Recursion
Lecture Question

Restriction: No state is allowed in this question. Specifically, the keyword "var" is banned. (ie. You are expected to use a recursive solution)

Method: In a package named "functions" create an object named "Algebra" with a method named "factor" that takes an Int as a parameters and returns the prime factorization of that parameter as a List of Ints.

The following apply to this method:
• If the input is negative or 0, return an empty list
• If the input is 1, return an empty list (Note: This is different than the original question)
• Do not include 1 in the output for any inputs
• The order of the factors in the output List is undefined

Example: functions.Algebra.factor(12) can return List(2,2,3) -or- List(2,3,2) -or- List(3,2,2)

Unit Testing: Testing will be checked by AutoLab, though you are encouraged to use your tests from the last time we had this question (Be sure to update your test for 1 as the input)
Recursion Example

```scala
def computeGeometricSum(n: Int): Int = {
  if (n <= 0) {
    0
  } else {
    n + computeGeometricSum(n - 1)
  }
}

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

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

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

- **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
• 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

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

- 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<=0 than n
- Regardless of the original value of n, it will eventually be decremented until the base case condition is true
Recursive Example

```python
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 calls creates a new stack frame
- Each frame remembers where it will resume running when it's on the top of the stack
Recursive Example

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

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

• Recursive calls are added to the stack until a base case is reached
Recursive Example

```scala
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
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
Recursive Example

```scala
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
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

<table>
<thead>
<tr>
<th>Program Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Frame</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Method Frame</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Method Frame</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Recursive Example

```scala
def computeGeometricSum(n: Int): Int ={
  if(n <= 0){
    0
  }
} else{
    n + computeGeometricSum(n - 1)
}

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

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

```scala
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
Recursive Example

```scala
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
Anagrams Example

```python
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 Monday's lecture
Anagrams Example

```scala
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

```scala
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
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 recursive call with the remaining characters
    - Append the removed character to all the returned anagrams
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!
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
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 each returned anagram, output is generated starting at the base case
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
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
  }
}
Anagrams Example

```scala
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 (Similar to Monday's lecture question)
    - The _ is shorthand syntax we can use when a function only has 1 input and its type can be inferred
Anagrams Example

```scala
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**

**Restriction:** No state is allowed in this question. Specifically, the keyword "var" is banned. (ie. You are expected to use a recursive solution)

**Method:** In a package named "functions" create an object named "Algebra" with a method named "factor" that takes an Int as a parameters and returns the prime factorization of that parameter as a List of Ints.

The following apply to this method:
- If the input is negative or 0, return an empty list
- If the input is 1, return an empty list (Note: This is different than the original question)
- Do not include 1 in the output for any inputs
- The order of the factors in the output List is undefined

Example: functions.Algebra.factor(12) can return List(2,2,3) -or- List(2,3,2) -or- List(3,2,2)

**Unit Testing:** Testing will be checked by AutoLab, though you are encouraged to use your tests from the last time we had this question (Be sure to update your test for 1 as the input)