Assignment 3 – Recursion

Write pseudo-code not Java for problems requiring code. You are responsible for the appropriate level of detail.

Q1 and Q2 are intended to help you get comfortable with recursion by thinking about something familiar in a recursive manner. Q3 – Q6 are practice in working with non-trivial recursive functions. Q7 deals with the idea of conversion between iteration and recursion.

1. Write a recursive algorithm to compute *a*+*b*, where *a* and *b* are nonnegative integers.

```
# this is weird but…
int addTwoNumbers(a,b)
{
   # this is the base case
   if(b == 0)
      return a;

   # This adds m as the base case and then 1 n times to it. It funky yo
   answer = add(a, b-1) + 1;
   return answer;
}
```

2. Let A be an array of integers. Write a recursive algorithm to compute the average of the elements of the array. Solutions calculating the sum recursively, instead of the average, are worth fewer points.

```
avgOfArray(int a[], int i, int n)
      // return last element, this is the base case
      if (i == n-1)
         return a[i];

      // When index is 0, start adding up everything.
      if (i == 0) {
         // divide by n at the very end, sum all the numbers first before dividing by 9. This is
        // sublty different because of the division than the recursive call at the end
         double average = (a[i] + avgOfArray(a, i+1, n))/n;
         return average;
      }
```

```
    // Compute sum but do not divide until the end.
    return (a[i] + avgOfArray(a, i+1, n));
  }


   average(int arrayOfNumbers, int n)
     length = arrayOfNumbers.length
     return avgOfArray(arrayOfNumbers, 0, length);
```

3. If an array contains n elements, what is the maximum number of recursive calls made by the binary search algorithm?

LogX is the max number of calls. (in base2). I think if you check boundaries it could be LogX + 1 more call.

Say we have 1,2,3,4,5,6,7,8 for 8 numbers.

Mid	Low	High	Call Number
4.5	1	8	Original call
2.5	1	4	1
1.5	1	2	2
1	1	1	3

LogX = 3 which is the max calls.

Algo could look like this:

```
private int binarySearch(int[ ] a, int x, int low, int high) {
        if (low > high)
                return -1;
        int mid = (low + high)/2;
        if (a[mid] == x)
                return mid;
        else if (a[mid] < x)
                return binarySearch(a, x, mid+1, high);
        else // a[mid] > x
                return binarySearch(a, x, low, mid-1);
}
```

4. The expression m % n yields the remainder of m upon (integer) division by n. Define the greatest common divisor (GCD) of two integers x and y by:

gcd(x, y) = y	**if (y <= x and x%y == 0)**
gcd(x, y) = gcd(y, x)	**if (x < y)**
gcd(x, y) = gcd(y, x%y)	**otherwise**

Write a recursive method to compute gcd(x,y).

```
int gcd(int x, int y) {
  if ((x % y) == 0)
    return y;
  if (x < y)
    return gcd(y, x)
  else
    return gcd(y, x%y);
}
```

5. A generalized Fibonacci function is like the standard Fibonacci function,, except that the starting points are passed in as parameters. Define the generalized Fibonacci sequence of f0 and f1 as the sequence gfib(f0, f1, 0), gfib(f0, f1, 1), gfib(f0, f1, 2), ..., where

gfib(f0, f1, 0) = f0
gfib(f0, f1, 1) = f1
gfib(f0, f1, n) = gfib(f0, f1, n-1) + gfib(f0, f1, n-2) if n> 1

Write a recursive method to compute gfib(f0,f1,n).

The whole point is that we can pass in the base cases as f0 and f1.

```
GenFib(f0, f1, n)
        if n == 0
                return f0
        elif n == 1
                return f1
        else
        return genFib(f0, f1, n-1) + genFib(f0, f1, n-2)
```

6. Ackerman's function is defined recursively on the nonnegative integers as follows:

a(m, n) = n + 1	if m = 0
a(m, n) = a(m-1, 1)	if m DNE 0, n = 0

a(m, n) = a(m-1, a(m, n-1)) if m ≠ 0, n 0

Using the above definition, show that a(2,2) equals 7.

- a(2,2) = a(1, a(2, 1))
- a(2,1) = a(1, a(2, 0)) = a(1, 3)
- a(2,0) = a(1, 1) = 3
- a(1,1) = a(0, a(1, 0)) = a(0,2) = 3
- a(1,0) = a(0, 1)
- a(0,1) = 2
- a(1,3) = a(0, a(1, 2)) = a(0, 4) = 5
- a(1, 2) = a(0, a(1, 1)) = a(0, 3) = 4
- a(1,3) = a(0, 4) = 5 = a(2,1)
- **a(2,2) = a(1,5) = a(0, a(1,4)) = a(0, 6) = 7**
- a(1,4) = a(0, a(1, 3))) = a(0,5) = 6

Weird but it is == 7.

7. Convert the following recursive program scheme into an iterative version that does not use a stack. *f(n)* is a method that returns TRUE or FALSE based on the value of n, and *g(n)* is a method that returns a value of the same type as *n* (without modifying *n* itself).

recursive method:

```
int rec(int n)
{
   if ( f(n) == FALSE ) {
      /* any group of statements that  do not change the value of n */
      return (rec(g(n)));
   }//end if
   return (0);
}//end rec
```

```
# iterative
while (f(n) == FALSE {
      /* any group of statements that  do not change the value of n */
      n = g(n)
}
return 0
```