CS 136 Fall 2008
Mark Giesbrecht

Assignment 7
Due Friday, November 28, 2008 at 2:30pm.

All questions are to be completed in C. All functions should be in the same file `poly.c`, with headers for all functions in the file `poly.h`. A file `poly_driver.c` with a `main()` function and appropriate tests should also be provided.

In high school and Math 135, you investigated the basic properties of polynomials in a variable x. In this assignment you will develop data structures and algorithms for computing with such polynomials, where the coefficients are from the finite field $\mathbb{Z}_p$, the integers modulo a prime p.

1. In Assignment 4 (Question 1b and 1d) you developed code for `addp`, `subp`, `mulp` and `invp` for arithmetic in $\mathbb{Z}_p$. Add code for these functions, and headers, to your files for this assignment. As before, we assume that $2 \leq p < 2^{15}$.

   You should implement the routines `eqp`, which returns 1 if its two `int` arguments represent equal values modulo p. Similarly, `zerop`, returns 1 if its `int` argument is 0 modulo p. These should have prototypes as follows.

   ```
   int eqp(int a, int b, int p);
   int zerop(int a, int p);
   ```

   Note that this question is as trivial as it sounds!

2. A polynomial $f(x)$ over $\mathbb{Z}_p$ is a formal mathematical object which can be written as

   $$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_n x^n,$$

   where n is a non-negative integer and $f_0, f_1, \ldots, f_n$ are in $\mathbb{Z}_p$ (where p is some prime number). We will assume that $f_n \neq 0$ and as usual call n the *degree* of $f(x)$.

   On a computer, we could represent a polynomial as a list of all its coefficients, which would be objects of class $\mathbb{Z}_p$ from above. For example, the polynomial $f(x) = 2x^3 + 10x^2 + 5$, with coefficients in $\mathbb{Z}_{17}$, might be represented be list (50102). The representation of a polynomial as a list of all its coefficients is called the *dense* representation of $f(x)$.

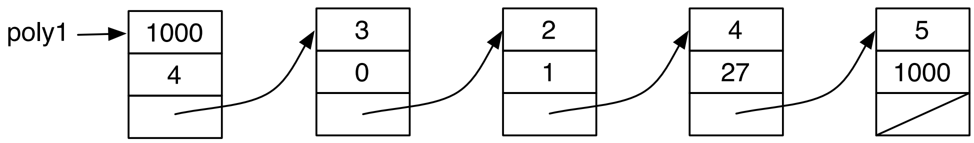
   Such a representation works acceptably when the degree of $f(x)$ is small, but a polynomial like $f(x) = 5x^{1000} + 4x^{27} + 2x + 3$ will be represented by a list of 1001 objects, which seems quite wasteful considering how compact the written form is. As an alternative, we can use a *sparse representation* of polynomials which consists of a list of `terms`, representing the powers of x which have non-zero coefficients. In C these terms could be represented with the following `struct`:

   ```
   struct term {
     int coeff;
     int expon;
     struct term *next;
   }
   ```

   A polynomial will be represented by a `struct poly`, which holds the degree of the polynomial, the number of non-zero terms, and a pointer term the first `term`:

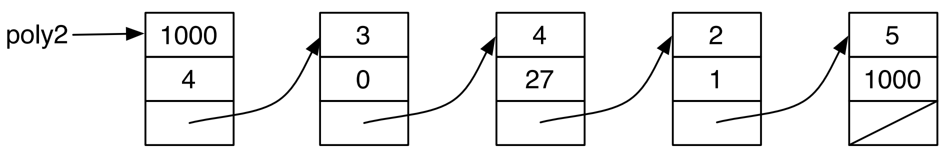
```
struct poly {
  int degree;
  int nzterms;
  struct term *pterms;
}
```

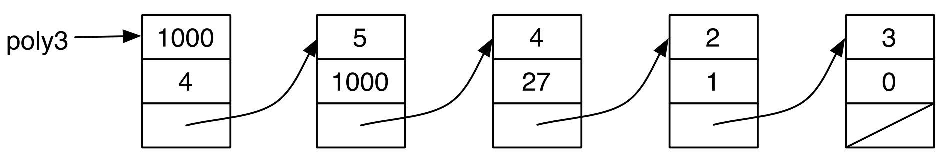
Thus, our polynomial would be represented as follows:



(where the leftmost box represents the `struct poly`, and the other boxes are `struct term`). You will first write functions which implement basic polynomial operations using the sparse representation.

