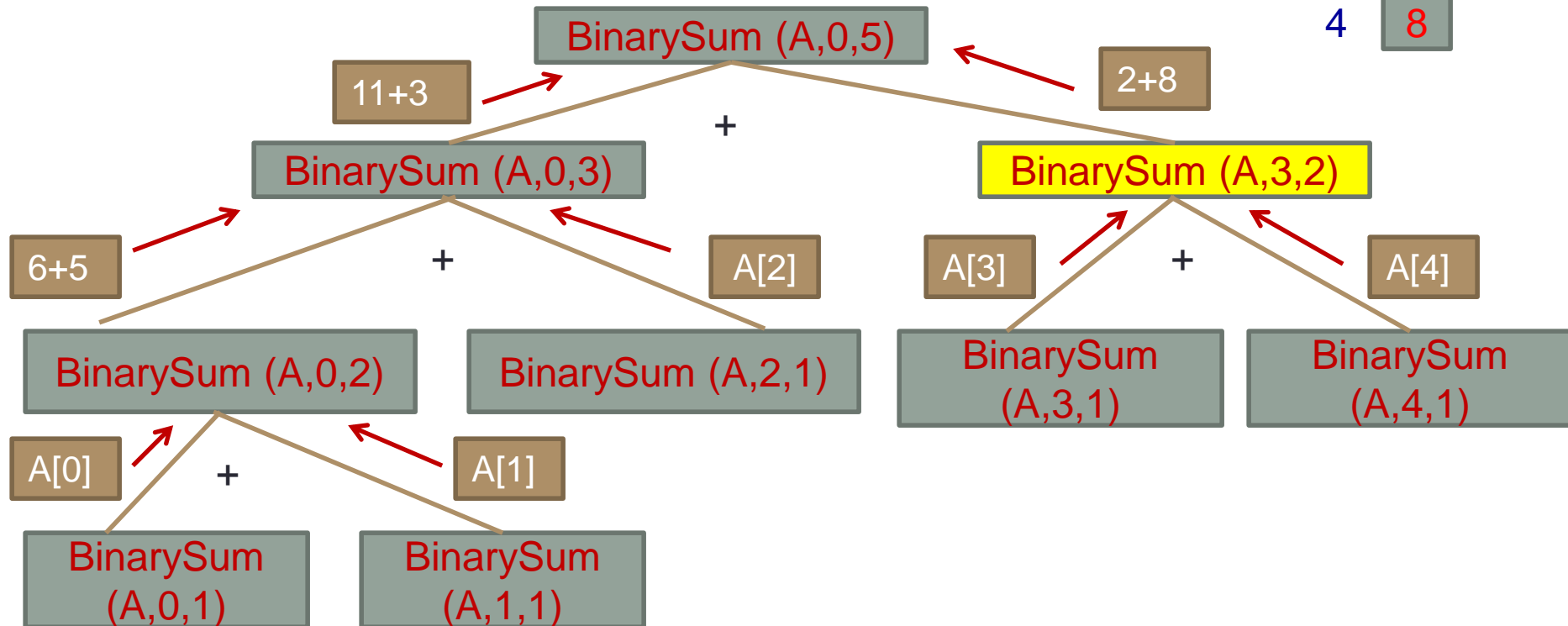


# Binary Recursion

## Example – 4 (Continued):

- Recursive trace

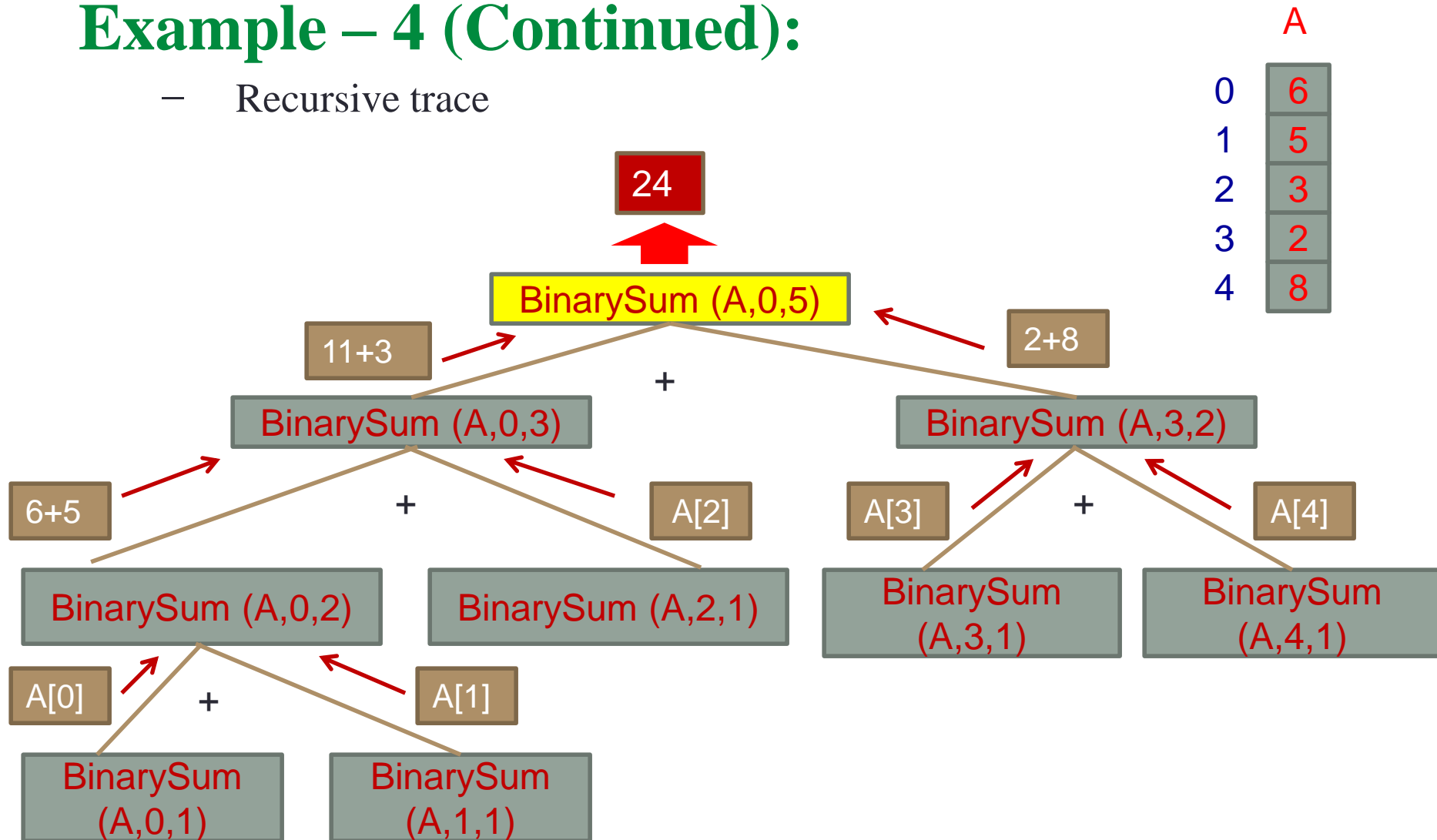
A	
0	6
1	5
2	3
3	2
4	8



# Binary Recursion

## Example – 4 (Continued):

- Recursive trace



# Binary Recursion

## Example – 5

- **The Fibonacci Number**  
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . . .
- Each number after the second number is the sum of the two preceding numbers.
- These numbers can naturally be defined recursively :

$$F(n) = \begin{cases} 1 & \text{if } n = 0 & \leftarrow \text{Base Case-1} \\ 1 & \text{if } n = 1 & \leftarrow \text{Base Case-2} \\ F(n-1) + F(n-2) & \text{if } n > 1 & \leftarrow \text{Recursive Case} \end{cases}$$



