Danny Sleator 15-451/651 Algorithms 12/03/14

Below you will find Avrim's notes from last year. After his notes I've included some notes of my own about applications, doing the integer FFT without the use of complex numbers, along with an ocaml implementation.

--DS

----------Avrim's Notes (start)----------

Avrim Blum 15-451/651 Algorithms 11/19/13

* Multiplying polynomials
* Fast Fourier Transform (FFT)

==========

FFT: The Fast Fourier Transform is a widely used algorithm in areas like signal processing, speech understanding, digital audio, radar.

We'll develop it in trying to solve the problem of "how to multiply quickly" we talked about on the first day of class. This is not how it was invented historically (and it obscures the connection to the usual kind of Fourier Transform), but I think it's most natural from the perspective of Algorithms. You don't need to know what a FT is for this lecture.

At the start of class we talked about the problem of multiplying big integers. Let's look at a simpler version of the problem, which is the problem of multiplying polynomials. It sounds more complicated, but really it's just the problem of multiplying integers without the carries.

MULTIPLYING POLYNOMIALS (aka COMPUTING CONVOLUTIONS)
==========
E.g., multiply (x^2 + 2x + 3)(2x^2 + 5) =

```
                      2x^2 + 0x + 5
                      1x^2 + 2x + 3
                      -------------
                      6      0   15
              4       0      10
      2       0       5
     ----------------------------
      2x^4 + 4x^3 + 11x^2 +10x+15
```

MODEL: view each individual small multiplication as a unit cost operation.

More generally, given $A(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$,
$B(x) = b_0 + b_1 x + ... + b_{n-1}x^{n-1}$

Our goal is to compute the polynomial $C(x) = A(x)*B(x)$.
$c_i = a_0*b_i + a_1*b_{i-1} + ... + a_i*b_0$.

If we think of A and B as vectors, then the C-vector is called the CONVOLUTION of A and B.

The FFT is a fast algorithm for computing this convolution. Given vectors $A = [a_0,...,a_{n-1}]$ and $B = [b_0,...,b_{n-1}]$ we want to compute $C = [c_0,...,c_{2n-2}]$ where c_i is defined as above.

- Straightforward computation is $O(n^2)$ time. Karatsuba is $n^{1.58..}$

- we'll use FFTs to do in $O(n \log n)$ time. This is then used in Schonhage-Strassen integer multiplication algorithm that multiplies

two n-bit integers in O(n log n loglog n) time. We're only going to do polynomial multiplication.

High Level Idea of Algorithm

Let m = 2n-1. [so degree of C is less than m]
1. Pick m points x_0, x_1, ..., x_{m-1} according to a secret formula.
2. Evaluate A at each of the points: A(x_0),..., A(x_{m-1}).
3. Same for B.
4. Now compute C(x_0),..., C(x_{m-1}), where C(x) is A(x)*B(x)
5. Interpolate to get the coefficients of C.

This approach is based on the fact that a polynomial of degree < m is uniquely specified by its value on m points. It seems patently crazy since it looks like steps 2 and 3 should take O(n^2) time just in themselves. However, the FFT will allow us to quickly move from "coefficient representation" of polynomial to the "value on m points" representation, and back, for our special set of m points. (Doesn't work for *arbitrary* m points.)

The reason we like this is that multiplying is easy in the "value on m points" representation. We just do: C(x_i) = A(x_i)*B(x_i). So, only O(m) time for step 4.

Let's focus on forward direction first. In that case, we've reduced our problem to the following:

GOAL: Given a polynomial A of degree < m, evaluate A at m points of our choosing in total time O(m log m). Assume m is a power of 2.

The FFT:

Let's first develop it through an example. Say m=8 so we have a polynomial

$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6x^6 + a_7x^7.$
(as a vector, A = [a_0, a_1, ..., a_7])

And we want to evaluate at eight points of our choosing. Here is an idea. Split A into two pieces, but instead of left and right, have them be even and odd. So, as vectors,
A_even = [a_0, a_2, a_4, a_6]
A_odd = [a_1, a_3, a_5, a_7]

or, as polynomials:

$A_{even}(y) = a_0 + a_2 y + a_4 y^2 + a_6 y^3$
$A_{odd}(y) = a_1 + a_3 y + a_5 y^2 + a_7 y^3.$

Each has degree < m/2. How can we write A(x) in terms of A_even and A_odd?

