ESO207 Programming Assignment-1

Suyash Srivastava (201031) and Mohan Krishna (200590)

Hackerrank ID: sumokrisri

31^{st} August, 2021

1 Adding two polynomials in O(n+m)

Input: Two polynomials with exponents in increasing order, and non-zero integer coefficients.

Output: A single polynomial in the same format, which is the sum of the two input polynomials.

The polynomials are stored in a doubly linked list with a sentinel node.

Each node has two node pointer fields (next and prev) and two integer fields (coefficient and exponent).

The algorithm for adding is similiar to the method for merging two sorting linked lists together. Here the polynomial nodes are sorted with respect to the exponent field. We add the polynomials in the following manner:

1. We use two pointers A and B, one for each list. Initially each points to the first node after the sentinel node of the respective lists P and Q.
2. If both A and B point to the NIL node of their fields, we
   (a) remove the nodes with zero coefficients in the sum list
   (b) return the sum list and terminate the function execution
3. (a) If the exponents are the same we add the coefficients and store this in a new node at the end of the sum list. We move A and B each by one step along their list.
   (b) Else we store the coefficient of the pointer with the smaller exponent in a new node at the end of the sum list Here we then move only the pointer with the smaller exponent by one step along its list.
4. Go to Step 2.

Note that in the case when one of the pointer returns to the NIL node, the program should continue to insert the remaining nodes of the other list. This

is done by setting the exponent field of the sentinel node of each node to be ∞, which ensures 3(b) to be executed in such cases.

1.1 Pseudo Code:

```
createNode(CoeffVal, ExpVal)
  node.coefficient = CoeffVal
  node.exponent = ExpVal
  node.next = node
  node.prev = node
  return node

createList()
  list.nil = (0, ∞)
  return list

append (list, node)
  node.next = list.next
  node.prev = list.prev
  list.nil.prev.next = node
  list.nil.prev = node
  return list

delete (node)
  node.prev.next=node.next
  node.next.prev=node.prev
  free (node)


removeZeroes (list)
  nodex = list.nil.next
  while (nodex != list.nil)
    na = nodex.next
    if (nodex.coefficient == 0)
      delete nodex
    nodex = na
  return list

printList (list)
  for (nodex = list.nil.next; nodex != list.nil;  nodex = nodex.next)
  print (nodex.coefficient, nodex.exponent)
```

```
add (a, b)
  s = createList()
  na = a.nil.next
  nb = b.nil.next
  while ((na != a.nil) or (nb != b.nil))
    if (na.exponent == nb.exponent)
      s = append(s,createNode(na.coefficient+nb.coefficient ,na.exponent))
       if(na!=a.nil)na = na.next
       if(nb!=b.nil)nb = nb.next
    else if (na.exponent > nb.exponent)
      s = append (s, createNode (nb.coefficient , nb.exponent))
      nb = nb.next
    else
      s = append (s, createNode (na.coefficient , na.exponent))
      na = na.next
  return s

inputlist (len)
  a = createList()
  na = a.nil
  for (i = 0; i < len; i ++)
    c = input()
    e = input()
    a = append (a, createNode (c, e))
  return a

main()
  plen = input()
  qlen = input()
  p = inputlist (plen)
  q = inputlist (qlen)
  printList(removeZeroes(add(p,q)))
```

1.2 Runtime Analysis

- It is clear from the PseudoCodes that the functions **createNode, createList, append** and **delete** take $O(1)$ time
- **removeZeroes** :
  Here again we are traversing only once through the entire list and are only executing constant time statements on each iteration, thus **removeZeroes** is $O(n)$
- Similiarly we can see that **printList** is also $O(n)$
- **add** :
  In this algorithm we traverse through both the lists simultaneously and on each iteration we advance through one node in atleast one of the lists. The loop is exitted once we have traversed through both the lists. This will take atmost $n + m$ iterations of the while loop, while all the other statements take $O(1)$ time.
  Thus the function takes $O(n + m)$ time

2 Multiplying two polynomials:

Input: Two polynomials with exponents in increasing order, and non-zero integer coefficients.

Output: A single polynomial in the same format, which is the product of the two input polynomials.

We multiply the two polynomials and store the result in a product polynomial, and then return it, using the following algorithm.

1. We use two pointers a and b, one for each inputted list. (a for list A, and b for list B). Initially each points to the first node after the sentinel nodes of their respective lists, and P and Q are two empty lists (with only the sentinel nodes).

2. If A points to the NIL Node of its field, we return the product list P and end program execution

3. We call an auxilliary function which traverses through the nodes of B. For each node b: (C_b, E_b) in B, it creates a node n_q: ($C_b * C_a$, $E_b + E_a$) and stores it at the end of a temporary list Q.

