ECE242F (Fall 2002) Assignment 1: Polynomial arithmetic

Danny Heap
heap@cs.utoronto.ca

September 20, 2002

You may prefer the HTML version of this document, at
http://www.cs.utoronto.ca/~heap/Courses/242F02/A1/a1/a1.html.

Purpose

You will implement some algorithms to add, multiply, divide and evaluate polynomials over the integers. C constructs such as loops, conditionals, and structs will be useful.

Description

A polynomial of degree n over the integers has the form:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 x^0$$

...where $x^1 = x$ and $x^0 = 1$, and a_i (called coefficient i) is an integer, for $0 \leq i \leq n$. There is a special case for the **zero polynomial** which we will define has having degree -1 and no coefficients.

Adding polynomials

Suppose you'd like to add two polynomials: $a_n x^n + \cdots + a_1 x + a_0$, and $b_m x^m + \cdots + b_1 x + b_0$. Their sum has degree $k = \max(m, n)$:

$$c^k x^k + \cdots + c_1 x + c_0$$

...where $c_i = a_i + b_i$ (a_i is zero if $i > n$, and b_i is zero if $i > m$).

Implement the function `plus` specified in Polynomial.h. You need to check that none of your coefficients fall outside $\pm$`INT_MAX`, specified in `<limits.h>` (in which case you should return {`NULL, OVERFLOW`}). For a polynomial of degree n, your function should have complexity $O(n)$.

Multiplying polynomials

Suppose you'd like to multiply the two polynomials of the previous subsection: $a_n x^n + \cdots + a_1 x + a_0$, and $b_m x^m + \cdots + b_1 x + b_0$. Their product has degree $k = m + n$:

$$c_{m+n} x^{m+n} + \cdots + c_1 x + c_0$$

...where $c_i = a_0 b_i + a_1 b_{i-1} + \cdots + a_i b_0$, where a_j is zero unless $0 \leq j \leq n$, and similarly b_j is zero unless $0 \leq j \leq m$.

A special case occurs when either polynomial is the zero polynomial. In that case, the product is the zero polynomial.

Implement the function `mult` specified in Polynomial.h. You need to be sure that none of your coefficients, or intermediate calculations, fall outside ±`INT_MAX`, specified in `<limits.h>` (in which case you should return {`NULL, OVERFLOW`}). For polynomials of degree n and m (assuming that $n \geq m$), your function should have complexity $O(n^2)$.

Evaluating a polynomial

Suppose you want to know the value of $a_n x^n + \cdots + a_1 x + a_0$ when x has a particular value. Here is an example of a really **inefficient** way to evaluate the polynomial, once you've set x to a particular value (e.g. $x = 7$):

$$a_n * \underbrace{x * \cdots * x}_{n \text{ times}} + a_{n-1} * \underbrace{x * \cdots * x}_{(n-1) \text{ times}} + \cdots + a_1 * x + a_0.$$

The problem is that this method uses $n - 1 + n - 2 + \cdots + 1$ (for a total of $[(n-1)n]/2$) multiplications, and n additions. You can rewrite the polynomial, using Cramer's rule, as:

$$a_0 + x * (a_1 + x * (a_2 + \cdots + x * (a_n) \cdots))$$

This reduces the number of multiplications to n, and preserves the number of additions.

The zero polynomial evaluates to 0 for every x.

Implement the function `eval` specified in Polynomial.h. Be sure to check whether your result, or any of your intermediate results, falls outside ±`INT_MAX` (in which case you should return `INT_MAX`). For a polynomial of order n, your function should have complexity $O(n)$.

Dividing by a monic polynomial

Division of polynomials P_1 by P_2 should behave like division of integers, that is you would like to find quotient polynomial Q and remainder polynomial R such that:

$$P_1 = P_2 \times Q + R \text{ AND } \deg(R) < \deg(P_2).$$

This is **NOT** always possible when P_1, P_2, Q, and R must be polynomials over the integers. For example, what quotient and remainder would you suggest for $P_1 = x^2$, and $P_2 = 3x$?

In the special case where P_2 has leading (highest) coefficient either 1 or -1 (that is, P_2 is monic), and P_2 has degree no greater than P_1, then division is possible. Here's a recipe:

1. Set the remainder R initially equal to P_1, and the quotient Q initially equal to 0.
2. While the degree of R is no less than the degree of P_2 do the following steps:
   (a) Construct a monomial M (a polynomial with one term) m by raising x (or whatever variable you're using) to the exponent equal to the degree of R minus the degree of P_2, and then multiplying this power of x by the leading coefficient of R times the leading coefficient of P_2 (either 1 or -1).
   (b) Recalculate Q by adding M to it.
   (c) Recalculate R by subtracting $(M \times P_2)$ from it. This new remainder will have lower degree than the old one.

Implement the function `monDiv` specified in Polynomial.h. For P_1 or order n, your function should have complexity $O(n^3)$.

What to submit

Submit your implementations of `plus`, `mult`, `eval`, and `monDiv` (specified in Polynomial.h) in a single file named `Polynomial.c` (note both the spelling and upper/lower case). Polynomial.c must include Polynomial.h. Each function must have a function header explaining any non-obvious details of the algorithm and its implementation. Variables should be named so as to make their purpose obvious, and commented when this is not possible.

Your `Polynomial.c` must compile when it is located in the same directory as TestPolynomial.c, Polynomial.h, and makefile, and the commands in makefile are executed. Once you have successfully built `TestPolynomial`, you can test drive it by typing:

```
TestPolynomial < fourByThree.txt
```

...where you can replace fourByThree.txt with any file (in the same directory as `TestPolynomial.c`) having the following format:

```
n m
a0 a1 ... an
b0 b1 ... bm
c0 c1 ... c(m+n)
d0 d1 ... d(max(m,n))
x
eval1
y
eval2
q0 q1 ... q(n-m)
r0 r1 ... rk
```

...where the meaning of the cryptic variables is: n and m are the degrees of `poly1` and `poly2` in TestPolynomial.c, $a0 \ldots an$ are the coefficients of `poly1`, $b0 \ldots bn$ are the coefficients of `poly2`, $c0 \ldots c(m+n)$ are the coefficients of `mult(poly1, poly2)`, $d0 \ldots d\max(m,n)$ are the coefficients of `plus(poly1, poly2)`, x is some integer, and `eval1` is `poly1(x)`, and `eval2` is `poly2(y)`. The quotient's coefficients are `q0, q1, ..., q(n-m)`, and `r0, r1 ...rk` are the remainder's coefficients. `poly2` must be monic and of degree no greater than `poly1`.

Grading

Here is the distribution of points for this lab, which is worth 4% of your final mark:

Correctness, 50 points: The functions you implement in `Polynomial.c` will be tested in a manner similar to TestPolynomial.c. We'll look at special cases, such as the zero polynomial.

Modularity, 17 points: Your code should be well-organized with an eye to reducing repeated code and making the meaning clear.

Readability, 17 points: Comments should make your implementation clear, indent to highlight grouping of code, use meaningful variable and function names.

Efficiency, 17 points: Your implementation should be within the big-Oh constraints given.