$A(x) = A_{even}(x^2) + x\, A_{odd}(x^2).$

What's nice is that the effort spent computing A(x) will give us A(-x) almost for free. So, let's say our special set of m points will have the property:

The 2nd half of points are the negative of the 1st half (*)

E.g., {1,2,3,4,-1,-2,-3,-4}.

Now, things look good: Let T(m) = time to evaluate a degree-m polynomial at our special set of m points. We're doing this by evaluating two degree-m/2 polynomials at m/2 points each (the squares), and then doing O(m) work to combine the results. This is great because the recurrence T(m) = 2T(m/2) + O(m) solves to O(m log m).

But, we're deluding ourselves by saying "just do it recursively".
Why is that? The problem is that recursively, our special points (now
{1, 4, 9, 16}) have to satisfy property (*). E.g., they should really
look like {1, 4, -1, -4}. BUT THESE ARE SQUARES!! How to fix? Just
use complex numbers! E.g., if these are the squares, what do the
original points look like?

        {1, 2, i, 2i, -1, -2, -i, -2i}

so then their squares are: 1, 4, -1, -4
and their squares are: 1, 16

But, at the next level we again need property (*). So, we want to
have {1, -1} there. This means we want the level before that to be
{1, i, -1, -i}, which is the same as {1, i, i^2, i^3}. So, for the
original level, let w = sqrt(i) = 0.707 + 0.707i, and then
our original set of points will be:

        1, w, w^2, w^3, w^4 (= -1), w^5 (= -w), w^6 (= -w^2), w^7 (= -w^3)

so that the squares are: 1, i, i^2 (= -1), i^3 (= -i)
and *their* squares are: 1, -1
and *their* squares are: 1

The "w" we are using is called a "primitive eighth root of unity"
(since w^8 = 1 and w^k != 1 for 0 < k < 8).

In general, a mth primitive root of unity is the number
w = cos(2*pi/m) + i*sin(2*pi/m)

Alternatively, we can use modular arithmetic: E.g., 2 is a primitive
8th root of unity mod 17.
{2^0,2^1,2^2,...,2^7} = {1,2,4,8 16,15,13, 9}
                      = {1,2,4,8,-1,-2,-4,-8}.

Then when you square them, you get {1,4,-1,-4}, etc.
This is nice because we don't need to deal with messy floating-points.

THE FFT ALGORITHM
=================
Here is the general algorithm in pseudo-C:

```
Let A be array of length m, w be primitive mth root of unity.
Goal: produce DFT F(A): evaluation of A at 1, w, w^2,...,w^{m-1}.
FFT(A, m, w)
{
  if (m==1) return vector (a_0)
  else {
    A_even = (a_0, a_2, ..., a_{m-2})
    A_odd  = (a_1, a_3, ..., a_{m-1})
    F_even = FFT(A_even, m/2, w^2)    //w^2 is a primitive m/2-th root of unity
    F_odd = FFT(A_odd, m/2, w^2)
    F = new vector of length m
    x = 1
    for (j=0; j < m/2; ++j) {
      F[j] = F_even[j] + x*F_odd[j]
      F[j+m/2] = F_even[j] - x*F_odd[j]
      x = x * w
  }
  return F
}
```

THE INVERSE OF THE FFT
======================
Remember, we started all this by saying that we were going to multiply
two polynomials A and B by evaluating each at a special set of m
points (which we can now do in time O(m log m)), then multiply the

values point-wise to get C evalauated at all these points (in O(m) time) but then we need to interpolate back to get the coefficients. In other words, we're doing F^{-1}(F(A) o F(B)).

So, we need to compute F^{-1}. Here's how.

First, we can look at the forward computation (computing A(x) at 1, w, w^2, ..., w^{m-1}) as an implicit matrix-vector product:

```
+-----------------------------------------+  +-----+    +-------+
|  1   1    1     1   ...    1           |  | a_0 |    | A(1)  |
|  1   w    w^2   w^3 ... w^{m-1}        |  | a_1 |    | A(w)  |
|  1   w^2  w^4   w^6 ... w^{2(m-1)}     |  | a_2 | == | A(w^2)|
|  1   w^3  w^6   w^9 ... w^{3(m-1)}     |  | a_3 |    | A(w^3)|
|           .....       .....           |  | ... |    | ...   |
|  1 w^{-1} w^{-2} w^{-3} ... w          |  |a_m-1|    | ...   |
+-----------------------------------------+  +-----+    +-------+
```

(Note w^{m-1} = w^{-1} since w^m = 1)

(We're doing this "implicitly" in the sense that we don't even have time to write down the matrix.)

