Merge Sort / Recursion

Lecture Question

Task: In a package name fp create a Scala object named ReturnFunction with a method named strangeAddition which

- Takes an Int as a parameter
- Return a function that takes an Int as a parameter and returns an Int
- The returned function returns the addition of the two provided Ints
- Example:
  - **val adder = ReturnFunction.strangeAddition(5)**
  - This returns a function that adds 5 to its input
  - assert(**adder(3) == 8**)

Runtime Analysis

- Last time we said Selection sort is inefficient
- Let's be more specific
- We'll measure the asymptotic runtime of the algorithm
    - Often use big-O notation
- Count the number of "steps" the algorithm take
    - A step is typically a basic operation (+, -, &&, etc)

Runtime Analysis

- Asymptotic runtime
  - Measures the order of magnitude of the runtime in relation to the size of the input
    - Name the input size **n**
    - For sorting - Size of the input is the number of values in the data structure
    - Ignore constants
- Ex. Runtime of O(n) grows linearly with the size of the input

Selection Sort - Runtime

- Abridged runtime analysis

Outer loop runs once for each index

Runs O(n) times

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort - Runtime

- Abridged runtime analysis

Inner loop runs once for each index from i to the end of the list

Runs for each iteration of the outer loop with a worst case of O(n)

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort - Runtime

- Abridged runtime analysis

Run O(n) iterations O(n) times results in an $O(n^2)$ total runtime

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort - Runtime

- Abridged runtime analysis

We reach $O(n^3)$ since apply takes O(n)

More details next week

```
def selectionSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  var data: List[T] = inputData
  for (i <- data.indices) {
    var minFound = data.apply(i)
    var minIndex = i
    for (j <- i until data.size) {
      val currentValue = data.apply(j)
      if (comparator(currentValue, minFound)) {
        minFound = currentValue
        minIndex = j
      }
    }
    data = data.updated(minIndex, data.apply(i))
    data = data.updated(i, minFound)
  }
  data
}
```

Selection Sort - Runtime

- More mathematical analysis
  - Inner loop runs Σ i times where i ranges from n to 1
  - $n + n-1 + n-2 + ... + 2 + 1 = n^2/2 + n/2$
  - For asymptotic we only consider the highest order term and ignore constant multipliers
  - Therefore $n^2/2 + n/2$ is $O(n^2)$
  - Selection Sort has $O(n^2)$ runtime

Merge Sort

- We briefly saw in CSE115 that we can do better by using merge sort and reaching O(n log(n)) runtime
- Let's analyze this in more depth

Merge Sort

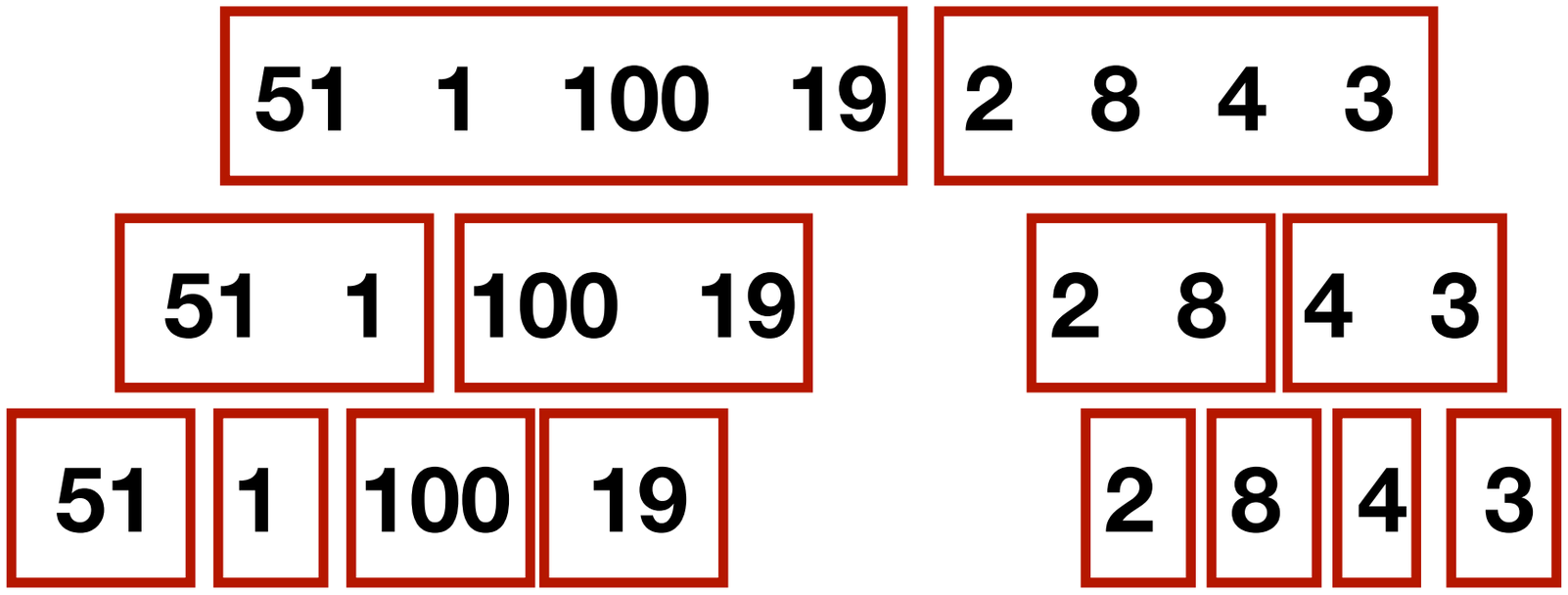
- The algorithm
  - If the input list has < 2 elements
    - Return it (It's already sorted)
  - Else
    - Divide the input list in two halves
    - Recursively call merge sort on each half (Repeats until the lists are size 1)
    - Merge the two sorted lists together into a single sorted list

