CS109 PROJECT: POLYNOMIALS; MEASURING RUNNING TIME
Due Friday, Feb. 28

The purpose of doing this lab is to gain some experience in measuring the running time of programs and to convince ourselves that the big-oh analysis is realistic, at least as input sizes get large. In a sense, it is like the first physics experiment where you measure the acceleration of gravity. We shall also gain some experience writing nontrivial ML programs.

The Setting

Let us represent the polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ (or $\sum_{i=0}^{n} a_ix^i$) by the list of coefficients $[a_0, a_1, a_2, \ldots, a_n]$. For example, the polynomial $2.5 - 3.7x + 5.2x^3$ would be represented by the list `[2.5, ~3.7, 0.0, 5.2]`. The list `[]` can be treated as an error (raise an exception), but you will probably find that it is easier to regard it as the polynomial 0, i.e., the same as the list `[0.0]`.

First Stage: Some Utilities

Write functions

1. `padd(P,Q)` takes polynomials (real lists) P and Q as arguments and produces the sum of P and Q. Note that P and Q may be of different degree (length, as a list).
2. `smult(P,a)` takes polynomial P and scalar (real number) a and produces the polynomial aP, that is, each term of P is multiplied by a.
3. `eval(P,a)` evaluates polynomial P with $x = a$; here a is a constant (real number). Note: we did something like this in class on the second day, but there we assumed that `[]` as a polynomial represented an error condition. Here, you may wish to regard `[]` as 0.0 and return 0.0 as a value, with no exception raised.
4. `gen1(n)` takes integer n and produces a list of n reals: `[1.0, 1.0,...,1.0]`.
5. `gen2(n)` takes integer n and produces the list of n reals: [n`.0`, $(n-1)$`.0`,`...`,`1.0`]. Note: `real(i)` produces the real number equal to the integer i.
6. `pmult(P,Q)` takes polynomials P and Q and produces the polynomial product PQ. We're going to discuss a simple program in class, which you may use. The idea is to break P into a head p and tail R. Multiply R by Q, then multiply by x and add pQ.

Important Aside

Multiplying polynomials by the method outlined in (6) seems natural and obvious, since $P = p + xR$, and therefore $PQ = (p + xR)Q = pQ + xRQ$. However, this algorithm takes $O(n^2)$ on polynomials of degree n, and there is an algorithm called the *Fast Fourier Transform* (FFT) that lets us multiply polynomials in time $O(n \log n)$. In this context, polynomial multiplication is usually called *convolution*.

Moreover, the FFT algorithm is far from an intellectual exercise, and multiplication of polynomials is not a game from High School algebra. Many signal processing problems like speech recognition, detection of radar signals, or computer-aided tomography ("CAT scans") depend upon the FFT or a similar algorithm for the speed that makes them feasible. In fact, this algorithm is so important, that special-purpose machines have been built to perform just that algorithm. (In general, implementing an algorithm in hardware will speed it up by a factor of 10 or more compared with a software implementation of the same algorithm running on a microprocessor.) However, we'll leave the FFT, which is a subtle divide-and-conquer type algorithm, for an advanced course in algorithms or signal processing.

Second Stage: Polynomial with Given Zeroes

Suppose we are given a list $(a_1, a_2, \ldots, a_n)$ of reals and we wish to find a polynomial that has these values as zeroes (repetitions are allowed). This polynomial is

$$(x - a_1)(x - a_2)\cdots(x - a_n)$$

so all we have to do is use `pmult` to multiply together the n degree-1 polynomials. There is a "dumb" way to do so, which we can suggest by the Pascal-like pseudo-code

```
P := (x-a1);
for i := 2 to n do
    P := P(x-ai)
```

There is also a "smart" way to form this polynomial, using divide-and-conquer. Recursively split the list of values into two half-sized lists, say L and M. Find two polynomials, one with zeroes at the elements of L and the other with zeroes at the elements of M. Then multiply these two polynomials together.

Third Phase: Interpolation of Polynomials

Rather than finding a polynomial from its zeroes, it is more common to want to find an nth-degree polynomial that goes through n given points (*interpolate* a polynomial). Suppose we are given n distinct x-values $(a_1, a_2, \ldots, a_n)$ and we are told that the desired polynomial P has values $(v_1, v_2, \ldots, v_n)$, respectively at these points. That is, $P(a_1) = v_1$, $P(a_2) = v_2$, and so on.

The trick is to find a polynomial P_i that has the value v_i at a_i and value 0 at all the other a_j's. If we sum all the P_i's, we get the polynomial P that has value v_i at a_i for $i = 1, 2, \ldots, n$.

