

Recursion

Lecture Question

- Genetic Algorithm: Lecture Task 3 -

Functionality: In the `statistics.Statistics` object, implement the following method:

- A method named “`bayesianAverage`” with:
 - A type Parameter `T`
 - Has parameters of type `List of T`, a function of type `T to Double`, an `Int` representing the number of extra “fake” ratings, and an `Int` representing the value of the extra ratings
 - Returns the bayesian average of the elements in the list after the input function is applied to them

Testing: In the `tests` package, complete the test suite named `LectureTask3` that tests this functionality.

Recursion Example

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

- Computes the geometric sum of the input
 - ex: if $n == 3$, geometric sum is $3+2+1 == 6$

Recursion Example

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

- Base Case:
 - An input with a trivial output
 - Geometric sum of 0 is defined as 0
 - We could also add 1 -> 1 as a base case

Recursion Example

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

- Recursive Step:
 - Any input that is not a base case will put another recursive call on the stack
 - Write the recursive step with the assumption that the recursive call will return the correct value

Recursion Example

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

- Recursive calls must get closer to the base case
 - All calls must eventually reach a base case or we'll go infinite
 - $n-1$ is closer to $n \leq 0$ than n
 - Regardless of the original value of n , it will eventually be decremented until the base case condition is true


Recursive Example

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

- Each recursive call creates a new stack frame
- Each frame remembers where it will resume running when it's on the top of the stack

Program Stack	
Main Frame	args
	pointer -> line 11
Method Frame	name:n, value:3
	pointer -> line 5
Method Frame	name:n, value:2
	pointer -> line 5
Method Frame	name:n, value:1
	pointer -> line 5

Recursive Example



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

- New frames start at the first line of the method
- Top frame on the stack executes the method one line at a time

[illegible]

Recursive Example

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

- When a method call returns, its frame is destroyed
- The calling frame resumes and uses the returned value

[illegible]

Recursive Example

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

- Method continues after the recursive call
- Sums $n + \text{return value}$ and returns this value

[illegible]

Recursive Example

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

- This frame reaches the end of the method
- Repeat the process

[illegible]

Recursive Example

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

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

[illegible]

Recursive Example

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

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

[illegible]

Recursive Example

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

- Main continues with the result from the recursive calls

[illegible]

Recursive Example

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

- Main continues with the result from the recursive calls

[illegible]

Anagrams Revisited

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Recall anagrams
 - Rewritten to use functional programming and no vars
 - The syntax may not fully make sense until the next Functional Programming lecture

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Base Case
 - A String of length 1 is itself its only anagram
 - If the length is 1, return a new list containing only that String

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Base Case Note
 - We will eventually return a list containing all anagrams from the top level call
 - The base case is the only time we create a new List

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Recursive Step
 - For each character in the input String
 - Remove that character and make a recursive call with the remaining characters
 - Append the removed character to all the returned anagrams

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Recursive Step
 - We write this code with the assumption that our recursive calls will return all the anagrams of the new Strings
 - If our logic is sound, this assumption will be true through the power of recursion!

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Always reach a base case
 - We always make recursive calls on the input String with 1 character removed
 - `newString.length == input.length - 1`
 - This always gets us closer to the base case

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
    output.flatten.distinct  
  }  
}
```

- Always reach a base case
 - When the base case is reached and returned, our logic starts working for us
 - If this code does append the removed character to each returned anagram, output is generated starting at the base case and built up as the stack frames return

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
    output.flatten.distinct  
  }  
}
```

- Example:
 - input == "at"
 - Makes 2 recursive calls to the base case
 - "a" and "t" are returned
 - Append "t" to "a" and "a" to "t" (The removed characters)
 - Return ["at", "ta"] to the next recursive call with an input of length 3

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Functional Programming notes (More detail Later)
 - yield: Creates a data structure containing the last expression that was evaluated on each iteration of a loop

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Functional Programming notes (More detail Monday)
- map: Creates a new data structure by applying a function to each element
- The `_` is shorthand syntax we can use instead of naming the parameters of a function when the types can be inferred, and each input is only used once

Anagrams Example

```
def anagrams(input: String): List[String] = {  
  if (input.length == 1) {  
    List(input)  
  } else {  
    val output: List[List[String]] = (for (i <- 0 until input.length) yield {  
      val newString: String = input.substring(0, i) + input.substring(i + 1, input.length)  
      anagrams(newString).map(_ + input.charAt(i))  
    }).toList  
  
    output.flatten.distinct  
  }  
}
```

- Functional Programming notes (More detail Monday)
 - Scala data structures come with many helpful FP style methods
 - Flatten: Creates a single List from a List of Lists containing all the elements from each List
 - Distinct: Creates a new List with all duplicate values removed

Lecture Question

- Genetic Algorithm: Lecture Task 3 -

Functionality: In the `statistics.Statistics` object, implement the following method:

- A method named “`bayesianAverage`” with:
 - A type Parameter `T`
 - Has parameters of type `List of T`, a function of type `T to Double`, an `Int` representing the number of extra “fake” ratings, and an `Int` representing the value of the extra ratings
 - Returns the bayesian average of the elements in the list after the input function is applied to them

Testing: In the `tests` package, complete the test suite named `LectureTask3` that tests this functionality.