

# Solution of ESO207 Programming Assignment-1

Harshit Raj                      Shubhan R

Aug 30, 2021

## 1 About this

- About Assignment

**Name:**     Programming Assignment 1

**Due on:**   Aug 31, 2021

- Team Member

**Name:**   Harshit Raj

**Email:**   harshitr20@iitk.ac.in

**Roll:**     200433

- Team Member

**Name:**   Shubhan R

**Email:**   shubhanr20@iitk.ac.in

**Roll:**     200971

- **HackerRank ID:** @shubhanr20

- About Course

**Course Code:**     ESO207A

**Name of course:**   Data Structure & Algorithm

**Offered by:**        Prof. Anil Seth

**Offered in:**        Semester 2021-22-I

There are two parts to the assignment: Theoretical & Practical. The theoretical part is to create and analyze algorithms and the practical part is to implement them into actual working codes.

This file includes Pseudo codes and complexity analysis of the Algorithm. Working code has been submitted to HackerRank Contest from group HackerRank ID using C++ programming language.

## 2 Problem Statement

### Instructions

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

$$p(x) = a_1x^{e_1} + a_2x^{e_2} + \dots + a_{n-1}x^{e_{n-1}} + a_nx^{e_n}$$

where  $0 \leq e_1 < e_2 < \dots < e_{n-1} < e_n$  are (non-negative) integers. We assume that coefficients  $a_1, \dots, a_n$  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 sentinel node, sorted in ascending order of exponents.

- (a) 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.
- (b) 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.

Note that output list should satisfy all constraints (non-zero coefficients, exponents in strict ascending order etc.) of representation of a polynomial. Make your code non-destructive, that is, it should not modify the lists for  $p(x)$  and  $q(x)$ .

### 3 Data Structure Used

In order to implement the problem statement an Abstract Data Type(ADT) ***Polynomial***. Each node of *Polynomial* class(Doubly Linked List) has the following structure:

```
Struct Node {
    int coeff;
    int exp;
    struct Node *next;
    struct Node *prev;
}
```

#### 3.1 Sentinel Node

A sentinel node is a specifically designated node used with linked lists as a traversal path terminator.

```
// Constructor initializing the sentinel node
Polynomial() {
    head = new node;
    head->next = head;
    head->prev = head;
}
```

#### 3.2 Input Method

This is a helper function to make linked list by taking input from standard input.

```
// input values to the list from standard input
void input(int n) {
    for (int i = 0; i < n; i++) {
        int coeff, exp;
        cin >> coeff >> exp; // Scan coefficient and exponent
        appendNode(coeff, exp); // Add the node to end of list
    }
}
```

### 3.3 Appending node to List

This helper function appends coefficient and exponent pair to the end of the polynomial.

```
// Append a node in the end of the list
void appendNode(int coeff , int exp) {
    // if coeff is 0 then do not append
    if (coeff == 0) return;

    // creating and appending the node
    newNode = createNode(coeff , exp);

    // Last node is prev node of head(i.e. sentinel node)
    last->next = newNode;
    newNode->prev = last;

    // Link such that new node becomes last node
    newNode->next = head;
    head->prev = newNode;
}
```

### 3.4 Delete Node from List

This helper function delete a particular node from the Linked list.

```
// Delete the node
void deleteNode(node *curr) {
    // fix the links around the node
    prev->next = next;
    next->prev = prev;

    // free the memory
    delete curr;
}
```

## 4 Polynomial Addition Algorithm

Algorithm for Polynomial addition is mentioned in this section. During this we will be provided with two proper polynomials, sorted in ascending order of exponents. The idea is to get proper function which is addition of two polynomials.

