CSE 373 Spring 1998
Assignment #2
Sample Solution

1. Suppose that our list type has nodes with the fields `coef`, `exp`, and `next`, and that polynomials are represented as lists which have been sorted in decreasing order of exponent. We also assume that each polynomial has at most one term with any given exponent value.

   (a) We write our solution in terms of three functions, `generate_terms`, `merge`, and `mergesort`. We assume the last function is given, but that it calls our `merge` function. Our `merge` function has the special feature that it removes duplicate exponent values as it merges the two lists. The function `generate_terms` simply generates all the terms appearing in the product of two polynomials, in no particular order and with terms with duplicate exponents allowed.

```
ListPtr generate_terms(ListPtr p1, ListPtr p2)
{
   ListPtr temp, result;

   while (p1 != null) {
     temp = p2;
     while (temp != null) {
       insert(result, new node(p1.coef * temp.coef,
                p1.exp + temp.exp));
       temp = temp.next;
     }
     p1 = p1.next;
   }
   return result;
}


/* Assumes that p1 and p2 are in decreasing order
   by exponent, and that p1 and p2 do *not* have
   duplicated exponents.  The result is a sorted
   list with no duplicated exponents. */
ListPtr merge(ListPtr p1, ListPtr p2)
```

```
{
   ListPtr result;

   if (p1.exp > p2.exp) then {
     result = p1;
     result.next = merge(p1.next, p2);
   }
   else if (p1.exp < p2.exp) then {
     result = p2;
     result.next = merge(p1, p2.next);
   }
   else /* p1.exp == p2.exp */ {
     result = new node(p1.coef + p2.coef, p1.exp);
     result.next = merge(p1.next, p2.next);
   }

   return result;
}


/* Finally, the polynomial multiplication algorithm
   is trivial ... */
ListPtr multiply_polynomials(ListPtr p1, ListPtr p2)
{
   ListPtr result;

   result = generate_terms(p1, p2);
   result = mergesort(result);  /* Remember, this
                                   removes duplicates too. */
   return result;
}
```

(b) The function `generate_terms` generates nm terms in nm steps. The sort on these elements then takes $\Theta(nm \log nm)$ time. By simple algebra, this is equivalent to $\Theta(nm \max(\log n, \log m))$ time. Note that there is a slightly more complicated algorithm that replaces the max with a min; this is not asymptotically better, but might be slightly better for some values of m and n.

2. $gfde * c + b * a + ++$.

3. We assume that we have a stack available with the common stack operations, push, pop, and "is empty". Then suppose that we have a string $s = s_0 s_1 \cdots s_n$ and we want to determine if it is a valid postfix expression. We read the string from left to right. For each s_i, if s is an operand, we push V, the symbol for "valid". If s is an operator, we pop two V's if possible (or return "error" if not), and replace them with a single V. We repeat until we have processed the entire string. At the end of this process, the stack must contain a single V and nothing else.

4. The key observation is that to go from n leaves to $n+1$, we "split" a leaf in the tree with n leaves, making it the root of a subtree with two leaves. This increases the number of leaves by 1, and the size of the tree by 2. Hence, the number of nodes in a tree with n leaves is $2n-1$ if $n \geq 1$, or 0 if n is 0.

5. We assume that each node has `data`, `left`, and `right` fields.

```
int Count(TreePtr r, int a, int b)
{
  if (r == null) then
    return 0;

  if (a <= r.data <= b) then {
    return 1 + Count(r.left, a, b) + Count(r.right, a, b);
  }
  else if (r.data < a) then
    return Count(r.right, a, b)
  else /* b < r.data b */
    return Count(r.left, a, b);
}
```

6. Here is a small example: