**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 and Q8 deal 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.**

Class Compute(int a, int b, int c){

Int Value;

If (c >= b){

Value = a;

} else {

Value = Compute(a+1,b,c+1)

}

Return Value;

}

Class main(){

Compute(a,b,0);

}

**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.

Float average(int[] arr, float Avg, int i){

Float Value;

If (i==0){

Value = Avg+a[i]/length(arr) //Once I is 0, calculate the last “average” and add it to the previous, and assign it to value. Length(arr) is used for the denominator.

} else {

Value = average(arr,Avg+a[i]/length(arr),i-1) //else pass the new average to the next recursion

}

Return Value;

}

Main(){

Average(arr, 0, length(arr) -1) //Start at length-1 to find last element of array (assuming array starts at 0)

}

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

An array of 50

So it seems like with 50 elements, it would be a max of 6 times. With 100, we would add one to it, or 7 times. With only 10 elements, it would be only 4 times (5->3->1->0). It appears that the cost is lg(n), rounded up.

2

0

1

4

9

3

9

3

3

9

9

3

43

31

18

6

12

37

25

**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).**

Class Gcd(x,y){

Int Value;

If (y<= x & x%y == 0){

Value = y;

} else if (x<y){

Gcd(y,x);

} else {

Value = gcd(y,x%y);

}

Return Value;

}

Class main(){

Gcd(10,4) //Should return 2

}

**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).**

Class Gfib(f0,f1,n){

Int Value;

If (n == 0){

Value = f0;

} else if (n==1){

Value = f1;

} else {

Value = gfib(f1,f0+f1,n-1);

}

Return Value;

}

Class main(){

Gfib(2,5,3) //should return 12

}

**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 ≠ 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.

Class Ackerman(m,n){

Int value;

If (m == 0){

Value = n+1;

} else if (m != 0, n ==0){

value = Ackerman(m-1,1);

} else {

Ackerman(m-1,a(m,n-1));

}

Ackerman(2,2)

m!=0,n!=0

Ackerman(1,Ackerman(2,1))

M!=0,n!=0

Ackerman(1,Ackerman(2,0))

M!=0,n=0

Ackerman(1,1)

M!=0,n!=0

Ackerman(0,Ackerman(1,0))

M!=0, N=0

Ackerman(0,1) = 2

Ackerman(1,0) = 2

Ackerman(0,2) = 3

Ackerman(1,1) = 3

Ackerman(2,0) = 3

Ackerman(1,3)

M!=0, N!=0

Ackerman(0,Ackerman(1,2))

M!=0,N!=0

Ackerman(0,Ackerman(1,1))

M!=0,N!=0

Ackerman(0,Ackerman(1,0))

M!=0,N=0

Ackerman(0,1) = 2

Ackerman(1,0) = 2

Ackerman(0,2) = 3

Ackerman(1,1) = 3

Ackerman(0,3) =4

Ackerman(1,2) = 4

Ackerman(0,4) = 5

Ackerman(1,3) =5

Ackerman(2,1) =5

Ackerman(1,5)

M!=0, N!=0

Ackerman(0,Ackerman(1,4))

M!=0,N!=0

Ackerman(0,Ackerman(1,3))

##To save some steps, as shown before, Ackerman(1,3) =5

Ackerman(0,5) = 6

Ackerman(1,4) = 6

Ackerman(0,6) = 7

Ackerman(1,5)=7

Acker(2,2)=7

**7. Show how to transform the following iterative procedure into a recursive procedure. *f(i)* is a method returning a logical value based on the value of *i*, and *g(i)* is a method that returns a value with the same attributes as *i*.**

void iter(int n)

{

int i;

i = n;

while ( f(i) == TRUE ) {

/\* any group of statement that \*/

/\* does not change the value of i \*/

i = g(i);

} // end while

} //end iter

Void iter(int n) {

Int I;

Int Value;

If (f(n) == FALSE){

Value = n;

} else {

/\* any group of statement that \*/

/\* does not change the value of i \*/

I = g(n);

Value = Iter(I);

}

Return Value;

}

**8. 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*).**

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

Int rec(int n){

Int I;

I = n;

While(f(i) == FALSE){

/\* any group of statements that do not change the value of n or i \*/

I = g(i);

}

Return I;

}