3. It will be convenient that the sparse representation of a polynomial is a list of coefficient-exponent pairs in which the coefficients are non-zero and the exponents are unique and in increasing order. We call this the *canonical sparse representation* of the polynomial. Note that the canonical sparse representation is unique for any polynomial. For example, the polynomial $f(x)$ above can also be represented by





but we choose the first representation `poly1`, where the terms are increasing order of exponent, as the canonical sparse representation.

Write a function `poly_canonize` which consumes a `struct poly` and modifies its list of terms so that they are in increasing order of exponent. This will put the polynomial in canonical sparse representation. This function should have prototype:

```
struct poly poly_canonize(struct poly f, int p);
```

It produces the new, canonical sparse representation of `f`.

Hint: think about using a simple sorting algorithm that worked well on lists.

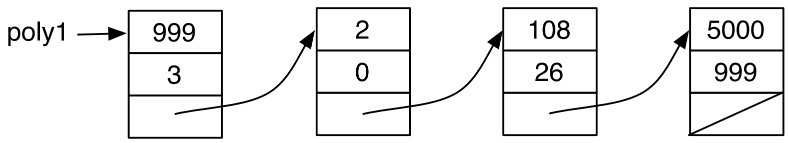
4. Write functions `poly_add`, `poly_subtract` and `poly_multiply` which consume the canonical sparse representation of polynomials $f(x)$ and $g(x)$ with coefficients in $\mathbb{Z}_p$ and produce the canonical sparse representation of $f(x) + g(x)$, $f(x) - g(x)$, $f(x)g(x)$. Make sure that when these functions are complete, the exponents of the result are in increasing order (and unique),

the coefficients are all non-zero, and the degree and number of coefficients are correctly recorded in the returned `struct poly`. Your functions should have prototypes:

```
struct poly poly_add(struct poly f, struct poly g, int p);
struct poly poly_sub(struct poly f, struct poly g, int p);
struct poly poly_mul(struct poly f, struct poly g, int p);
```

The polynomials `f` and `g` should *not* be modified during this operation.

5. Write a function `poly_diff` which consumes a polynomial as above, and a prime p, and produces its derivative. For example, `poly_diff(poly1)` would produce the following:



Your function should have prototype:

```
struct poly poly_diff(struct poly f, int p);
```

The polynomial past as a parameter should *not* be modified.

6. Write functions `poly_degree` and `poly_lcoeff` which consume a `struct poly` and return, respectively, the degree of the polynomial and the leading coefficient (the coefficient of the term of highest exponent). These should have prototypes as follows:

```
int poly_degree(struct poly f);
int poly_lcoeff(struct poly f);
```

Thus `poly_degree(poly1)` will return 1000, while `poly_lcoeff(poly1)` will return 5.

The following question is for up to 20% bonus, and is not required for full marks.

7. As you know from Math 135, given two polynomials $f(x)$ and $g(x)$, there exists a unique polynomial $r(x)$ with degree less than $g(x)$, and a polynomial $q(x)$, such that $f(x) = q(x)g(x) + r(x)$. The polynomials $q(x)$ and $r(x)$ are called the *quotient* and *remainder* respectively. The algorithm to compute the quotient and remainder has a surprisingly simple recursive definition. For the quotient:

   - If $\deg f(x) < \deg g(x)$, then the quotient is 0 and we are finished.
   - Otherwise assume $m = \deg f(x) - \deg g(x) \geq 0$. Let a and b be the leading coefficients of $f(x)$ and $g(x)$ respectively. Let $\widehat{f}(x) = f(x) - (a/b)x^m g(x)$. Then the quotient of $f(x)$ and $g(x)$ is $(a/b)x^m$ *plus* the quotient of $\widehat{f}(x)$ and $g(x)$.

   For the remainder:

   - If $\deg f(x) < \deg g(x)$, then the remainder is $f(x)$, and we are finished.
   - Otherwise assume $m = \deg f(x) - \deg g(x) \geq 0$. Let a and b be the leading coefficients of $f(x)$ and $g(x)$ respectively. Let $\widehat{f}(x) = f(x) - (a/b)x^m g(x)$. Then the remainder of $f(x)$ and $g(x)$ is the remainder of $\widehat{f}(x)$ and $g(x)$.

Note that in both cases the degree of $\widehat{f}(x)$ is less than that of $f(x)$ (try an example).

You are to write functions `poly_quo` and `poly_rem`, both of which consume two `struct poly`s and produce the quotient and remainder respectively. Your functions should have prototypes as follows:

```
struct poly poly_quo(struct poly f, struct poly g, int p);
struct poly poly_rem(struct poly f, struct poly g, int p);
```

The polynomials `f` and `g` should not be modified during these operations.