ESO207A - Assignment 1

Group members:
Arnav Gupta(200186)
Kushagra Sharma(200539)

Q1: Adding two polynomials

Problem analysis

We are given two polynomials with terms in the increasing order of their exponents. Now, we want to construct the polynomial which is equal to their sum, while maintaining the exponents in the correct order.

We notice that all the terms in the required polynomial would be having the same exponent in a term in at least one of the polynomials i.e. no terms with “new” exponents would be created. Hence, it makes sense to take all the terms in both polynomials and simply store their contribution for each exponent in a linked list. However, we should also ensure that the terms in the resultant linked list are sorted as per their exponents.

To maintain this order, at each step we must take the term in both polynomials which has the lowest exponent among the ones not visited and then make a new node for that exponent. This can be done with the two-pointers technique since the input guarantees us that the exponents are given in increasing order.

Hence, we take two pointers ptr1 and ptr2, one for each polynomial, pointing to the first term of each polynomial. At each step, we take the pointer which has the lower exponent field, create a node for that, and advance that pointer. If both have the same exponent, we create a node and simply advance both pointers. This helps us in always choosing the term having the least exponent among the terms not visited. Hence, all the terms in our final polynomial will become sorted according to their exponent.

Pseudocode

We shall use some concepts of the C++ language in the pseudocode. Also, here are some function definitions that will be used -

1. **struct node**; //we define the node as having 4 attributes: exponent, coefficient, and the pointers to the next and previous node respectively.

2. Function **make_sentinel**(head); //make head the sentinel node.

3. Function **insert**(coeff,exp,head); //this will insert a new node **after** the current node and **before** the head(or the sentinel node).

4. Function **init_list**(head); //to take in the input polynomial and construct the linked list for it.

5. Function **print_list**(head); //to print the linked list, given the starting pointer

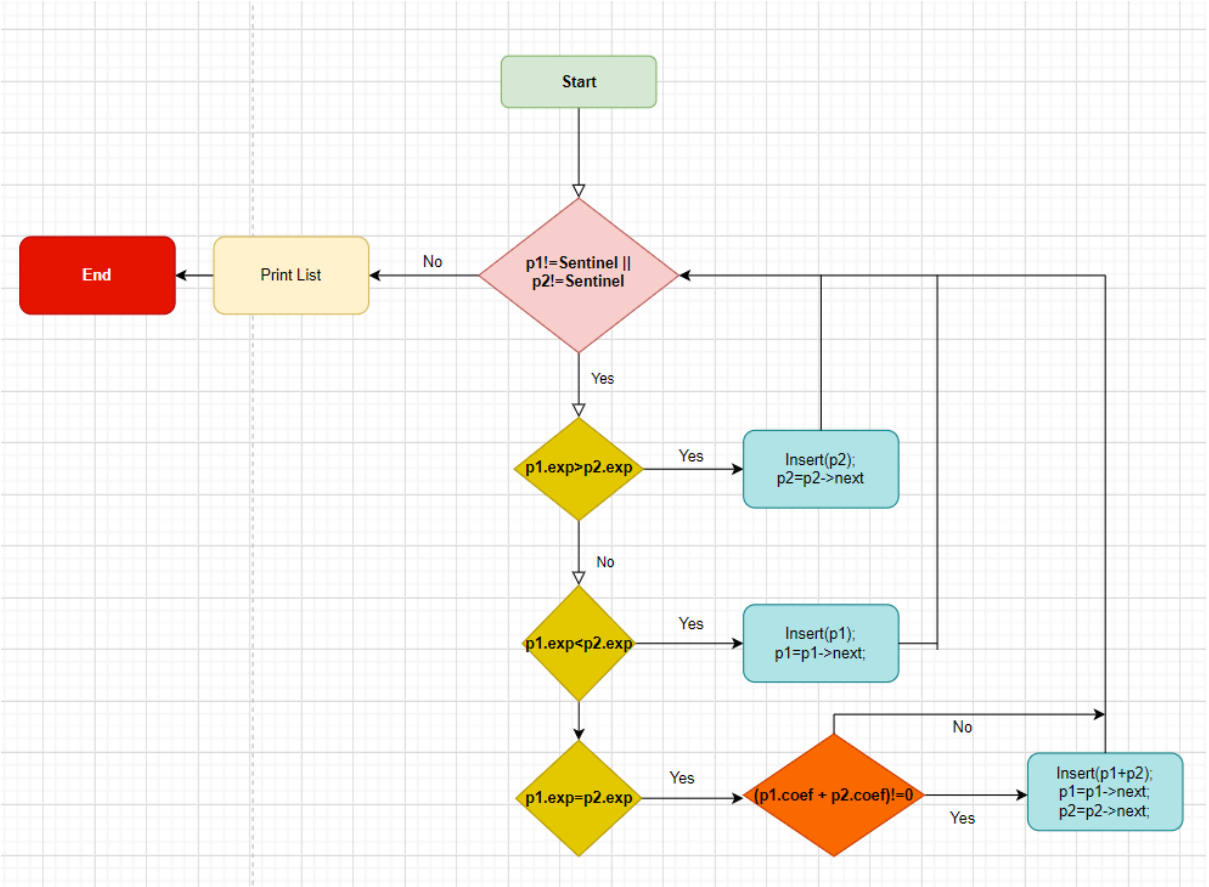
6. Function **allocate**(head,size_of(node)); //to allocate memory for new list.