### 4.1 Pseudo code

*head* is the sentinel node of Polynomial.

```
// add two polynomials
void add(Polynomial p1, Polynomial p2)
{
    // loop thru the list of p1 and p2
    node *temp1 = p1.head->next;
    //since p1.head points to sentinel node
    node *temp2 = p2.head->next;

    // loop till the either of the loop reaches its end
    while (temp1 != p1.head && temp2 != p2.head)
    {
        // If exponent is same then add the coefficients
        if (temp1->exp == temp2->exp)
        {
            // append this pair at end
            appendNode(temp1->coeff + temp2->coeff, temp1->exp);

            // move to next node in both polynomials
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        // If exponent of p2 is greater than p1
        // then add the node of p1
        else if (temp1->exp < temp2->exp)
        {
            appendNode(temp1->coeff, temp1->exp);
            // move to next node in polynomial1
```

```

        temp1 = temp1->next;
    }
    // If exponent of p1 is greater than p2
    // then add the node of p2
    else {
        appendNode(temp2->coeff , temp2->exp);
        // move to next node in polynomial2
        temp2 = temp2->next;
    }
}

// when only p1 is left keep adding p1
// Simply append till end
while (temp1 != p1.head)
{
    appendNode(temp1->coeff , temp1->exp);
    temp1 = temp1->next;
}

// when only p2 is left keep adding p2
// Simply append till end
while (temp2 != p2.head)
{
    appendNode(temp2->coeff , temp2->exp);
    temp2 = temp2->next;
}
}

```

## 4.2 Mathematical Completeness

The idea is fairly simple. We first start with starting node of each polynomial. By definition, the starting node will have least value of exponent among all nodes and exponent keep on increasing as we go further.

Lets define *active polynomials* as the part of a polynomial that is yet to be summed up. The first check is to determine which exponent among the already existing *active* polynomials, is the least. The least is then appended to the end of the result so that the result is consistent

to the idea of having exponents in ascending order.

In the event that the least exponent in both the polynomials are equal, their corresponding coefficients is summed and then appended. In the event that one of the polynomial is exhausted, we simple keep on appending nodes of other polynomial to result.

### 4.3 Complexity

Let's assume there are  $m$  and  $n$  terms in the two polynomials to be added. There are three loops one after the other in the pseudo code. The worst case complexity of first loop is  $O(m + n - k)$  where  $k$  is the number of elements which will be added in the individual second or third loop of the pseudo code (extra terms of one of the polynomials after other is exhausted). Only one of second or third loop would run if one of the polynomials has been appended fully. The overall worst case for all three loops together is when **all ( $m+n$ )** terms need to be appended separately. Therefore, complexity of addition algorithm is  $O(m + n)$

Final complexity of the algorithm is  $O(m + n)$ , which is to be expected and is also indicated in the problem statement.

## 5 Polynomial Multiplication Algorithm

Algorithm for Polynomial multiplication is mentioned in this section. During this we will be provided with two proper polynomials, sorted in ascending order of exponents. The idea is to get a proper polynomial which is product of two polynomials.

### 5.1 Pseudo code

*head* is the sentinel node of polynomial.

```
// multiply two polynomials
void multiply(Polynomial p1, Polynomial p2)
{
    // loop through and multiply each
    // term of p1 with each term of p2
    // as p.head points to sentinel node
    node *temp1 = p1.head->next;
    node *temp2 = p2.head->next;
    while (temp1 != p1.head)
    {
        node *temp = head->next;
        // temp keeps track of previous update in result

        while (temp2 != p2.head)
        {
            // temp keeps track of previous update in result
            // keep updating nodes as required
            temp = updateNode(temp1->coeff * temp2->coeff ,
                               temp1->exp + temp2->exp , temp);
            temp2 = temp2->next;
        }
        // start p2 from the beginning
        temp2 = p2.head->next;
        temp1 = temp1->next;
    }
}
```