Merge Sort

- Given an input

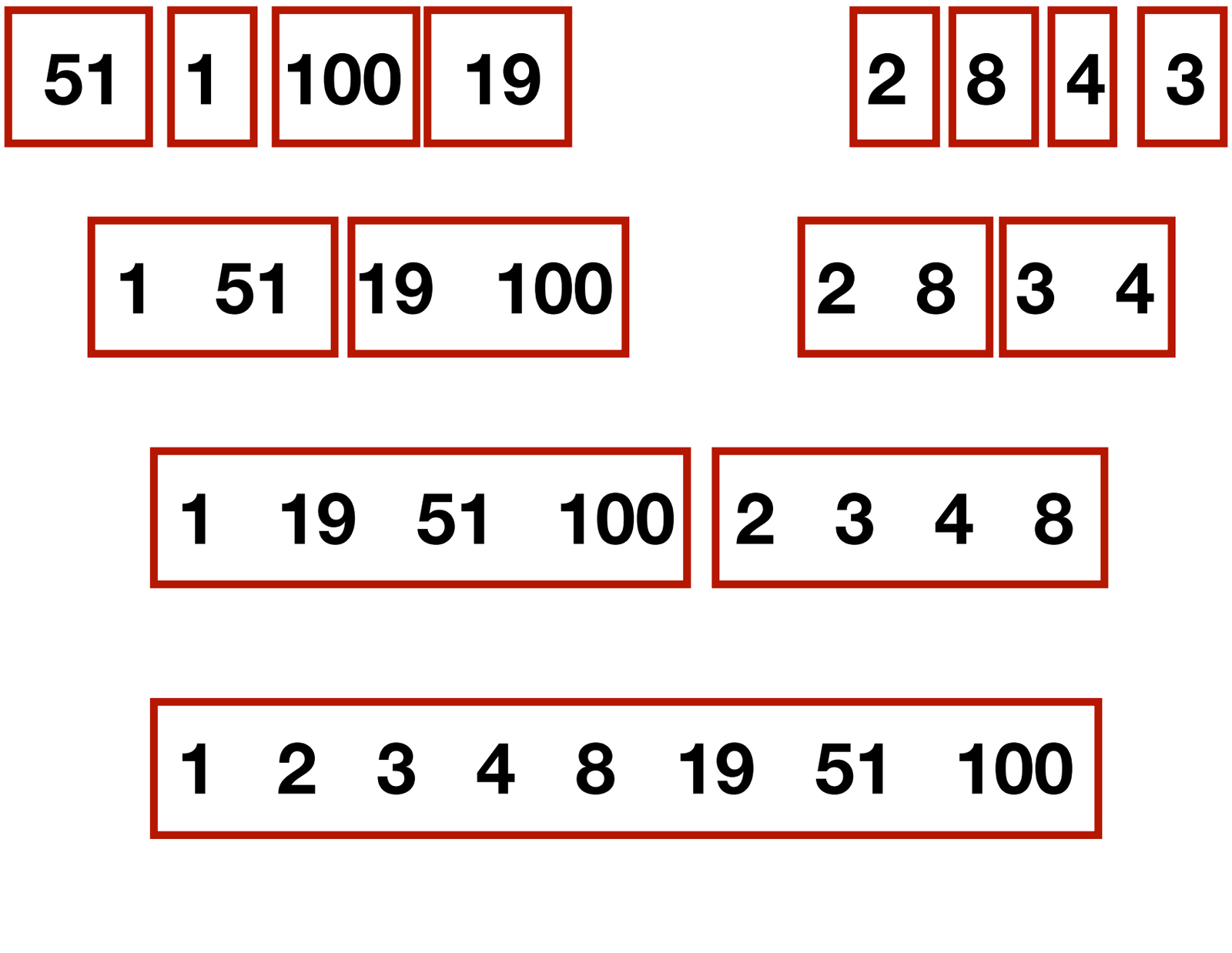
51 1 100 19 2 8 4 3

- Divide into two lists recursively until n=1



Merge Sort

- Merge lists until the original input is sorted



Merge Sort - Merge

```
def mergeSort[T](inputData: List[T], comparator: (T, T) => Boolean): List[T] = {
  if (inputData.length < 2) {
    inputData
  } else {
    val mid: Int = inputData.length / 2
    val (left, right) = inputData.splitAt(mid)
    val leftSorted = mergeSort(left, comparator)
    val rightSorted = mergeSort(right, comparator)

    merge(leftSorted, rightSorted, comparator)
  }
}
```

Recursion!

Merge Sort - Runtime

- Each level of the recursion has 2^i lists of size $n/2^i$
- Recursion ends when is $n/2^i == 1$
    - i = log(n)
    - log(n) levels of recursion
- Each level needs to merge a total of n elements across all sub-lists