Using the function definitions provided above, we write the pseudocode.

```
struct node
allocate(head1, size_of(node))
allocate(head2, size_of(node))

make_sentinel(head1)
make_sentinel(head2)

init_list(head1)          //taking the input
init_list(head2)

allocate(head3,size_of(node))
make_sentinel(head3)

pointer ptr1 = head1->next
pointer ptr2 = head2->next


// Addition Pseudocode.
while(ptr1 ≠ head1 or ptr2 ≠ head2) do:
        if(ptr1.exponent<ptr2.exponent) do:      //this is done to maintain sorted order of exp
                insert(ptr1.coeff,ptr1.exp,head3)
                ptr1 = ptr1.next
        else if(ptr2.exponent<ptr1.exponent) do:
                insert(ptr2.coeff,ptr2.exp,head3)
                ptr2 = ptr2.next
        else do:
                if(ptr1.coefficient+ptr2.coefficient ≠ 0) do:
                        insert(ptr1.coeff + ptr2.coeff,ptr1.exp,head3)
                ptr1 = ptr1.next
                ptr2 = ptr2.next

print_list(head3)
```

Complexity Analysis

The following flow-chart gives the flow of execution of the addition pseudocode:
(The sentinel node's exponents are kept above the given constraints of the polynomial exponents to allow smooth functioning of the loop).



In the algorithm described above, the two pointers(i.e., the running variables) start from the first term of their respective polynomial and proceed till they return to the sentinel node. If we consider the first pointer, it will iterate one by one through the n terms, after which it comes back to the sentinel node. Also, we notice that whatever be the case(if, else if or else), the steps described inside those blocks will take a constant time since insertion(and deletion) in a linked list takes a constant time.

Let the average of this constant value be c_1. Also, let the time it takes to check the condition in the while statement be c_2. Then, the time taken for the first pointer to get back to the sentinel node is $c_1*n + c_2*(n+1)$. It is easy to see that this is O(n).

Performing a similar analysis for the second pointer, we see that it will also be O(m). Since our program performs both these steps, the time complexity of our algorithm is **O(m+n)**.

Q2: Multiplying two polynomials

Problem analysis

We are given two polynomials with terms in the increasing order of their exponents. Now, we want to construct the polynomial which is equal to their product, while maintaining the exponents in the correct order.

Looking at the constraints, the best strategy is to iterate over all possible terms formed by multiplying the terms in the two polynomials and store them such that they are sorted by the exponents.

Pseudocode

We shall use some concepts of the C++ language in the pseudocode. Also, here are some function definitions that will be used -

1. **struct node**; //we define the node as having 4 attributes: exponent, coefficient, and the pointers to the next and previous node respectively.