4. We add the product list P and temporary list Q using the addition function defined in Section 1..

5. We now set A to point to the next node of its list, empty the temporary list Q, and go to Step 2.

The idea is that $p(x)q(x) = (a_1x^{e_1} + p_1(x))(q(x)) = a_1x^{e_1}q(x) + p_1(x)q(x)$, where $p_1(x)$ has strictly lesser terms than $p(x)$. If $p(x)$ has no terms, i.e. $p(x) = 0$, then simply $p(x)q(x) = 0$

2.1 Pseudo Code:

```
createNode (coeffVal, exponentVal)
  node.coefficient = coeffVal
  node.exponent = exponentVal
  node.next = node
  node.prev = node
  return node

createList()
  list.nil = createNode(0, ∞)
  return list
```

```
append (list, node)
  node.next = list.next
  node.prev = list.prev
  list.nil.prev.next = node
  list.nil.prev = node
  return list

delete (node)
  node.prev.next=node.next
  node.next.prev=node.prev
  free (node)

empty (list)
  for (nodeX=list.nil.next; nodeX !=list.nil; nodeX=list.nil.next)
    delete(nodeX)
  return list

removeZeroes (list)
  nodeX = list.nil.next
  while (nodeX != list.nil)
    na = nodeX.next
    if (nodeX.coefficient == 0)
      delete (nodeX)
    nodeX = na
  return list

printList (list)
  for (nodeX =list.nil.next;nodeX!=list.nil; nodeX = nodeX.next)
    print (nodeX.coefficient , nodeX.exponent)

add (a, b)
  s = createList()
  na = a.nil.next
  nb = b.nil.next
  while ((na != a.nil) or (nb != b.nil))
    if (na.exponent == nb.exponent)
      s = append(s, createNode(na.coefficient+nb.coefficient ,na.exponent))
      if(na!=a.nil)na = na.next
      if(nb!=b.nil)nb = nb.next
    else if (na.exponent > nb.exponent)
      s = append (s, createNode (nb.coefficient , nb.exponent))
      nb = nb.next
    else
      s = append (s, createNode (na.coefficient , na.exponent))
      na = na.next
```

```
  return s

MultiplyConst (b, na)
  p1 = createList()
  nb = b.nil
  for (nb = b.nil.next;nb!= b.nil;nb=nb.next)
    p1 = append
    (p1,createNode (nb.coefficient*na.coefficient , nb.exponent+na.exponent))
  return p1

multiplyList (a, b)
  p = createList()
  for (na = a.nil.next; na != a.nil; na = na.next)
    q = MultiplyConst (b, na)
    p = add (p, q)
    empty (q)
  return p

InputList (len)
  a = createList()
  na = a.nil
  for (i = 0; i < len; i ++)
    c = input()
    e = input()
    a = append (a, createNode (c, e))
  return a

main()
  plen = input()
  qlen = input()
  p = InputList (plen)
  q = InputList (qlen)
  printList(removeZeroes(multiplyList(p,q)))
```

2.2 Run-time Complexity Analysis

empty :
The statements **delete(nodex), nodex = list.nil.next, nodex!= list.nil, nodex = list.nil.next** all take constant time on each execution and each is executed $\leq n$ times where m is the number of nodes in the list.
Thus **empty** is $O(n)$.

multiplyList:
We shall assume that the input lists are of sizes m and n respectively. In our algorithm, we first multiply second list (of length n) throughout with the first element of m, and store the result in a temporary list.

The multiplication by a constant is done n times (size of the list), so there are n primitive steps (up to a constant multiple). Then we add this temporary list with the sum list (which is initially empty, denoting the zero polynomial), and the time taken for addition of two lists is O(m+n); using our previous algorithm for adding two polynomials. We store this new partially-complete sum list and empty the temporary list.

Now, we multiply the **second** element of first list with the second list and store this result again in the temporary list. We then add this temporary list to the partially complete sum list.

This multiplication with constant also has n primitive steps up to a constant multiple. Addition also has (n+n) primitive steps up to a constant multiple.

Now, for the worst case complexity, we assume that the length of list increases by n, that is the new partially-complete list is of length 2n. So the next iteration will have n primitive steps(up to a constant multiple) for multiplication and n+2n steps(up to a constant multiple) for addition.

Similarly, the $\mathbf{x}^{th}$ iteration will have **(n+xn)** primitive steps(up to a constant multiple).

Empty function is executed in every iteration. In net there are m iterations, and in each iteration (as calculated above), the complexity of empty is O(n). Hence the sum of all primitive steps over all iterations that are executed by empty() is O(mn). So we get:

Runtime:

$[(c_1 n)+(c_2 n)]+[(c_1 n)+c_3(n+n)]+[(c_1 n)+c_4(n+2n)]+..+[(c_1 n)+c'(n+mn)]+c"(mn)+const.$
$\leq c\times n\times[2+3+4+..+(m+1)]+c"(mn)$
$=c\times n\times[\frac{m(m+3)}{2}]+c"mn$
$=O(m^2 n)$.

Hence, the runtime complexity of our algorithm is $O(m^2 n)$, where m and n are the sizes of our input lists.