ESO207 Programming Assignment 1

Devansh Kumar Jha(200318) and Divyansh Gupta(200351)

2021–08–30

1 Problem Statements

Polynomials may be represented as linked lists. Consider a polynomial

$$p(x) = a_1x^{e_1} + a_2x^{e_2} + a_3x^{e_3} + ... + a_{n-1}x^{e_{n-1}} + a_nx^{e_n}$$

with non-zero terms, $p(x)$ where $0 <= e_1 < e_2 < ... < e_{n-1} < e_n$ are (non-negative) integers. We assume that coefficients a1, . . . , an 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) **(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) and q(x) respectively.Implement your pseudo-code as an actual program.

(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) and 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).

2 Programming Problem 1

2.1 Problem Description

We are required to add the polynomials according to the given input where we are first given two integers n and m which are followed by 2n and 2m integers in the next two lines. The main complexities and boundary cases in the problem were to design a O(m+n) algorithm and to recognize that zero polynomials are the exceptional cases which do not directly fit into the algorithm and so are to be specifically handled. To take care of integer overflow we by default use big integers as our storage containers.

2.2 Data Structure Usage

We will be using a dynamically allocated doubly linked list with a sentinel node for easier and error free implementation.

2.3 Algorithm Explanation

The algorithm is quite simple. We just take two temprorary node pointers which will correspond to the current term of that particular polynomial in the addition process. There are 3 major cases as follows -

- **1st polynomial already traversed**
  In this case we just add the term of 2nd polynomial to the answer polynomials doubly linked list. Here we don't need to check anything as the question statement says that the coefficients will anyways be non-zero.
- **2nd polynomail already traversed**
  Similar to the case above with the difference that this time we will be traversing the 1st polynomial and adding new nodes to the resultant polynomial.
- **Both polynomials are being traversed**
  This is the most complex case of the mentioned three with some exceptions to be handled within. While we are simultaneously traversing the 1st and 2nd polynomial the currently checked term might have different variable exponents. In that case we cannot add them and we simply need to add one of them to the list. We chose to add the element with smaller exponent into the list as it ensures the automatic sorting of the resultant polynomial. In case of the exponents being equal we need to confirm that the coefficients don't add to zero. In case they do than we just skip these terms however if not than we add them. So it can easily be seen that in this case the iteration would be a bit more costly as compared 1st and 2nd case iteration.

2.4 Pseudo Code

These are the variables used -

- **HEAD** - The first element of addition polynomial.
- **HEAD1** - The first element of first polynomial.
- **HEAD2** - The first element of second polynomial.
- **SENT** - Sentinel node for addition polynomial.

** - Similar goes for TAIL,TAIL1,TAIL2,SENT1,SENT2

Algorithm 1: Polynomial Addition Algorithm

Input : The two linked lists showing p(x) and q(x) and pointers to make output polynomial r(x)

Output: No output

```
1  temp1 = HEAD1 and temp2 = HEAD2
2  /* Temprorary pointers to traverse along the available
      polynomials.                                          */
3  while temp1! = SENT1 or temp2! = SENT2 do
4      if temp1 == SENT1 then
5          TAIL.insert − node(temp2)    // insert-node() is a function
            which makes a new node and adds to the list
           temp2 ← (temp2 → next)
6      else
7          if temp2 == SENT2 then
8              TAIL.insert − node(temp1)
9              temp1 ← (temp1 → next)
10         else
11             if temp1 → exp == temp2 → exp then
12                 if temp1 → cof! = temp2 → cof then
13                     a=make-
                           node(temp1 → cof + temp2 → cof,temp1 → exp)
                       /* make-node() function allocates memory for a
                       new node and fills with the parameters given */
14                     TAIL.insert − node(a)
15                 else
16                 end if
17                 temp1 ← (temp1 → next)
18                 temp2 ← (temp2 → next)
19             else
20                 if temp1 → exp < temp2 → exp then
21                     TAIL.insert − node(temp1)
22                     temp1 ← (temp1 → next)
23                 else
24                     TAIL.insert − node(temp2)
25                     temp2 ← (temp2 → next)
26                 end if
27             end if
28         end if
29     end if
30 end while
31 return
```

3 Programming problem 2

3.1 Problem Description

We are required to multiply the polynomials according to the given input where we are first given two integers n and m which are followed by 2n and 2m integers in the next two lines. The main complexities and boundary cases in the problem were to design a O(mn) algorithm and to recognize that zero polynomials are the exceptional cases which do not directly fit into the algorithm and so are to be specifically handled. To take care of integer overflow we by default use big integers as our storage containers.