# Binary Recursion

## Example – 5 (Continued)

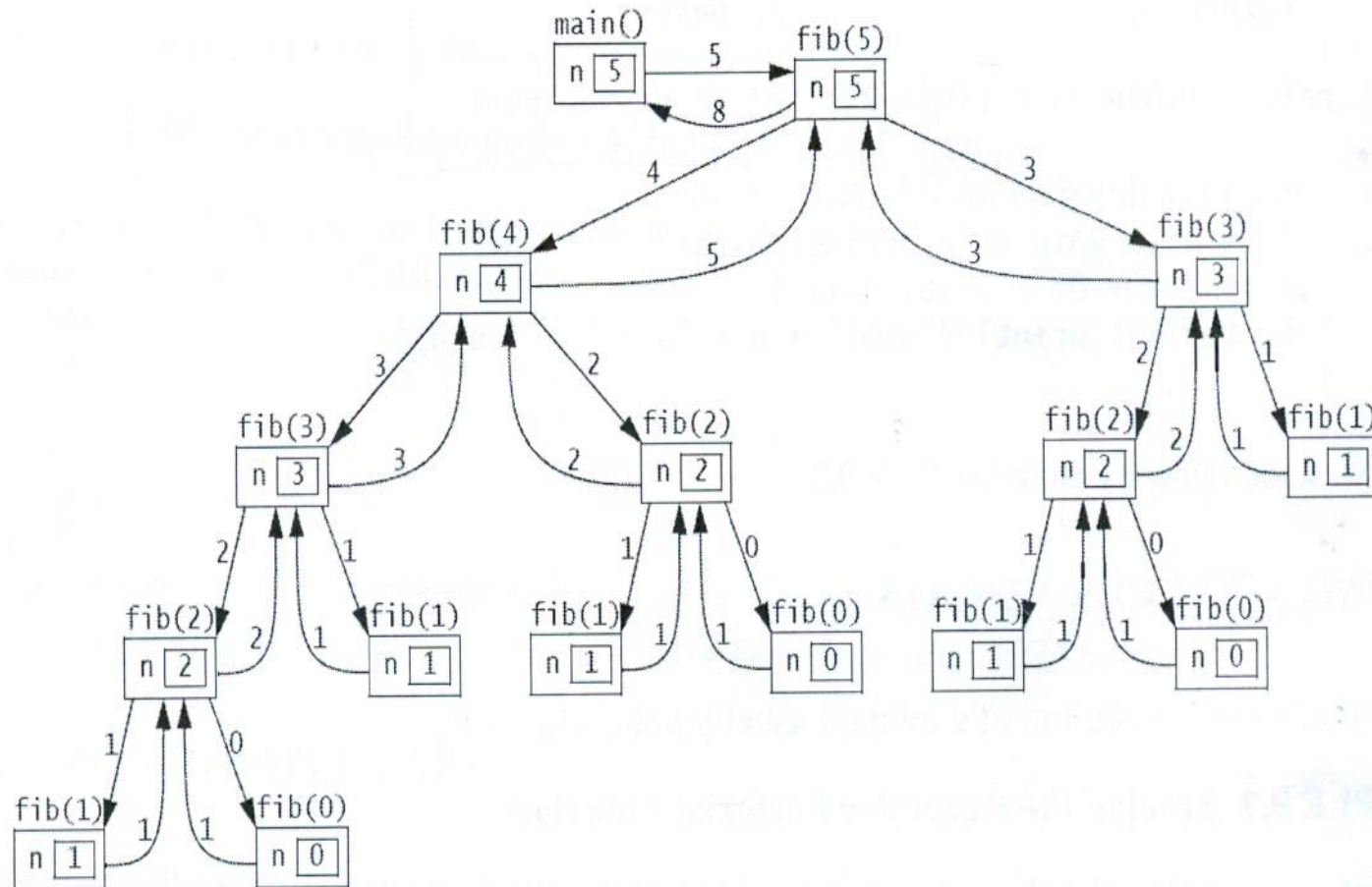
- Recursive Implementation of Fibonacci Function

```
public static int fib(int n)
{
    if (n < 2)
        return 1; // base cases
    else
        return fib(n-1)+fib(n-2); // recursive part
}
```

# Binary Recursion

## Example – 5 (Continued)

- Recursive Trace of Fibonacci Function: fib(5)



# Linear Recursion

## Example – 6: Binary Search

- **Problem:** Given  $S = \{s_0, s_1, \dots, s_{n-1}\}$  is a sorted sequence of  $n$  integers, and an integer  $x$ . Search whether  $x$  is in  $S$ .
- **Binary Search Algorithm:**
  - If the sequence is empty, return **-1**.
  - Let  $s_i$  be the middle element of the sequence.
    - If  $s_i = x$ , return its **index  $i$** .
    - If  $s_i < x$ , apply the algorithm on the subsequence that lies above  $s_i$ .
    - Otherwise, apply the algorithm on the subsequence of  **$S$**  that lies below  $s_i$ .

# Linear Recursion

## Example – 6 (continued): Binary Search

- Implementation:

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=2

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else { // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 14)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=4

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 14)

search(a, 3, 5, 14)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```



# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=2

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 2)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=0

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else { // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 2)

search(a, 0, 1, 2)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=2

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 5)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=0

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 5)

search(a, 0, 1, 5)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=0

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else { // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 5)

search(a, 0, 1, 5)

search(a, 1, 1, 5)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=2

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 21)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=4

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                    // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 21)

search(a, 3, 5, 21)



# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=5

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi/2);
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 21)

search(a, 3, 5, 21)

search(a, 5, 5, 21)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{ // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 21)

search(a, 3, 5, 21)

search(a, 5, 5, 21)

search(a, 6, 5, 21)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=2

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 12)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=4

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 12)

search(a, 3, 5, 12)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

i=3

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{                  // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 12)

search(a, 3, 5, 12)

search(a, 3, 3, 12)

# Linear Recursion

## Example – 6 (continued): Binary Search

– Implementation:

a	2	5	7	11	14	20
---	---	---	---	----	----	----

```
public static int search(int a[], int lo, int hi, int x)
{
    if (lo > hi) return -1; // Basis
    else{ // Recursive part
        int i = (lo+hi)/2;
        if(a[i] == x) return i;
        else if(a[i] < x)
            return search (a, i+1, hi, x)
        else
            return search (a, lo,i-1, x);
    }
}
```

search(a, 0, 5, 12)

search(a, 3, 5, 12)

search(a, 3, 3, 12)

**search(a, 4, 3, 12)**