Question 2

Divyansh Gupta

2021–08–30

1 Programming problem 2

1.1 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.

1.2 Data Structure Usage

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

1.3 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.

1.4 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

1.5 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

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

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

Algorithm 1: List Making Algorithm

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

Output: No output

```
less = HEAD1 → exp + HEAD2 → exp,more = TAIL1 → exp + TAIL2 → exp
for i ← less to more do
    TAIL.insert − node(0,i)
end for
temp1 = HEAD1 // This is the temprorary node pointer which will help us to traverse through the 1st array
while temp1! = SENT1 do
    temp2 = HEAD2, temp = HEAD
    /* These are the temprorary node pointers to traverse through the 2nd polynomial and the resultant polynomial */
    while temp2! = SENT2 do
        find = temp1 → exp + temp2 → exp
        while temp! = SENT and temp → exp! = find do
            temp ← (temp → next) /* We are trying to find the appropriate place for the product to be added in the resultant. */
        end while
        (temp → cof) ← (temp → cof) + (temp1 → cof ∗ temp2 → cof)
        temp2 ← (temp2 → next)
    end while
    temp1 ← (temp1 → next)
end while
temp = HEAD
/* Now we will adjust the HEAD and TAIL of resultant polynomial and than remove all nodes containing zero values */
if HEAD → cof == 0 then
    HEAD ← (HEAD → next)
end if
if TAIL → cof == 0 then
    TAIL ← (TAIL → prev)
end if
while temp! = SENT do
    if temp → cof == 0 then
        remove(temp)    // remove() is just a series of 3 to 4 steps in which the particular pointed element is deleted
    else
    end if
    temp ← (temp → next)
end while
return
```

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)