2. Function **make_sentinel**(head); //make head the sentinel node.

3. Function **insert**(coeff,exp,ptr); //this will insert a new node **before** the node pointed by 'ptr'.

4. Function **init_list**(head); //to take in the input polynomial and construct the linked list for it.

5. Function **print_list**(head); //to print the linked list, given the starting pointer

6. Function **allocate**(head,size_of(node)); //to allocate memory for new list.

Using the function definitions provided above, we write the pseudocode.
Also, we set the exp attribute of the sentinel node to a very large value(say 10^{18}) to indicate the end of the polynomial.

```
struct node
allocate(head1, size_of(node))
allocate(head2, size_of(node))

make_sentinel(head1)
make_sentinel(head2)

init_list(head1)          //taking the input
init_list(head2)

allocate(head3,size_of(node))
make_sentinel(head3)

pointer ptr1 = head1->next
pointer ptr2
pointer ptr3


// Multiplication Pseudocode.
while(ptr1 ≠ head1){
        ptr2 = head2->next
        ptr3 = head3->next
        while(ptr2 ≠ head2){        //iterating through all the m*n terms generated by
multiplying the 2 polynomials
                if(ptr3=head3)
                        ptr3=head3.next
                p = (ptr1.exp)+(ptr2.exp)
                c = (ptr1.coeff)*(ptr2.coeff)   //coefficient of the exponent
                while(!((ptr3.next).exp>p))    //advancing ptr3 till we find the optimal location
for the term
                        ptr3=ptr3.next

                if(ptr3.exp==p)     //now if the term of the exponent has already been created
by a previous term, we update it.
                        ptr3.coeff+=c

                else
                        insert(c,p,ptr3.next);//else, we make a new node for that exponent just
                before the first node which has a larger exponent.
                ptr2=ptr2.next
        }
        ptr1=ptr1.next
}

print_list(head3)
```

Correctness

Say the size of the first and second lists(polynomials) is n and m respectively. We multiply all the terms of the second list with each element of the first list and keep adding the terms to a third list, the product polynomial.

We iterate through the lists using 3 pointers, say p1,p2,p3, pointing to the first, second, and third lists, respectively.

How the addition of new terms is carried out optimally:

Assuming the current length of the third list is **l**. And the terms are in the increasing order of their exponents.

The new exponent which we obtain after multiplying the two terms(p1 and p2): **New_exp**

We move p3 to the next term until we encounter a term with an exponent greater than **New_exp**.

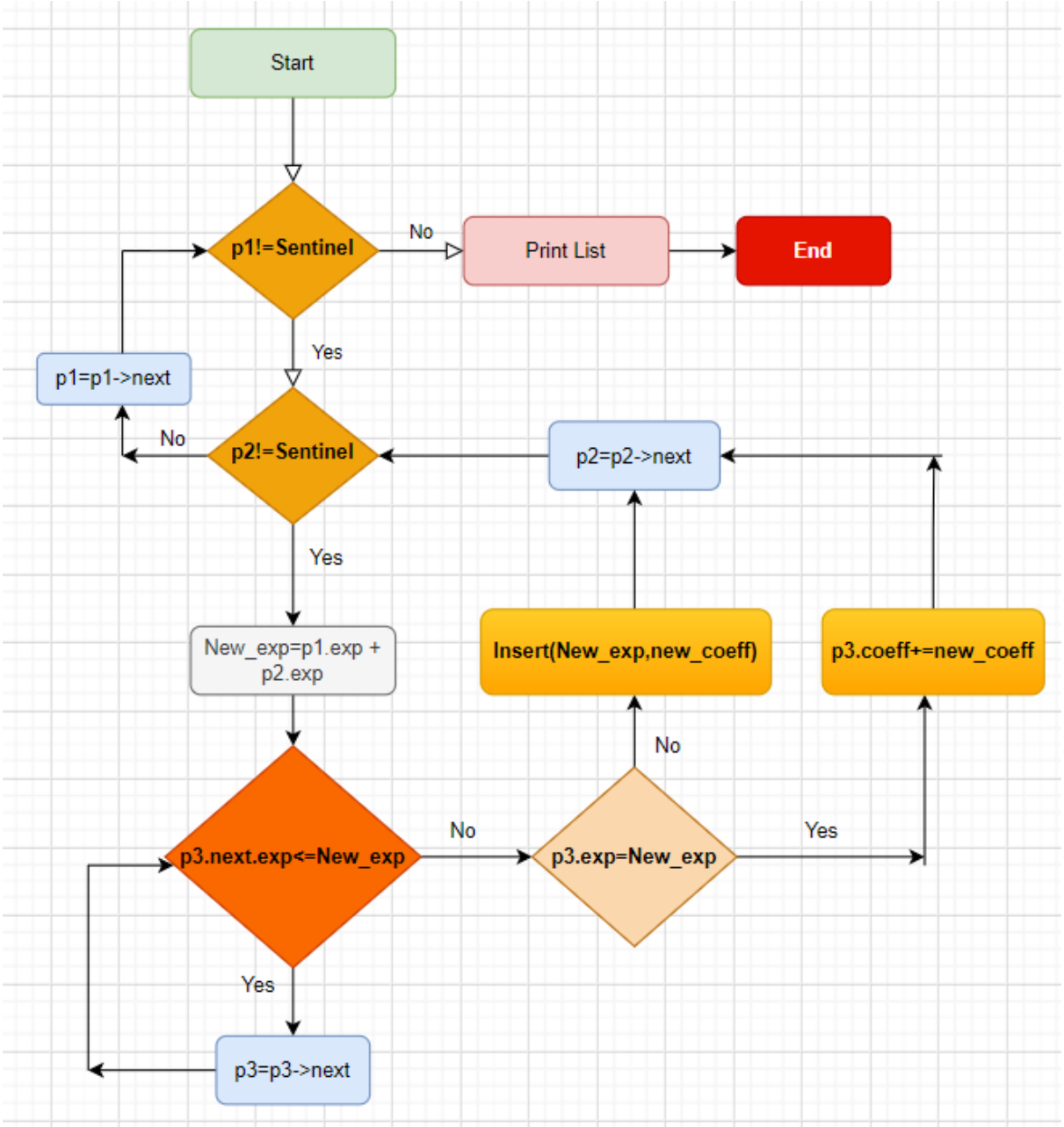
Seeing the current value of exponent(of p3), if it is equal to **New_exp**, we add the coefficients, or insert a new node.