- If we can merge in O(n) time we'll have O(n log(n)) total runtime

Merge Sort - Merge

- Merge two sorted lists in O(n) time
- Take advantage of each list being sorted
- Start with pointers at the beginning of each list
- Compare the two values at the pointers and find which come first based on the comparator
    - Append it to a new list and advance that pointer
- When a pointer reaches the end of a list copy the rest of the contents

Merge Sort - Merge

1 19 51 100

2 3 4 8

Merge Sort - Merge

1 19 51 100

2 3 4 8

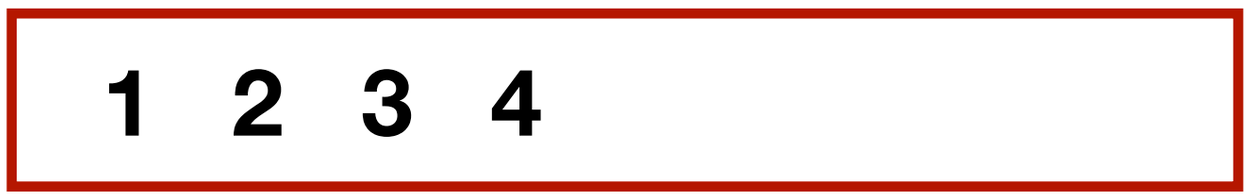
Merge Sort - Merge

1 19 51 100

2 3 4 8

Merge Sort - Merge

Merge Sort - Merge

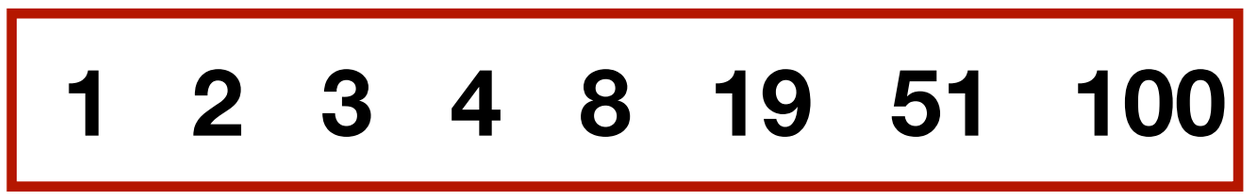


Merge Sort - Merge

When a pointer reaches the end of a list, copy the rest of the other list to the result

Merge Sort - Merge

When a pointer reaches the end of a list, copy the rest of the other list to the result



