ESO207 Programming Assignment-1 Pseudo-Codes

(a) Algorithm and pseudo-code for this problem is similar to the set union in lecture 12.

Algorithm 1: PolyAdd(A, B)

```
Data: A, B /* Doubly linked lists representing two polynomials */
Result: List representing A + B is returned.
i ← (A.nil).next, j ← (B.nil).next;
Create(C) ;  //An empty doubly linked list, sentinel node only, is created in C.
while (i ≠ A.nil) and (j≠ B.nil) do
    if i.exp == j.exp then
        a ← i.coeff + j.coeff;
        if a ≠ 0 then
            n ← new() ;              //Creates a fresh node of doubly linked list.
            n.coeff ← a;
            n.exp ← i.exp;
            insertEnd(C, n);
        end
        i ← i.next;
        j ← j.next;
    else
        if (i.exp < j.exp) then
            n = new();
            n.coeff = i.coeff;
            n.exp = i.exp;
            insertEnd(C, n);
            i = i.next;
        else
            // j.exp < i.exp
            n = new();
            n.coeff = j.coeff;
            n.exp = j.exp;
            insertEnd(C, n);
            j = j.next;
        end
    end
end
if i == A.nil then
    Copy(B,j,C)
else
    Copy(A,i,C)
end
return C
```

Algorithm 2: InsertEnd(C,n)

Data: List C, node n
Result: Node n is appended at the end of C

```
n.next ← C.nil;
n.prev ← (C.nil).prev;
(n.next).prev ← n;
(n.prev).next ← n;
```

Algorithm 3: Copy(A, i, C)

Data: List A, Node i in A, list C
Result: Copies list A from node i onwards at the end of C

```
j ← i;
while j ≠ A.nil do
    InsertEnd(C, j);
    j ← j.next
end
```

Let n, m be the number of terms in A, B respectively. The complexity of PolyAdd(A,B) is easily seen to be $O(n + m)$.

(b) A simple strategy is to multiply each term of the A (or B) to the polynomial B (or A). This results in as many polynomials as the number of terms in A (or B). We now add all these polynomials. Following pseudo-code does this.

Algorithm 4: Mult

Data: Polynomial A, coefficient $a \neq 0$, exponent e
Result: Multiplies polynomial A with $a.x^e$. Returns the resulting polynomial.

```
Create(C);
i ← (A.nil).next;
while i ≠ A.nil do
    n ← new();
    n.coeff ← a · (i.coeff);
    n.exp ← i.exp + e;
    InsertEnd(C,n)
end
return C
```

Algorithm 5: PolyMult

```
Data: Polynomials A,B
Result: Returns A · B
//n = number of terms in A and m = number of terms in B.
//Takes n · m^2 time.
Create(C);
j ← (B.nil).next;
while j ≠ B.nil do
    D ← Mult(A, j.coeff, j.exp);
    C ← PolyAdd(C, D);
    j ← j.next
end
return C
```

Complexity Analysis of Algorithm PolyMult

Let $n = length(A)$ and $m = length(B)$. Consider the invariant that at the beginning of i^{th} iteration of while loop, C has at most $(i-1)\cdot n$ terms and at the end of this iteration, C has at most $i \cdot n$ terms.

This is easily verified. D is a polynomial with n terms. So, in the i^{th} iteration of while loop, PolyAdd(C,D) gives a polynomial with at most $(i-1)\cdot n + n = i \cdot n$ terms.

Time taken in this addition is $O(i \cdot n)$. Time taken in computing D is $O(n)$. Therefore total time taken in the i^{th} iteration of while loop is $O(i \cdot n)$.

Time taken by PolyMult is therefore

$$\Sigma_{i=1}^{m} c \cdot (i \cdot n) = cn(\Sigma_{i=1}^{m} i) = cn(\frac{m(m+1)}{2})$$

This is $O(n \cdot m^2)$.

Improvements

Time $O(mn \cdot log\ m)$ may be obtained if we add m polynomials in PolyMult in a different way. First we group these polynomials into groups of size 2 each and add polynomials in each group separately. Total time taken in this step is $c \cdot \frac{m}{2} \times (2n) = c \cdot mn$. At the end of this step, we have $\frac{m}{2}$ polynomials each with at most $2n$ terms.

Doing this step on the new set, again requires $c \cdot mn$ time and gives $\frac{m}{4}$ polynomials each with at most $4n$ terms each.

Continuing in this way, we obtain a single polynomial after about $log\ m$ passes with each pass requiring $c \cdot mn$ time. This gives the total time to be $O(mn \cdot log\ m)$.

Following is a pseudo-code for this algorithm.

Algorithm 6: PolyMultImproved

```
Data: Polynomials A,B
Result: Returns A · B
// n = number of terms in A and m = number of terms in B.
//Takes nm · log m time.
Create(E);
//E is a list of polynomials.
//Each node in E has three fields poly, next and prev.
//poly field contains (pointer to) a polynomial,
//next field points to the next node in list E
j ← (B.nil).next;
while j ≠ B.nil do
    D ← Mult(A, j.coeff, j.exp);
    Insert(E, D);
    //Inserts polynomial D at the end of list E
    j ← j.next
end
if E==empty then
    return (Zero polynomial)
end
while ((E.nil).next).next ≠ E.nil do
    //E has more than one polynomial
    Create(F);
    //F is a temporary list to store sum of pairs of polynomials in E
    j ← (E.nil).next;
    while j ≠ E.nil do
        k ← j.next;
        if k ≠ E.nil then
            Insert(F,PolyAdd(j.poly, k.poly));
            //Adds two adjacent polynomials in E and
            //inserts the resulting polynomial in F
            j ← k.next
        else
            Insert(F, j.poly));
            Break
        end
    end
    CopyPoly(E,F);
    //Copies list F back to list E
end
return ((E.nil).next).poly
```

Another way to obtain complexity $O(mn \cdot min\{log\ m, log\ n\})$ is using the idea that m sorted lists of length n each can be merged (into a sorted list of length mn) in $(log\ m) \cdot mn$ time. Running pointers of each list may be stored in a heap. Of course, you were not expected to give this algorithm as we had not covered heaps by the time of this assignment.

Our final algorithm is $Main(A, B)$.

Algorithm 7: Main(A,B)

```
Data: Polynomial A, B
Result: A · B /* minimizes steps by changing the order of arguments in PolyMult */
n =length(A) ;                    //length computes number of terms in a polynomial
m =length(B);
if (n < m) then
  | return PolyMult(B,A)
else
  | return PolyMult(A,B)
end
```

Complexity of Main

Main takes $O(min\{mn^2, nm^2\}) = O(mn \cdot min(m, n))$ time. With PolyMultImproved, instead of PolyMult, Main takes time $O(mn \cdot min(log\ m, log\ n))$.