ESO207 Programming Assignment-1

-by Akhil Agrawal (200076) *and Aditya Tanwar* (200057)

Q1

Polynomials may be represented as linked lists. Consider a polynomial $p(x)$, with n non-zero terms,

$$p(x) = a_1 x^{e_1} + a_2 x^{e_2} + ... + a_{n-1} x^{e_{n-1}} + a_n x^{e_n}$$

where $0 \leq e_1 < e_2 < ... < e_{n-1} < e_n$ are (non-negative) integers. We assume that coefficients $a_1, ..., a_n$ are *non-zero* integers.

Polynomial $p(x)$ can be represented as a linked list of nodes. Each node has three fields: coefficient, exponent and link to the next node. Let us assume that list is a doubly linked list, with a sentinel node, sorted in ascending order of exponents.

(a) (marks 5+15) Write pseudo-code to add two polynomials $p(x)$ and $q(x)$ in this representation. Your algorithm should take $O(n + m)$ time, where n, m are the number of terms in $p(x)$, $q(x)$ respectively. Implement your pseudo-code as an actual program.

Pseudo-code:

```
Add ( P, Q ) :

    p , q = P.head , Q.head //iterators of polynomials P and Q
    R  = Polynomial //resultant polynomial
    t = node //temporary node

    while (p ≠ P.end) and (q ≠ Q.end) do:
    // append nodes to r in order of increasing exponents
    //if p or q has reached the end, simply append others' node to R
        if ( p == P.end )
```

```
        t.coef = q.coef
        t.expo = q.expo
        q = q.next
    else if ( q == Q.end )
        t.coef = p.coef
        t.expo = p.expo
        p = p.next
    else
        if ( p.expo > q.expo )
            t.coef = q.coef
            t.expo = q.expo
            q = q.next
        else if ( p.expo < q.expo )
            t.coef = p.coef
            t.expo = p.expo
            p = p.next
        else
            t.coef = p.coef + q.coef
            t.expo = p.expo
            p  = p.next
            q  = q.next
            if ( t.coef == 0 )    //if sum is zero, then node
                continue          //need not be appended

    append t at the end of R
```

(b) (marks 10+20) Write pseudo-code to multiply two polynomials $p(x)$ and $q(x)$ in this representation. Do runtime complexity analysis of your algorithm in terms of n, m, the number of terms in $p(x)$, $q(x)$ respectively. State this complexity in 'O' notation. Implement your pseudo-code as an actual program.

Pseudo-code:

```
Insert ( i , t ) :  //insert t just after i ( i, t are nodes )
        t.next = i.next
        t.prev = i
        i.next.prev = t
        i.next = t

Delete-Zeroes ( R ) : //deletes all nodes with co-efficient zero in R
        i = R.head
        While ( i ≠R.end ) do :
                temp = i.next
                if ( i.coef == 0 )
                        i.next.prev = i.prev
                        i.prev.next = i.next
                i = temp


Multiply ( P, Q ) :
        R, t = polynomial, node
        i = R.head //node that iterates over resultant polynomial
        p  = P.head
        while ( p ≠ P.end ) do:
                q = Q.head
                while( q ≠ Q.end ) do:
                        t.coef = p.coef * q.coef
                        t.expo = p.expo + q.expo

                        //find the node (in r) whose exponent
```

```
            //is just smaller or equal to t.expo
            if (q == Q.head)
                //for the first product of a node in p, it might
                //be needed to travel backwards to find the node
                while ( i.expo > t.expo ) do:
                    i = i.prev
            else
                //in case the required node is
                //ahead of the current node
                while true do:
                    if ( i.next == R.end )
                        break
                    if ( i.next.expo ≤ t.expo )
                        i = i.next
                    else
                        break

            if ( t.expo == i.expo )
                i.coef = i.coef + t.coef
            else
                Insert (i, t) //insert t just after i
                i = t

            q = q.next
        p = p.next
    Delete-Zeroes ( R )
```

Complexity Analysis:

Each node in P is accessed once and multiplied with each node in Q only once. Without loss of generality, we assume that P is the polynomial with lesser size, i.e., $n \leq m$. After all the multiplications of a node in P are completed, the iterator in R ends up at the end of the linked list[0].

During the multiplication of the first node in P with the nodes in Q, each new node in R is appended in $O(1)$ time since both multiplication[1] and insertion[2] at the end are completed in constant time. After all the multiplications of the first node in P, the iterator in R is at the end. As there are m multiplications, the total time taken will be $O(m)$[3] for the first node.

