CMPT 881 course project list: Spring 2013

Instructor: Michael Monagan
Due 5pm Wednesday April 24th.

Part I: Karatsuba's algorithm and the FFT (50 marks).

REFERENCE: Chapter 4 of the Geddes text.

Let $a, b \in \mathbb{Z}_p[x]$ be two polynomials of degree $n-1$. The classical algorithm uses n^2 multiplications in $\mathbb{Z}_p$ to multiply $a \times b$. Karatsuba's algorithm does $O(n^{1.585})$ multiplications. If p satisfies $2^{n+1} | p-1$ then we can use the FFT to multiply using $O(n \log n)$ multiplications.

The goal of this project is to implement the classical multiplication and either Karatsuba's algorithm for $\mathbb{Z}_p[x]$ or the FFT for $\mathbb{Z}_p[x]$ and compare them. Note, I think Karatsuba's algorithm is more difficult to implement than the FFT.

Note, to implement the algorithms in C or C++ or Java you should restrict the size of prime p such that $p^2 < 2^{63}$ on a 64 bit machine so you can multiply $x, y \in \mathbb{Z}_p$ using `x*y % p`. For the classical algorithm, most of the time is in divisions by p. You can speed up the classical algorithm a lot by reducing the number of divisions by p from $O(n^2)$ to $O(n)$.

For the FFT you need to compute inverses in the $\mathbb{Z}_p$. Use the extended Euclidean algorithm for this. After you get the basic FFT working, modify it to precompute the powers of ω that you need in an array so that you can reuse them in the FFT. This optimization doubles the speed of the FFT.

For Karatsuba's algorithm, use the classical algorithm at the base of the recursion for small degrees, i.e., pick a cutoff M such that if $\deg a < M$ or $\deg b < M$ you will use the classical algorithm. You will pick M to optimize the actual running time by experiment.

The main difficulty with implementing the fast algorithms is how you manage the storage for the intermediate polynomials. Since both algorithms are recursive, you could end up spending 90% of the total time allocating and deallocating memory if you are not careful. For both algorithms it is desirable to eliminate all storage management calls by allocating one piece of working storage W of size $O(n)$ at the beginning such that all intermediate data are stored in W. This is easy to do for the FFT but more difficult Karatsuba's algorithm.

Compare your Karatsuba or FFT code with the classical algorithm. Hence for suitable degrees n, say, $n = 2500, n = 5000, 10000, 20000, 40000, ...$ time both methods. You need sufficient data to see clearly when the fast algorithm beats the classical algorithm.

You will need to generate polynomials with random coefficients. Do this using a primitive element $\alpha \in \mathbb{Z}_p$ which you can obtain using Maple's `alpha := numtheory[primroot](a,p);` command which computes the first primitive element greater than a in $\mathbb{Z}_p$. Then use the powers of α modulo p for the random coefficients. I.e. use

```
for( x=1,i=0; i<n; i++ ) { x = alpha*x % p; a[i] = x; }
```

Hand in your code and test data and timing data.
Please tell me what was the most difficult part of this project.

Part 2: Newton's Method for Fast Division (20 marks)

Study the material in section 4.9 of the Geddes text. It explains how to use Newton's method to invert a power series $a(x)$ over a field F, i.e., to compute $a(x)^{-1}$ to $O(x^n)$. It explains, briefly, how to use this to divide fast.

(a) Reproduce the approximations in Example 4.12 by executing Newton's algorithm in Maple. If you want to program the iteration in a loop, that's fine.

(b) The presentation of Newton's method in Algorithm 4.6 FastNewtonIteration is not helpful when you need an $a(x)^{-1}$ to order $O(x^n)$ and n is not a power of 2. Modify Algorithm 4.6 to compute $a(x)^{-1}$ to $O(x^n)$ instead of $O(x^{2^n})$. The easiest way to do it is to compute the inverse recursively to $O(x^{\lceil n/2 \rceil})$. Implement your algorithm in Maple and compute $a(x)^{-1}$ to $O(x^{22})$ using the $a(x)$ in Example 4.12.

(c) Let $T(n)$ denote the number of operations in F the Newton iteration takes to compute $a(x)^{-1}$ to $O(x^n)$. On page 140 the text gives

$$T(2^{k+1}) = T(2^k) + cM(2^k)$$

where $M(2^k)$ is the cost of multiplying two polynomials of degree k. I'd prefer to substitute n for 2^{k+1} and rewrite it as

$$T(n) = T(n/2) + cM(n/2)$$

and assume n is a power of 2 i.e. $n = 2^k$. The analysis given, which assumes the FFT is used for multiplication, suggests that if you don't use the FFT then the cost is $O(n^2)$, but in fact we can do better. Assuming only that $M(n) \geq 2M(n/2)$, and using $T(1) = 1$ for the cost of inverting a_0, show that $T(n) \leq cM(n)$ thus $T(n) \in O(M(n))$. Hence conclude that we can compute the inverse as fast as we can multiply (up to a constant factor) no matter what multiplication algorithm is used.