**Mason Shepherd**

**Homework 3: due February 6th 11:59PM.**

R-5.1 Describe a recursive algorithm for finding the maximum element in an array, *A*, of *n* elements. What is your running time and space usage?

Ans:

**Algorithm** max(*A, low, high*):

**if** low == high **then**

**return** A[*low*]

**else**

mid = (low + high)/2 {integer division}

maxLeft = max(A, low, mid)

maxRight = max(A, mid+1, high)

**if** maxLeft > maxRight **then, return** maxLeft

**else**, **return** maxRight

For recursive version: Time-Complexity is O(logn); Space-Complexity O(logn)

For iterative version: Time-Complexity is O(n); Space-Complexity is O(1)

C-5.17 Write a short recursive Java method that takes a character string*s* and outputs its reverse. For example, the reverse of 'pots&pans' would be 'snap&stop'.

Ans:

public static String recursiveReverse(String s){

if (s.length() == 1 || s.isEmpty()) // base case 1

return s;

else if (s.length() == 2){ // base case 2

String newS = String.format("%c%c", s.charAt(1), s.charAt(0));

return newS;

}

else{

int last = s.length() -1;

String midS = s.substring(1, last);

String revMid = recursiveReverse(midS); // recursive portion

String newS = String.format("%c%s%c", s.charAt(last), revMid, s.charAt(0));

return newS;

}

}

// the code for this is attached as well

C-5.25 Describe a **fast recursive** algorithm for reversing a singlylinkedlist *L*, so that the ordering of the nodes becomes opposite of what it was before.

Ans:

**Algorithm** revSLL(*L*)

***INPUT***: a singly linked list *L*

***OUTPUT***: *L* reversed

**if** size == 0 or size == 1

**return**

**else**

last = head

L.removeFirst()

revSLL(L)

L.addLast(last)

**return**

**//** this is O(n), right? (Not part of the question, I’m just curious if I got it right)

C-5.16 In the ***Towers of Hanoi*** puzzle, we are given a platform with three pegs, *a*, *b*, and *c*, sticking out of it. On peg *a* is a stack of *n* disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg *a* to peg *c*, moving one disk at a time, so that we never place a larger disk on top of a smaller one. See Figure 5.15 for an example of the case *n* = 4. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary *n*. (Hint: Consider first the subproblem of moving all but the *n* th disk from peg *a* to another peg using the third as “temporary storage.”)

Ans:

**Algorithm** hanoi(*n*, *A*, *B*, *C*)

***INPUT***: *n* disks, start position *A*, spare position *B*, target/end position *C*

***OUTPUT***: each step to complete task with minimal steps

**if** *n* == 1 **then**

**print** (“Move a disc from ”, *A*, “to ”, *C*, “.”)

**return**

**else**

hanoi(*n-1*, *A*, *C*, *B*)

**print**(“Move a disc from ”, *A*, “to ”, *C*, “.”)

hanoi(*n-1*, *B*, *A*, *C*)

**return**

**Project 2**

1. [20 points] Compile and Test the recursive algorithm **factorial** in the textbook (page 191). You can also use different programming language
2. Using the last two digits of your L number as an argument to run the algorithm.
3. Verify and report the output of the algorithm.
4. Turn in your complete Java code as well as the output of your program in a separate file.

**Grading:**

Programs compile 50%

Programs give correct output: 50%