## 5.2 Helper Function for multiplication

*updateNode* is a helper function for multiplication which inserts or modifies appropriate nodes of the result *Polynomial*. It returns pointer to the next node of current update.

```
// Update the coefficient of existing node with the given
// exponent or create a new node if not found
node *updateNode(int coeff, int exp, node *curr)
{
    // loop till you add it somewhere
    while (true)
    {
        //curr is pointer to next node from previous update
        //if curr == head, it implies
        // the new exponent is greater than the max exponent of
        // result and append this node at end of result
        if (curr == head)
        {
            appendNode(coeff, exp);
            return head;
        }
        else
        {
            // if the incoming exponent is equal to the current
            // node's exponent then update the coefficient
            if (curr->exp == exp)
            {
                curr->coeff += coeff;

                // if this makes coefficient to 0
                // then remove the node
                if (curr->coeff == 0) deleteNode(curr);
                return nextNode;
            }
            // if the incoming exponent is greater than the curr
            // node's exponent then move to the next node
            // as this is neither right place to insert
```

```

// or to update coefficient
else if (curr->exp < exp) curr = curr->next;
else if (curr->exp > exp)
{
    // the incoming exponent is less than curr
    // node's exponent then insert the node
    // before the curr node
    node *newNode = createNode(coeff, exp);

    // Fix its links
    newNode->next = curr;
    newNode->prev = prev;
    prev->next = newNode;
    curr->prev = newNode;

    // as curr could still be updated
    return curr;
}
}
}
}

```

### 5.3 Mathematical completeness

The idea of this too is simple. First keep multiplying each of the terms of 1st polynomial with each of 2nd polynomial and keep updating the result *Polynomial*. We must observe that while looping through the inner polynomial, we just need to remember at which place the value from last iteration was inserted, as the polynomials are already sorted in ascending order, so all further iterations of inner loop have higher exponent terms than previous iterations. After the inner loop completes, we reset the pointer of polynomial 2 to the first term and run the next iteration of outer loop. The Resultant *Polynomial* will also be a *proper Polynomial* with ascending order of coefficients.

The *updateNode* function is will Modify the coefficient of the node with the given exponent if it exists else it will create a new node and insert the node at appropriate place.

## 5.4 Complexity

The outer loop iterates through all the terms of polynomial p1. The inner loop multiplies each term of p2 with the term from outer loop, and updates the result. The *updateNode()* function returns the node next to previous modification from previous iteration of inner loop and keeps track of it. We can consider a single iteration of inner loop and *updateNode()* as a bloc, because each iteration of inner loop generates a higher exponent than previous iteration and as *updateNode()* keeps track of previous modification in result *Polynomial*. So, in complete execution of inner loop, the *updateNode()* function goes over the entire resultant *Polynomial* only *once*.

Say, n, m are the number of terms in  $p(x), q(x)$  respectively.

To put it simply, the code looks like this:

```
loop (Polynomial p1)
    loop (Polynomial p2)
        updateNode (...)
```

This makes the complexity to be  $O(nm) * O(updateNode)$ . But complexity of *updateNode()* not constant and dependent on the *pointer(curr)* argument. Now, let's evaluate this in a different way. Let's consider the inner loop as one block instead of a loop and see the same algorithm in a different way.

```
loop (Polynomial p1)
    // multiply current iteration term of p1 with 1st term of p2
    node = multiply()
    // search where in result polynomial node can be inserted
    insert(node)

    [... so on m times ...]


    // multiply current iteration term of p1 with mth term of p2
    node = multiply()
    // search where in result polynomial node can be inserted
    insert(node)
```

Here, clearly, the complexity is  $O(n) * (O(m) + O(size(resultantPoly)))$ .

The size of Result *Polynomial* is also problem dependent. But the worse case scenario, the maximum size of result polynomial will be  $m * n$ .

Hence, Complexity becomes  $O(n) * (O(m) + O(mn))$  which can also be written as  $O(mn^2)$ .  
Finally, **Complexity of this multiplication algorithm** is  $O(mn^2)$ .

## 6 Footnotes

- Entire code of this assignment could be found here. 
- The programming was done in C++ by defining a self defined class *Polynomial*.
- Necessary hyperlinks have been added at appropriate places.