Q1

```
/* Algorithm to recursively compute a+b, where a and b are non-neg integers */
add( int a, int b )
   if( b == 0 )
      return a;
   return add( a+1, b-1 );

/* Test case */
a = 3, b = 2
add( 3, 2 )
   return add( 4, 1 )
      return add( 5, 0 )
         return 5
= 5
```

Q2

We need to find the sum of the array by adding each item recursively, and then divide by the number of elements in the array (N = array.length). There will be three cases, as follows:

1. i == 0
2. i == N-1 (base case)
3. all other values of i

Thus, the Java/pseudocode will be as follows:

```
/* Java/pseudocode for average of array */
averageArray( int[] A, int i, int N )
   if( i == 0 ) // Case 1 - this case returns the average
      return ( A[i] + averageArray( A, i+1, N ) ) / N;
   else if( i == N-1 ) // Case 2 (base case)
      return A[i];
   else  // Case 3
      return A[i] + averageArray( A, i+1, N );
```

Then one could simply call averageArray(A, 0, A.length) with the initial call having i = 0 as a requirement for proper calculation of average.

Q3

There are three cases here as follows:

1. Case 1 (base case 1) → n == 0, return f0
2. Case 2 (base case 2) → n == 1, return f1

3. Case 3 → n > 1, return gfib(f0, f1, n-1) + gfib(f0, f1, n-2)

The Java/pseudocode is as follows:

```java
/* Java/pseudocode for generalized fibonacci */
gfib( int f0, int f1, int n )
  if( n == 0 )
    return f0;
  else if( n == 1 )
    return f1;
  else
    return gfib(f0, f1, n-1) + gfib(f0, f1, n-2);
```

Q4

For this question I wrote and executed a Java program to compute Ackerman(2,2). The code is as follows:

```java
public class Ackerman
{
    public static void main(String[] args)
    {
        System.out.println("Ackerman(2,2) = " + ack(2,2));
    }

    public static int ack(int m, int n)
    {
        if( m == 0 )
        {
            return n + 1;
        }
        else if( n == 0 )
        {
            return ack(m-1, 1);
        }
        else
        {
            return ack(m-1, ack(m, n-1));
        }
    }
}
```

The result of this is:

```
C:\Users\Sean\Documents\jhu-cs\Data Structures\Module 3>java Ackerman
Ackerman(2,2) = 7
```

Q5

For this problem, it is simple to convert the recursive function to an iterative function using a while loop. The Java/pseudocode for this is as follows:

```
int rec(int n)
   bool x = f(n);
   int a = n;
   while( x == FALSE )
      /* any group of statements that do not change the value of n */
      a = g(a);
      x = f(a);
   return 0;
```

In this way, f(n) is initially calculated using n. By setting a variable 'a' equal to n, n remains unchanged and a can be updated to equal g(n) and so on recursively. Finally, when x = f(a) returns TRUE, the method returns 0.

Q6

There are four obvious ways to implement the queue. They are...

- Sorted array
- Sorted linked list
- Unsorted array
- Unsorted linked list

For my ADT, I chose to use an unsorted array, with methods modified to account for the additional requirements (namely searching for highest priority item in array and shifting items for deletion).

```
ADT PriorityQueue
   Data
      An empty unsorted array of values with a reference to the first (front) item, which
          is highest priority. The queue stores information on two parts of an item - the
          data and the priority
   Methods
      isEmpty
         Input   None
         Precondition  None
         Process    Check if the queue contains any data items
         Postcondition   None
         Output  Return true if queue is empty, and false otherwise
      Insert
         Input   A data item to be stored in the queue
         Precondition  Item has an assigned priority
         Process    Store an item at the rear of a queue
         Postcondition   The queue contains one additional data item
         Output  None
      Delete
         Input   None
         Precondition  Queue contains meaningful data values
```

```
        Process    Remove the highest priority element, accomplished by searching the
            queue
        Postcondition  Shift all elements greater than index of removed element left by
            one element
        Output  Return deleted value
     Peek
        Input   None
        Precondition   Queue contains meaningful data values
        Process  Search queue for highest priority item
        Postcondition  None
        Output  Return the value of the data item at the front of the queue (highest
            priority)
end ADT PriorityQueue
```