Merge Sort - Merge

```
def merge[T](left: List[T], right: List[T], comparator: (T, T) => Boolean): List[T] = {
  var leftPointer = 0
  var rightPointer = 0

  var sortedList: List[T] = List()

  while (leftPointer < left.length && rightPointer < right.length) {
    if (comparator(left.apply(leftPointer), right.apply(rightPointer))) {
      sortedList = sortedList :+ left.apply(leftPointer)
      leftPointer += 1
    } else {
      sortedList = sortedList :+ right.apply(rightPointer)
      rightPointer += 1
    }
  }

  while (leftPointer < left.length) {
    sortedList = sortedList :+ left.apply(leftPointer)
    leftPointer += 1
  }
  while (rightPointer < right.length) {
    sortedList = sortedList :+ right.apply(rightPointer)
    rightPointer += 1
  }

  sortedList
}
```

Use the comparator to make ordering decisions

One Last Sorting Example

Custom Sorting

- We can sort any type with any comparator
- But what if we want to sort points by their distance from a reference point
    - In general: what if the comparator needs more parameters than just the two elements?
    - [Without using global state]
- We can dynamically create a new function with the additional parameters "built-in"

Returning Functions

- We can write a function/method that takes all the needed parameters and returns a function that fits the signature of a comparator
- The distanceComparator method returns a comparator that compares the distance to a reference point

```
def distance(v1: PhysicsVector, v2: PhysicsVector): Double = {
  Math.sqrt(Math.pow(v1.x - v2.x, 2.0) + Math.pow(v1.y - v2.y, 2.0) + Math.pow(v1.z - v2.z, 2.0))
}

def distanceComparator(referencePoint: PhysicsVector): (PhysicsVector, PhysicsVector) => Boolean = {
  (v1: PhysicsVector, v2: PhysicsVector) => {
    distance(v1, referencePoint) < distance(v2, referencePoint)
  }
}
```

Returning Functions

- Use distanceComparator to create a comparator function when needed
- Can create different comparators with different reference points
  - Global state would only allow one comparator at a time

```
val referencePoint = new PhysicsVector(0.5, 0.5, 0.0)
val sortedPoints = MergeSort.mergeSort(points, distanceComparator(referencePoint))
```

```
def distance(v1: PhysicsVector, v2: PhysicsVector): Double = {
  Math.sqrt(Math.pow(v1.x - v2.x, 2.0) + Math.pow(v1.y - v2.y, 2.0) + Math.pow(v1.z - v2.z, 2.0))
}

def distanceComparator(referencePoint: PhysicsVector): (PhysicsVector, PhysicsVector) => Boolean = {
  (v1: PhysicsVector, v2: PhysicsVector) => {
    distance(v1, referencePoint) < distance(v2, referencePoint)
  }
}
```

Recursion

Recursion Example

```
def computeGeometricSum(n: Int): Int ={
 if(n>0) {
    var result: Int = computeGeometricSum(n - 1)
    result += n
    result
  }else{
    0
  }
}

def main(args: Array[String]): Unit = {
  val result: Int = computeGeometricSum(3)
  println(result)
}
```

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Each recursive calls creates a new stack frame
- Each frame remembers where it will resume running when its on the top of the stack

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	pointer -> line 3

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Each time the method is called a new frame is created
- New frame starts at the first line of the method

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	pointer -> line 3
Method Frame	name:n, value:0
	pointer -> line 1

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Top frame on the stack executes the method one line at a time

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	pointer -> line 3
Method Frame	name:n, value:0
	pointer -> line 2

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Top frame on the stack executes the method one line at a time

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	pointer -> line 3
Method Frame	name:n, value:0
	pointer -> line 7

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	pointer -> line 3
Method Frame	Returning 0

- When a method call returns, its frame is destroyed
- Calling frames resumes and uses the return value

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- When a method call returns, its frame is destroyed
- Calling frames resumes and uses the return value

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	name:result, value:0
	pointer -> line 3

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Method continues after the recursive call

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	name:n, value:1
	name:result, value:1
	pointer -> line 3

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Method continues after the recursive call

Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	pointer -> line 3
Method Frame	returning 1

