1. For a doubly-linked lists with headers, where each element stores a non-negative integer, using the basic list operations presented in class (CREATE, FIRST, LAST, SIZE, NEXT, PREV, INSERT, DELETE, DATA, etc.), give a recursive algorithm to determine the maximum value in the list (i.e., it returns an integer equal to the maximum of any value stored in the list and leaves the list unchanged). Analyze the time used.

First recall the iterative algorithm:

\[
\begin{align*}
    p & := \text{FIRST}(L) \\
    m & := 0 \\
    \textbf{while } p \neq \text{nil} \textbf{ do begin} \\
        m & := \text{MAX}\{m, \text{DATA}(p)\} \\
        p & := \text{NEXT}(p) \\
    \textbf{end}
\end{align*}
\]

For a recursive algorithm, we can compare the value of the first element of a list pointed to by a pointer \( p \) with a recursive call on the tail of the list.

\[
\begin{align*}
\text{function } \text{MAXVAL}(p) \\
\text{if } p = \text{nil} \text{ then return } 0 \\
\text{else return } \text{MAX}\{\text{DATA}(p), \text{MAXVAL}(\text{NEXT}(p))\}
\end{align*}
\]

For lists that have headers, we can call \( \text{MAXVAL}(\text{FIRST}(L)) \)

\textit{Time:} The algorithm spends \( O(1) \) time in addition to the call on the tail of the list. For a list of size \( n \), \( n \) calls are made, each requiring \( O(1) \) time for a total of \( O(n) \) time.
2. Recall the recursive version of binary search presented in class. Give a recursive algorithm to perform 3-ary search that is a generalization of the recursive binary search presented in class. That is, divide the portion of the sorted array \( A \) from position \( a \) to position \( b \) \((0 \leq a \leq b)\) that is to be searched for a value \( x \) into 3 regions instead of 2 regions. Analyze the time used.

\[
\text{function } \text{RBS3}(a, b) \\
\text{if } (a+1) < b \text{ then begin} \\
\hspace{1cm} m1 := \left\lfloor a + \frac{(b-a)}{3} \right\rfloor \\
\hspace{1cm} m2 := \left\lfloor a + 2\frac{(b-a)}{3} \right\rfloor \\
\hspace{1cm} \text{if } x \leq A[m1] \text{ then return } \text{RBS3}(a, m1) \\
\hspace{1cm} \text{else if } x \leq A[m2] \text{ then return } \text{RBS3}(m1 + 1, m2) \\
\hspace{1cm} \text{else return } \text{RBS3}(m2 + 1, b) \\
\text{end} \\
\text{else if } x = A[a] \text{ then return } a \\
\text{else if } x = A[b] \text{ then return } b \\
\text{else return } -1 \\
\end{function}
\]

\textit{Time:} Since each recursive call reduces the size of the region to be searched by a factor of 3, the time is \( O(\log(n)) \).
3. Describe in English and give pseudo code for an algorithm \textsc{MULTP}(n,A,B) that multiplies two polynomials \(A[n-1]x^{n-1} + \ldots + A[1]x + A[0]\) and \(B[n-1]x^{n-1} + \ldots + B[1]x + B[0]\), where all values are \(\geq 0\) and \(n\) is a power of 2, and returns an array \(C\) of \(2n\) elements containing the coefficients of their product (padded with 0's beyond the highest non-zero term), where it works by employing recursive calls to the multiplication of polynomials of degree \(n/2\).

If we rewrite a polynomial in terms of the upper and lower halves of its coefficients to be

\[
P(x) = x^{n/2}P_{\text{upper}} + P_{\text{lower}}
\]

then for two \(n\)\(^{th}\) degree polynomials \(P\) and \(Q\):

\[
PQ = (x^{n/2}P_{\text{upper}} + P_{\text{lower}})(x^{n/2}Q_{\text{upper}} + Q_{\text{lower}}) = x^nP_{\text{upper}}Q_{\text{upper}} + x^{n/2}(P_{\text{upper}}Q_{\text{lower}} + P_{\text{lower}}Q_{\text{lower}}) + P_{\text{lower}}Q_{\text{lower}}
\]

For example:

\[
(ax^3 + bx^2 + cx + d) \times (ex^3 + fx^2 + gx + h) = [(ax+b)x^2 + (cx+d)] \times [(ex+f)x^2 + (gx+h)] = (ax+b)(ex+f)x^4 + [(ax+b)(gx+h) + (cx+d)(ex+f)]x^2 + (cx+d)(gx+h) = aex^6 + (af+be)x^5 + (ag+bf+ce)x^4 + (ah+bg+cf+de)x^3 + (bh+cg+df)x^2 + (ch+dg)x + dh
\]

\[
\text{function \textsc{MULTP}(n,A,B) if } n < 1 \text{, or } n \text{ is not a power of 2, or } A \text{ or } B \text{ are not arrays of } n \text{ elements RETURN ERROR else if } n = 1 \text{ return } C[0] = A[0]*B[0] else begin}
\]

/*Break the arrays into upper and lower halves:*/

\[A0 = A[0] \ldots A[n/2-1]\]
\[A1 = A[n/2] \ldots A[n-1]\]
\[B0 = B[0] \ldots B[n/2-1]\]
\[B1 = B[n/2] \ldots B[n-1]\]

/*Recursively multiply the halves:*/

\[U = \textsc{MULTP}(n/2, A1, B1)\]
\[V = \textsc{MULTP}(n/2, A1, B0)\]
\[W = \textsc{MULTP}(n/2, A0, B1)\]
\[X = \textsc{MULTP}(n/2, A0, B0)\]

/*Enlarge the arrays to length \(2n\) by padding with 0's:*/

Replace \(U,V,W,X\) with copies that have \(2n\) elements, where the upper \(n\) elements are 0's.

/*Shift the coefficients appropriately (X is ok as is):*/

Shift the coefficients of \(U\) forward \(n\) positions.
Shift the coefficients of \(V\) and \(W\) forward \(n/2\) positions.

/*Add up the four polynomials and return the result:*/

\[\text{return } (U+V+W+X)\]

end

\textit{NOTE:} As we shall see later in the course, time is \(O(n^2)\) for the straightforward approach above, which is no better than using the standard method. However it can be \(O(n^{3/2})\) by using more additions (and subtractions) but only 3 recursive calls, and it can be done in \(O(n\log(n))\) time.
4. Consider the following algorithm:

```python
function A(i,n)
    if i=0 then return n+1
    else if n=0 then return A(i-1,1)
    else return A(i-1,A(i,n-1))
end
print A(0,0)
print A(1,1)
print A(2,2)
print A(3,3)
print A(4,4)
```

Implement this program in the language of your choice, include a print out of the program with this assignment, and answer what happens when you run it.

Here is a C program:

```c
#include <stdio.h>
int A(int i,int n) {
    if (i==0) return (n+1);
    else if (n==0) return A(i-1,1);
    else return A(i-1,A(i,n-1));
}
int main() {
    int i;
    for (i=0; i<=4; i++) printf("A(%d,%d) = %d\n",i,i,A(i,i));
    return 0;
}
```

When the program runs, what happens is:

A(0,0) = 1
A(1,1) = 3
A(2,2) = 7
A(3,3) = 61
A(4,4) = too big to calculate

**Note:**

\[ A(0,n) = n + 1 \]
\[ A(1,n) = n + 2 \]
\[ A(2,n) = 2n + 3 \]
\[ A(3,n) = 2^{n+3} - 3 \]
\[ A(4,n) = \frac{2^{n+3} - 3}{n} + 3 \]