Arithmetic Progressions

1. The following code would produce the desired output.

```
PROGRAM arith_practice_1
  INTEGER :: i, a

  a = 4

  DO i = 1, 15, 1
    a = a + 42
    WRITE(*,*) a
  END DO
END PROGRAM arith_practice_1
```

This would produce the output:

```
          46
          88
         130
         172
         214
         256
         298
         340
         382
         424
         466
         508
         550
         592
         634
```

2. The following code would produce the desired output.

```
PROGRAM arith_practice_2
  INTEGER :: i, a

  a = 3

  DO i = 1, 6, 1
    a = a + 8
    IF (i >= 3) THEN
      WRITE(*,*) a
    END IF
  END DO
END PROGRAM arith_practice_2
```

This would produce the output:

```
          27
          35
          43
          51
```

3. The code would produce a sequence of numbers in the correct order and values, but each would not be the value at index `i`. Using the solution to practice question 1., the output would be:

```
          46
          88
         130
```

```
 172
 214
 256
 298
 340
```

The first eight elements in the sequence. However, these are the values a_1, a_2, a_3...a_8. If they were each meant to be the values at index `i` in the code (ie a_1, a_3, a_5 etc, with `i` incrementing by two), they would not be the values for those indices. This is because the calculation itself, `a = a + 42` is only dependent on how many times it has been performed, not what the specific value of `i` is.

Geometric Progressions

1. The following code would produce the desired output.

```
PROGRAM geo_practice_1
  INTEGER :: i, a

  a = 1

  DO i = 1, 5, 1
    a = a * 10
    WRITE(*,*) a
  END DO
END PROGRAM geo_practice_1
```

This would produce the output:

```
     10
    100
   1000
  10000
 100000
```

2. The following code would produce the desired output.

```
PROGRAM geo_practice_2
  INTEGER :: i, a

  a = 3

  DO i = 1, 16, 1
    a = a * (-2)
    WRITE(*,*) a
  END DO
END PROGRAM geo_practice_2
```

This would produce the output:

```
    -6
    12
   -24
    48
   -96
   192
  -384
   768
 -1536
  3072
 -6144
 12288
```

```
-24576
 49152
-98304
196608
```

Fibonacci Sequence

Loop Limits: The loop goes from `2` to `total-1` in order to print the correct index numbers along with the values themselves, and end with the correct total number of values. Since the first value is a_0, if we wanted 20 values, that number is reached once we have a_{19}, not a_{20} (which would be a total of 21 values, 1 to 20 inclusive, plus our starting zero).

Additional Consideration: The reason for the presented Fibonacci Sequence program not working for values beyond a_{46} is due to size of the `INTEGER` variable type. Being a fixed size in memory, it has upper and lower value bounds, -2,147,483,648 to 2,147,483,647 for Fortran `INTEGER`s on most systems. The 47^{th} value in the Fibonacci sequence is higher than the upper bound, $a_{47} = 2,971,215,073$; this causes an 'overflow' error, in which the `INTEGER` value in memory undesirably cycles round to a form in which it stores negative values. If limited precision is required, a simple `REAL` could be used instead of `INTEGER`, which permits numbers which are larger, but to a limited precision (only around seven decimal places). This would mean that any value above about 99,999,999 would be truncated, eg. 100,000,003 could only be represented as 1.0000000×10^8, 100,000,000. *(See below for a note regarding this matter.) Alternatively, one could use a variable type with twice the precision. For floating point values, this is permitted natively with `DOUBLE PRECISION`, a decimal number with twice the precision of the regular `REAL` (technically using twice as much memory, the scaling of its precision isn't quite the same).

*The definition of 'precision' given here, describing a number of decimal places of a regular decimal value is not strictly true. As all computer values are stored in binary, the value stored is not intrinsically a decimal value, and importantly does not have the same intervals as decimal values do. The smallest interval for a given precision in base 2 is different to the smallest interval for a given precision in base 10. This means that it is difficult to define how many 'decimal places' a floating point value is stored to. This also has an interesting effect that some base 10 numbers are not representable in binary to the displayed precision. The closest binary representation of decimal 0.005 with the available precision of a `REAL` is `4.99999989E-03` when rendered back into decimal for display. This can occasionally cause problems with accuracy, especially if many such representations are combined. This has to be taken into account when discussing the 'precision' of a computer stored value formally.