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

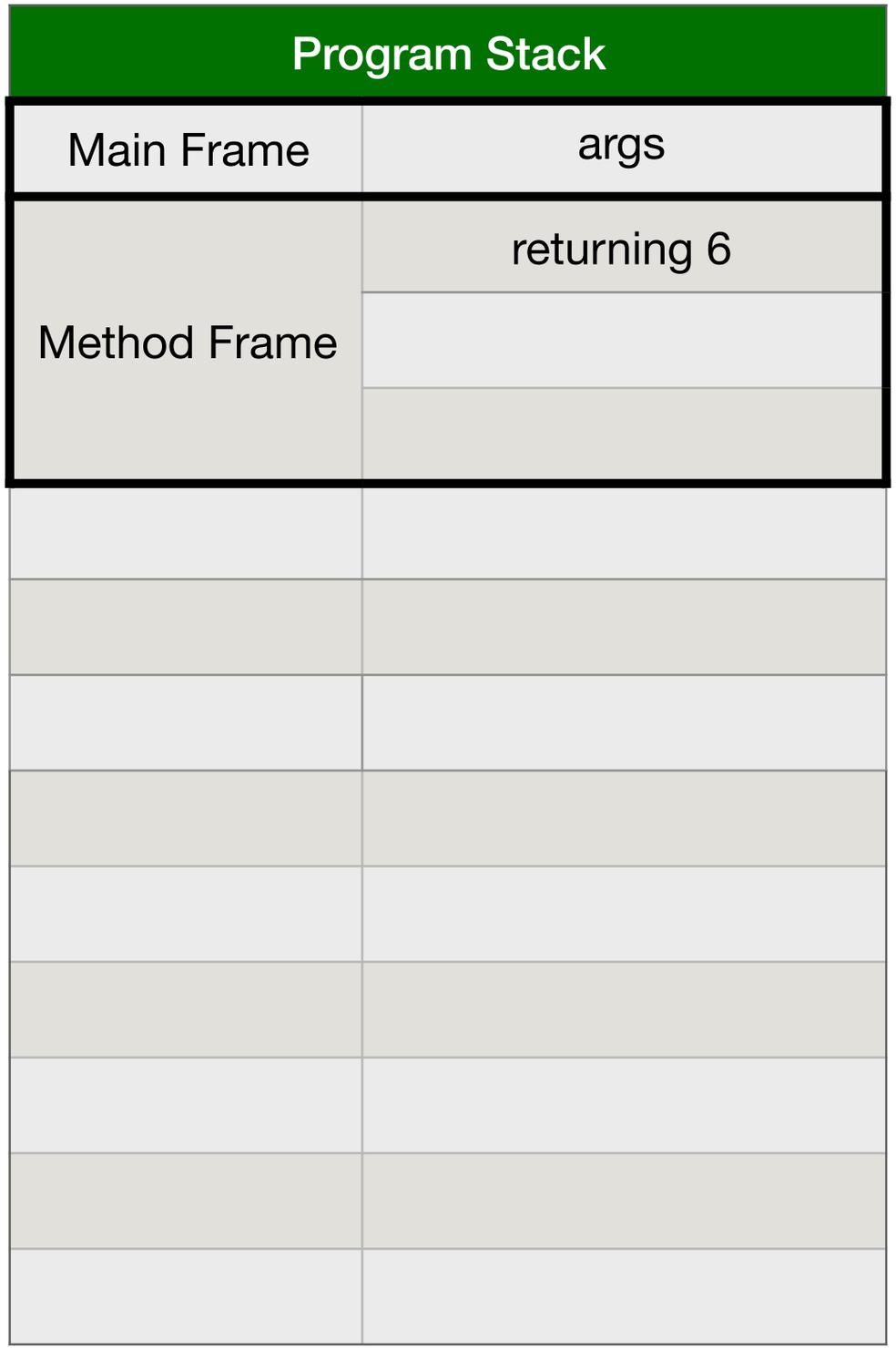
Program Stack	
Main Frame	args
Method Frame	name:n, value:3
	pointer -> line 3
Method Frame	name:n, value:2
	name:result, value:1
	pointer -> line 3