3.2 Data Structure Usage

We will be using a dynamically allocated doubly linked list with a sentinel node for easier and error free implementation.

3.3 Algorithm Explanation

The algorithm is quite simple. We first allocate nodes to the answer list starting from lowest exponent to highest exponent with coefficient=0. Then we take two temporary node pointers which will traverse through the polynomial and for each term of the first polynomial the 2nd pointer will traverse through the whole 2nd polynomial and hence the complexity O(mn).Though this is just a approximate analysis and a detailed analysis follows below.

3.4 Pseudo Code

These are the variables used -

- **HEAD** - The first element of answer polynomial.
- **HEAD1** - The first element of first polynomial.
- **HEAD2** - The first element of second polynomial.
- **SENT** - Sentinel node for answer polynomial.

****** - Similar goes for TAIL,TAIL1,TAIL2,SENT1,SENT2

Algorithm 2: Polynomial Multiplication Algorithm

Input : The two linked lists showing p(x) and q(x) and pointers to make output polynomial r(x)

Output: No output

```
1  less = HEAD1 → exp + HEAD2 → exp,more = TAIL1 → exp + TAIL2 → exp
2  for i ← less to more do
3  |  TAIL.insert − node(0,i)
4  end for
5  temp1 = HEAD1 // This is the temprorary node pointer which will help us to traverse through the 1st array
6  while temp1! = SENT1 do
7  |  temp2 = HEAD2, temp = HEAD
8  |  /* These are the temprorary node pointers to traverse through the 2nd polynomial and the resultant polynomial */
9  |  while temp2! = SENT2 do
10 |  |  find = temp1 → exp + temp2 → exp
11 |  |  while temp! = SENT and temp → exp! = find do
12 |  |  |  temp ← (temp → next) /* We are trying to find the appropriate place for the product to be added in the resultant. */
13 |  |  end while
14 |  |  (temp → cof) ← (temp → cof) + (temp1 → cof ∗ temp2 → cof)
   |  |   temp2 ← (temp2 → next)
15 |  end while
16 |  temp1 ← (temp1 → next)
17 end while
18 temp = HEAD
19 /* Now we will adjust the HEAD and TAIL of resultant polynomial and than remove all nodes containing zero values */
20 if HEAD → cof == 0 then
21 |  HEAD ← (HEAD → next)
22 end if
23 if TAIL → cof == 0 then
24 |  TAIL ← (TAIL → prev)
25 end if
26 while temp! = SENT do
27 |  if temp → cof == 0 then
28 |  |  remove(temp)    // remove() is just a series of 3 to 4 steps in which the particular pointed element is deleted
29 |  else
30 |  end if
31 |  temp ← (temp → next)
32 end while
33 return
```

3.5 Runtime Analysis

The exact runtime analysis would be quite difficult in the algorithm we have implemented so rather we would do some approximate runtime analysis of the code. So we take the following quite logical and practically appropriate assumptions -

- **The length of empty list created** is approximately equal to **m+n**. This basically implies that there are no large gaps in the exponents provided to us and they stand all nearly at some distance to each other. There might be some terms which contribute to a single exponent also so the value of actual length can be sometimes less than (m+n) also.
- **The cost of each iteration of the algorithm while calculating the result** is approximately equal to each other except for the innermost loop which tries to find the exponent where the values are to be added in empty list.

So with these assumptions we can conclude that -

- **The first for loop**
  Will have a total of (m+n) iterations with some constant time owing to total
  $time = c_1(m+n)$
- **The main nested loop set**
  Will have a outer loop running n times and a inner loop running m times. The cost of each iteration of the inner loop will be $time_{ji} = c_2 + c_3 x_i$ where $[x_i]$ for $1 <= i <= m$ would be the number of iteration of the tiny loop iterating over the resultant polynomial. Here j is the index for outer loop $1 <= j <= n$.
  It could be easily seen that $\sum_{i=1}^{m} x_i = (m+n)$ thus the total cost of all iterations i.e. $time_j = \sum_{i=1}^{m} time_i = c_2 m + c_3(m+n)$. Adding over all the times we get
  $time = c_4 mn + c_5 n^2$
- **Zero clearing for loop**
  This also iterates over the (m+n) elements and delete the nodes with coefficient as zero.So
  $time = c_6(m+n)$

Totalling all the costs of all steps we get

$$runtime = c^{'}(m+n) + c^{''}mn + c^{'''}n^2 \tag{1}$$

$$runtime <= cmn \tag{2}$$

For some sufficiently large value of constant we can say that the terms mn and n^2 are of the same order and thus we choose convert it in form of mn as it is a more useful notation.
So the algorithm is having a time complexity of O(m*n)