We find P_i by starting with the product of the $(x - a_j)$'s, for all j between 1 and n, except for i. This product, which we shall call $Q_i(x)$, has value 0 at all points on the list except at a_i. If it had value v_i at a_i we'd be done, but in general it doesn't; it has value $Q_i(a_i)$. Thus, we must normalize by multiplying Q_i by $v_i/Q_i(a_i)$. That is the desired polynomial P_i. It, like Q_i, has value 0 at points on the list other than a_i, and at a_i it has value $P_i(a_i) = Q_i(a_i)v_i/Q_i(a_i) = v_i$.

This is all we're going to say about how to interpolate a polynomial. You need to work out an efficient algorithm for yourself, but you can use the function `pmult` from Phase 1 for polynomial multiplication, rather than trying to implement the FFT. There are still many opportunities to do things right and speed up the overall algorithm.

What to Hand In

You need to do two things in this lab problem. First, implement programs for the above operations in ML. Then, measure the running time of your algorithms, both experimentally and by doing a "big-oh" analysis.

1. (30%) Implement the "dumb" and "smart" algorithms described there, using the utilities from Phase 1. Most of these utilities will be written in class for you; write the others yourself.
2. (20%) Give a big-oh analysis of the dumb and smart algorithms.
3. (20%) Confirm your analysis experimentally by running your two algorithms on the lists produced by `gen1(n)` for various values of n. Values of n up to several hundred should result in a polynomial that has n zeroes at $x = 1$ with a few seconds of computation. Note: see how to measure running time of ML programs in the section below.
4. (20%) Implement an interpolation algorithm as described in Phase 3. Half credit will be given for any algorithm that works; the rest of the credit is allotted according to the efficiency of your algorithm.
5. (5%) Give a big-oh analysis of your interpolation algorithm.
6. (5%) Confirm your analysis in (5) experimentally. We suggest you use `gen2(n)` to generate both the points and the values.

Note that the bulk of the credit is given for Phase 2, and Phase 3 is in a sense "bonus."

Measuring Running Time

Suppose you have written an ML program on a file `foo`. You can execute it by typing

```
sml <foo
```

in response to the UNIX prompt. You can run the same program and get an analysis of the running time by typing instead

```
time sml <foo
```

After whatever output you get from the `sml` interpreter and your own program, you'll see a line like

```
0.6u 0.7s 0:04 31% 1128+1839k 30+1io 127pf+0w
```

The first element says you took 0.6 seconds of "user time," i.e., your program kept the processor busy for that long. The second element says that in addition, the processor spent 0.7 seconds handling the system calls (for disk operations, e.g.) required by your program. The third says that the total elapsed time was 4 seconds, including time the machine was handling somebody else's program. The fourth says that you got 31% of the processor while you were running. The remaining elements tell you about the space used by the program, and how many disk operations of various types were performed on your behalf. If you are curious, a complete explanation can be had by consulting the on-line UNIX manual; type

```
man time
```

In a sense, the appropriate measure of the running time of your program is the sum of user and system time, or 1.3 sec. in the above example. However, system time is often a function of the load on the machine, and can vary for different runs. (User time can vary too, but generally not by much.) In this experiment, let us take only the user time as the "running time."

When you run a program under `sml`, there will be some "overhead" that is not really part of the running time of your program. These costs include starting up the interpreter and preprocessing your program. We can get a good estimate of these costs if you run your programs on empty lists first (or lists of length 1 if you treat empty lists as exceptions), so true running time is approximately 0. You can then subtract the "overhead" from all measurements you take.

When you do large ML jobs, the system may have to pause to garbage collect. If you are not familiar with "garbage collection," a language like ML creates data, uses it, and then drops it from the active data, just as you might create a record in Pascal by `new` and then change the only pointer to that record to point somewhere else. ML, being a functional language, drops from use almost all of the results of functions eventually. To keep going, ML periodically has to run through its data area and find what is still needed, making the rest of the space available for new data (how it does so is a very interesting subject, but not for CS 109). This garbage collection takes a significant amount of time. Fortunately, ML prints out the amount of time it took for garbage collection, each time it does so. When you subtract these times from the user time that the `time` command gives, you get reasonable results.

Verifying Experimental Running Times

It may not be obvious how one determines if one has data that conforms to a particular upper bound like $O(n^2 \log n)$. That is, we are given a sequence of values of n: $n_1, n_2, \ldots, n_k$, and we are given corresponding measured running times, say $t_1, t_2, \ldots, t_k$. We plot them, and they look sort of proportional to $n^2 \log n$. But are they?

There is a theory of least-squares curve fitting that can help, but we won't go into that here. We suggest instead that you normalize the t_i's so that if there were some constant a such that t_i were exactly equal to $a(n_i)^2 \log_2 n_i$, then all the normalized values would be a. As we can generally eyeball data and tell whether it is a straight line, we should have a sound, although informal test for whether experiment fits theory. In our example, we could plot $t_i/(n_i)^2 \log_2(n_i)$ vs. n_i, for $i = 1, 2, \ldots, k$. Note that when there is a logarithm involved, picking n_i's that are powers of 2 makes life simpler.