- Return value
- Pop off the stack
- Resume execution of the top frame

Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

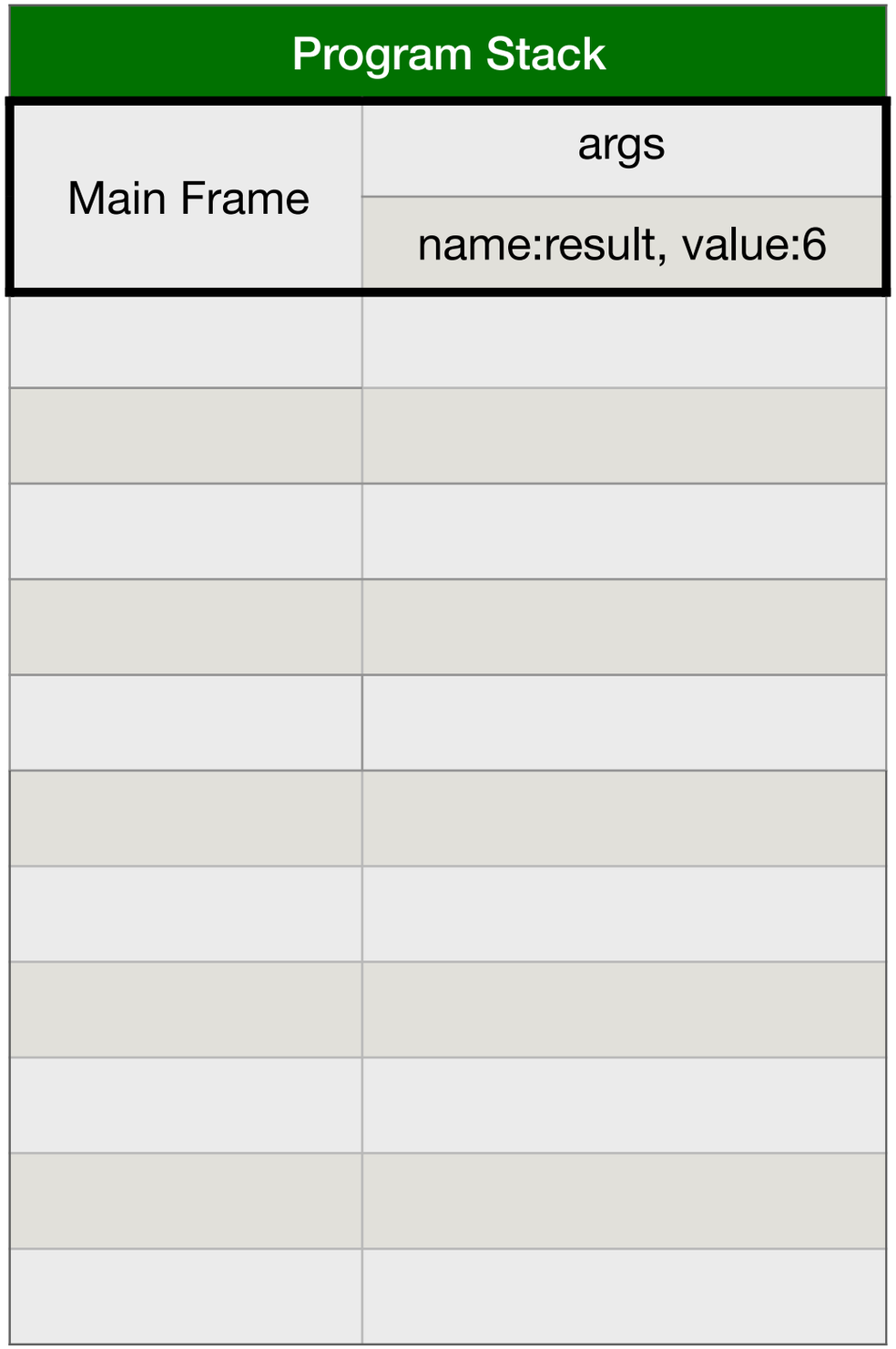
- Process continues until all recursive calls resolve
- Last frame returns to main



Recursive Example

```
1:  def computeGeometricSum(n: Int): Int ={
2:    if(n>0) {
3:      var result: Int = computeGeometricSum(n - 1)
4:      result += n
5:      result
6:    }else{
7:      0
8:    }
9:  }
10:
11: def main(args: Array[String]): Unit = {
12:   val result: Int = computeGeometricSum(3)
13:   println(result)
14: }
```

- Main continues with the result from the recursive calls



Lecture Question

Task: In a package name fp create a Scala object named ReturnFunction with a method named strangeAddition which

- Takes an Int as a parameter
- Return a function that takes an Int as a parameter and returns an Int
- The returned function returns the addition of the two provided Ints
- Example:
  - **val adder = ReturnFunction.strangeAddition(5)**
  - This returns a function that adds 5 to its input
  - assert(**adder(3) == 8**)