For each subsequent i^{th} node in P ($i \geq 2$), the iterator node of R has to travel at most to the $(i-1)^{th}$ node[4] in R i.e. $(m-1)*(i-1)$ traversals[5] backwards to find a node in R that has the maximum exponent ≤ exponent of the resultant node.
After this, the iterator can travel at most $(m-1)*i$ times forward[6].
It is guaranteed that the iterator lands at the end of R again[7] after the multiplications of this node in P are completed.
So for the i^{th} node in P the runtime complexity for the node is in $O(m*i)$[8].
As, for each node in P, the time complexity is $O(m*i)$, and there are n nodes in P, the time complexity for multiplying all the terms is $O(m*n^2)$[9].

Further, *Delete-Zeroes*(R) accesses each node in R only once and either traverses onto the next node, or deletes the node, both of which are O(1) operations. As there can be at most $n*m$ nodes in R, the overall time complexity for *Delete-Zeroes* is $O(m*n)$.

The runtime complexity is therefore $O(m*n^2)+O(m*n)=O(m*n^2)$. We had assumed initially that $n \le m$, so the overall complexity can actually be simplified to $O(max(m, n)*min(m, n)^2) = O(m*n*min(m, n))$.

Loop Invariants:

- Before each insertion, $i.expo \le t.expo$, where i is the iterator node in R and t is the resultant node to be inserted. After the insertion, i is updated to t.
  This ensures that after t is inserted, the polynomial R calculated until now, remains sorted in order of increasing exponents. Since an empty polynomial is trivially sorted, and R remains sorted after each insertion, the polynomial after Multiply(R) is completed, has all the nodes sorted in the order of increasing exponents.
- After a node in P has been multiplied with all the nodes in Q, the iterator i lands at the end of polynomial R. This is useful in calculation of maximum backward traversals.

[(0), (7)] Using the loop invariant, at all times, R is sorted with respect to exponents. Now since P is also sorted with respect to exponents, product with the last node of Q (which is also sorted), will lead to an exponent that is greatest amongst the exponents calculated in R up until that point and will be inserted in the end.

[(1)] Multiplication of two nodes in P and Q includes calculation of *t.coef* which is a simple multiplication of two integers and can be upper bounded by a constant, say c_m', and addition of two integers for *t.expo* which can be upper bounded by a constant, say c_a. So, multiplication of two nodes can itself be bounded by a constant $c_m = c_m' + c_a$ which is in O(1).

[2] Insertion of a node t in front of node i involves simple assignment of *four* addresses. Each assignment can be denoted by a time c_i', so insertion of a node can be bounded by a constant $c_i=4c_i'$ which is in O(1).

[3] There are m multiplications of nodes involved since there are m nodes in Q, with each node multiplication involving calculation of t.coef and t.expo, and insertion of the resultant node t. All of these operations are constant time (using [1] and [2]), so time taken for a particular resultant node is a constant, say, c_r=O(1). As there are m nodes in Q, the total time taken will be $m*c_r=O(m)$.

[4] The exponent of product of i^{th} node of P by the first node of Q is at least greater than the exponents of products of first $(i-1)$ nodes of P with the first node of Q.

[5] There will be at most $m*(i-1)$ nodes in R, when i^{th} node in P is reached. And using [4] we need to reach the $(i-1)^{th}$ node. Hence number of backward traversals required are $(m*(i-1) - (i-1)) = (m-1)*(i-1)$.

[6] Using [5] we are at $(m-1)*(i-1)$ nodes back from the end of R, and we need to insert at most $(m-1)$ distinct nodes, after insertion of the first node. Hence number of forward traversals required are $((m-1)*(i-1) + (m-1)) = (m-1)*i$.

[8] Traversing a node takes a constant time, say c_t . Using [5] number of backward traversals are $(m-1)*(i-1)$. Therefore time taken to travel backwards is $c_t*(m-1)*(i-1) \le c_t*m*i$.
Using [6] number of forward traversals are $(m-1)*i$. Therefore time taken to travel forwards is $c_t*(m-1)*i \le c_t*m*i$. Hence overall runtime complexity is $\le(2c_t)*m*i$ which is O(m*i).

[9] Total time complexity $T\le \Sigma c*m*i$ where $(1 \le i \le n)$ for some c, which gives $T \le c*m*n*(n-1)/2$. Hence overall time complexity is $T\le c*m*n*(n-1)/2 \le c*m*n*n = O(m*n^2)$.