The algorithm used in my assignment is iterative. While it could easily be transformed into a recursive algorithm I am personally more comfortable working with iterative algorithms. The method that contains this algorithm accepts two sorted integer arrays and an integer to define what nth term it is finding. It assumes the arrays are sorted in ascending order.

```
@Override
public int findTerm(int[] one, int[] two, int term)
{
    int eleOne = one.length - 1;
    int eleTwo = two.length - 1;
    int count = 0, max = 0;
```

As you can see at the top of the method, several variables are declared. The first two being the array index locations being declared to the array length minus one. This allows my algorithm to examine the arrays starting from the last element. The variables count and max will be used to hold the current nth term and the related integer in the arrays respectively.

```
while (eleOne >= 0 || eleTwo >= 0)
{
    //If count reaches the correct term. Return the value.
    if (count >= term)
    {
        return max;
    }
```

My algorithm is programmed to deal with a total of five cases. The first case being that count equals the term that I am looking for. If this is the case, the max is returned. Once again, this is the number that represents the searched for nth term.

```
else
{
    //If the index for array one is less than 0. The next max can be found in two.
    if (eleOne < 0 && eleTwo >= 0)
    {
        max = two[eleTwo];
        eleTwo--;
        count++;
    }
```

The second case, is executed when the element index for array one is less than zero. This means that I have examined all of the elements in array one and the next largest can be found in array two. This results in a new max being assigned, the array index of array two being

decremented, and the counter for the nth term being incremented. This pattern is similar for all the cases.

```
//Else if, the index for array two is less than 0. The next max can be found in one.
else if (eleTwo < 0 && eleOne >= 0)
{
    max = one[eleOne];
    eleOne--;
    count++;
}
```

The third case is executed when the element index for array two is less than zero. This means that I have examined all of the elements in array two and the next largest element can be found in array one.

```
//Else if, the value at one might be greater than or equal to the value at two.
else if (one[eleOne] >= two[eleTwo])
{
    max = one[eleOne];
    eleOne--;
    count++;
}
```

The fourth case is executed when the value at the current index of array one is equal or greater than the value at the current index of array two. This means that the next largest value can be found at the index of array one.

```
//Else, the value at two must be greater than the value at one.
else
{
    max = two[eleTwo];
    eleTwo--;
    count++;
}
```

The last case is executed when all else fails. This means that the value in array two must be greater than the value at the current index of array one. This means that the next largest value can be found in array two.

```
//If the nth term couldn't be found. Return -1.
return -1;
```

If both arrays have been searched fully, the nth term doesn’t exist, and negative one is returned.