To do the inverse transform, what we want is to multiply by the inverse of the F matrix. As it turns out, this inverse looks very much like F itself. In particular, notice that w^{-1} is also a principal mth root of unity. Let's define \bar{F} to be the fourier transform matrix using w^{-1} instead of w. Then,

Claim: F^{-1} = (1/m) * \bar{F}. I.e., 1/m * \bar{F} * F = identity.

Proof: What is the i,j entry of \bar{F} * F? It is:

1 + w^{j-i} + w^{2j-2i} + w^{3j-3i} + ... + w^{(m-1)j - (m-1)i}

If i=j, then these are all = 1, so the sum is m. Then when we divide by m we get 1.

If i!=j, then the claim is these all cancel out and we get zero. Maybe easier to see if we let z = w^{j-i}, so then the sum is:

1 + z + z^2 + z^3 + z^4 + ... + z^{m-1}.

Then can see these cancel by picture. For instance, try z = w, z = w^2.

Or can use the formula for summations: (1 - z^m)/(1-z) = 0/(1-z) = 0.

So, the final algorithm is:

```
Let F_A = FFT(A, m, w)                      // time O(n log n)
Let F_B = FFT(B, m, w)                      // time O(n log n)
For i=1 to m, let F_C[i] = F_A[i]*F_B[i]    // time O(n)
Output C = 1/m * FFT(F_C, m, w^{-1}).       // time O(n log n)
```

NOTE: If you're an EE or Physics person, what we're calling the "Fourier Transform" is what you would usually call the "inverse Fourier transform" and vice-versa.

----------Avrim's Notes (end)----------

If the polynomials have integer coefficients, then the convolution is also integer valued. Avrim mentions that it would be nice if our algorithm for doing the convolution did not depend on using a floating point representation of complex numbers. Here are some more details on how to do this.

Let B be a bound on the biggest number we're interested in. (For example, if we're doing a convolution of two vectors of length n whose values are bounded by b then we need B>b^2n.) Our exposition assumes that the vectors we're convolving are comprised of non-negative integers. This assumption is easily eliminated, but we stick to it here for clarity.

We will run the algorithm in the field of integers modulo p (for some prime p to be suitably chosen). The multiplicative group of this field is denoted Z_p^*. We must choose p >= B so that arithmetic (adding and multiplying) on any pair of numbers that results in a number less than B (over the field of all integers) will give the same result when we operate in the field of integers modulo p.

Let F_p be the field of integers modulo p. Here are the properties we need:

(1) p >= B
(2) F_p contains a principle nth root of unity (n is a power of 2)

Consider a prime of the form p = k*n + 1, for some integer k.
By Fermat's theorem we know that for any 0 < g < p:

g^(p-1) = 1 (mod p)

==> g^(k*n) = 1 (mod p)

==> (g^k)^n = 1 (mod p)

Therefore g^k = r is a candidate for a principle nth root of unity.
We just need to check that r^1, r^2, ... , r^n = 1 are all distinct.

So here's a practical way to find the necessary numbers. Start trying all k's (sufficiently large), trying to find a value such that k*n+1 is a prime. Primes are dense, so it's very easy to find such values.

Now pick g at random and test r = g^k is a principle nth root of unity. Keep trying until one works. In practice this is very fast. Below is ocaml code to do this.