This way, each term is added optimally so that the list p3 remains in increasing order as we keep inserting new nodes(terms) in it.

Note: Since all the terms of both the given lists are in ascending order of their exponent, p3 will only have to move in the forward direction.

For a fixed value of p1, as we iterate p2, the value of **New_exp** only increases (since the exponents of p2 increase). Thus p3 only has to move rightwards(or stay) over all the iterations of p2, for a fixed p1. In simpler words, for a fixed value of p1, both p2 and p3 traverse their respective lists only once.

The following flow-chart gives the flow of execution of the Multiplication pseudocode:



Complexity Analysis

For each iteration of the outer loop, our algorithm gets all terms of the second polynomial(m elements) and inserts/updates the node. As said earlier, the insertion/update of a node takes constant time.

Also, a key observation here is that since the exponents in both polynomials are sorted, we get a larger exponent each time as we progress with the inner(at 2nd indentation) while loop(fixing the term of the outer loop). To insert these nodes, we move in our third list in the forward direction. Hence, it is easy to see that we will traverse the entire linked list at most once. This is because if we were to traverse it more than once, that would mean that

we would have to pass the sentinel node, and the current resultant exp should be greater than 1e18(the value of exp we set in the sentinel). But that is guaranteed not to happen in the question.

So, for each iteration of the outer loop, the worst case is that we traverse the entire linked list. The complexity for this step is O(X), where X is the length of the list at that instant.

The length of the list in the worst case will be when all possible multiplications of terms in the 2 polynomials give us distinct exponents. So for the ith iteration of the outer loop, the maximum possible length of the resultant list is Σm from 1 to l, since m elements are created in each step. This is just m*l.

Now, X = m*l (X being mentioned in 3rd para). Now we just need to sum this up as l (denoting the iterations of the outer loop) goes from 1 to n.

So finally, the worst-case complexity is given by :

$$\sum_{l=1}^{(n)} (l) \cdot m \;=\; \frac{\big((n) \cdot (n-1)\big)}{2} \cdot m$$

Hence, the final time complexity is **$O(m*n^2)$**, or **$O(n*m^2)$**, depending on which of p(x) and q(x) is chosen for the outer loop. This complexity is when the polynomial having n elements is chosen for the outer loop.

A more general expression for the time complexity is O(m*n*x), where x is the number of terms in the polynomial chosen for the outer loop. Naturally, we would want the polynomial with fewer terms to be in the outer loop. However, in the question m and n have the same constraints, so in the average case, the choice of p(x) and q(x) for the outer loop doesn't influence the complexity.