----------ocaml code to generate the field----------

```
(* We're given n, which is 2^q.  We want a field to compute
   convolutions in.  The field needs to be isomorphic to the integers
   (if they're small enough).  Say we know B, an upper bound on the biggest
   number that could occur in the convolution.

   Let Z_p^* be the multiplicative group modulo p.  It's known to be
   cyclic.  Let g be a generator.  If it is the case that p = kn+1,
   then we know that g^(p-1) = 1.  Therefore g^(kn) = 1.  Therefore
   (g^k)^n = 1.  Also since g is a generator we know that all of these
   n powers of g^k are distinct.  Therefore g^k is a principal nth
   root of unity.

   Let's check here if it's easy to generate such numbers.
*)

let isprime n =
  let rec loop i = (i*i > n) || (n mod i <> 0 && loop (i+1)) in
  n >= 2 && loop 2

let rec powermod a p n =
  let sq x = x*x mod n in
  if p=0 then 1 else
    let x = sq (powermod a (p/2) n) in
    if p land 1 = 0 then x else (a*x) mod n
```

```
let find_good_prime q b =
  (* find a prime p such that p > b and p=(2^q)*k+1 for some k *)
  let n = 1 lsl q in

  let min_k = (b+n-1)/n in  (* ceiling b/n *)

  let max_k = (max_int - 1)/n in

  let rec loop k = if k>=max_k then -1 else
      if isprime (n*k+1) then k else loop (k+1)
  in

  let k = loop min_k in
  (k, n*k+1)

let () = Random.self_init ()

let find_principle_root_of_unity q p =
  let n = 1 lsl q in
  let k = (p-1)/n in

  let rec loop () =
    Printf.printf "*\n";
    let g = 2 + Random.int (min (1 lsl 29) (p-2)) in (* between 2 and p-1 *)
    let r = powermod g k p in
    let h = Hashtbl.create 10 in
    for i=1 to q do
      Hashtbl.replace h (powermod r (1 lsl i) p) true;
    done;
    if Hashtbl.length h = q then r else loop ()
  in

  loop ()

let inverse a p =  (* compute the inverse of a modulo p *)
  let rec egcd a b =
    (* given a and b this returns (x, y, g) where g = gcd(a,b) and ax+by=g  *)
    if a = 0 then (0,1,b) else
      let (x,y,g) = egcd (b mod a) a in
      (y-(b/a)*x, x, g)
  in
  let (x,_,_) = egcd a p in
  if x >= 0 then x else x+p

let q = 8
let n = 1 lsl q
let b = 1_000_000
let (k,p) = find_good_prime q b
let w = find_principle_root_of_unity q p
let n' = inverse n p
let w' = inverse w p

let () = Printf.printf "q=%d  b=%d ----> n=%d  n'=%d  k=%d  p=%d  w=%d  w'=%d\n"
  q b n n' k p w w'
```

--------------------end ocaml code to generate the field----------------------

Below is an implementation of the FFT algorithm in ocaml.

-----------------------ocaml FFT implementation-----------------------------

```
let q=3
let n=8
let n'=99
let b=100
let k=14
```

```
let p=113
let w=95
let w'=69

(* Arithmetic modulo p. Assuming an input number is in [0,p-1] the
   output is also in [0,p-1]. *)
let ( ++ ) a b = (a+b) mod p
let ( ** ) a b = (a*b) mod p
let ( -- ) a b = if a>=b then a-b else a-b+p

let rec fft a n w =
  (* array a of length n with w a principle nth root of unity mod p *)
  (* return the fft of this vector *)

  if n = 1 then a else
    let n' = n/2 in
    let w' = w ** w in
    let aeven = Array.init n' (fun i -> a.(2*i)) in
    let aodd  = Array.init n' (fun i -> a.(2*i+1)) in
    let veven = fft aeven n' w' in
    let vodd  = fft aodd  n' w' in
    let v = Array.make n 0 in

    let rec loop j wj = if j < n' then (
      v.(j) <- veven.(j) ++ wj ** vodd.(j);
      v.(j+n') <- veven.(j) -- wj ** vodd.(j);
      loop (j+1) (w**wj)
    ) in
    loop 0 1;
    v

let fft_inverse a n w' =
  let v = fft a n w' in
  Array.init n (fun i -> v.(i) ** n')

open Printf

let print_vector a name =
  let n = Array.length a in
  printf "%15s =" name;
  for i=0 to n-1 do
    Printf.printf " %4d" a.(i)
  done;
  print_newline()

let () =
  let a = Array.init n (fun i -> i) in
  let v = fft a n w in
  let a' = fft_inverse v n w' in

  print_vector a "a";
  print_vector v "fft a";
  print_vector a' "fft inverse";

  let c = [|1;1;0;0;0;0;1;1|] in
  let d = [|1;1;0;0;0;0;1;1|] in

  let c' = fft c n w in
  let d' = fft d n w in
  let prod = Array.init n (fun i -> c'.(i) ** d'.(i)) in
  let conv = fft_inverse prod n w' in

  print_newline();
  print_vector c "c";
  print_vector d "d";
  print_vector conv "conv c d";
```

----------END ocaml FFT implementation----------

Applications of the FFT.

- The algorithm works in higher dimensinons (2D is important)
- Signal Processing
- Image Compression
- Force computations
- Computerized Axial Tomography (CAT scans)
- Simulating Linear Cellular Automata