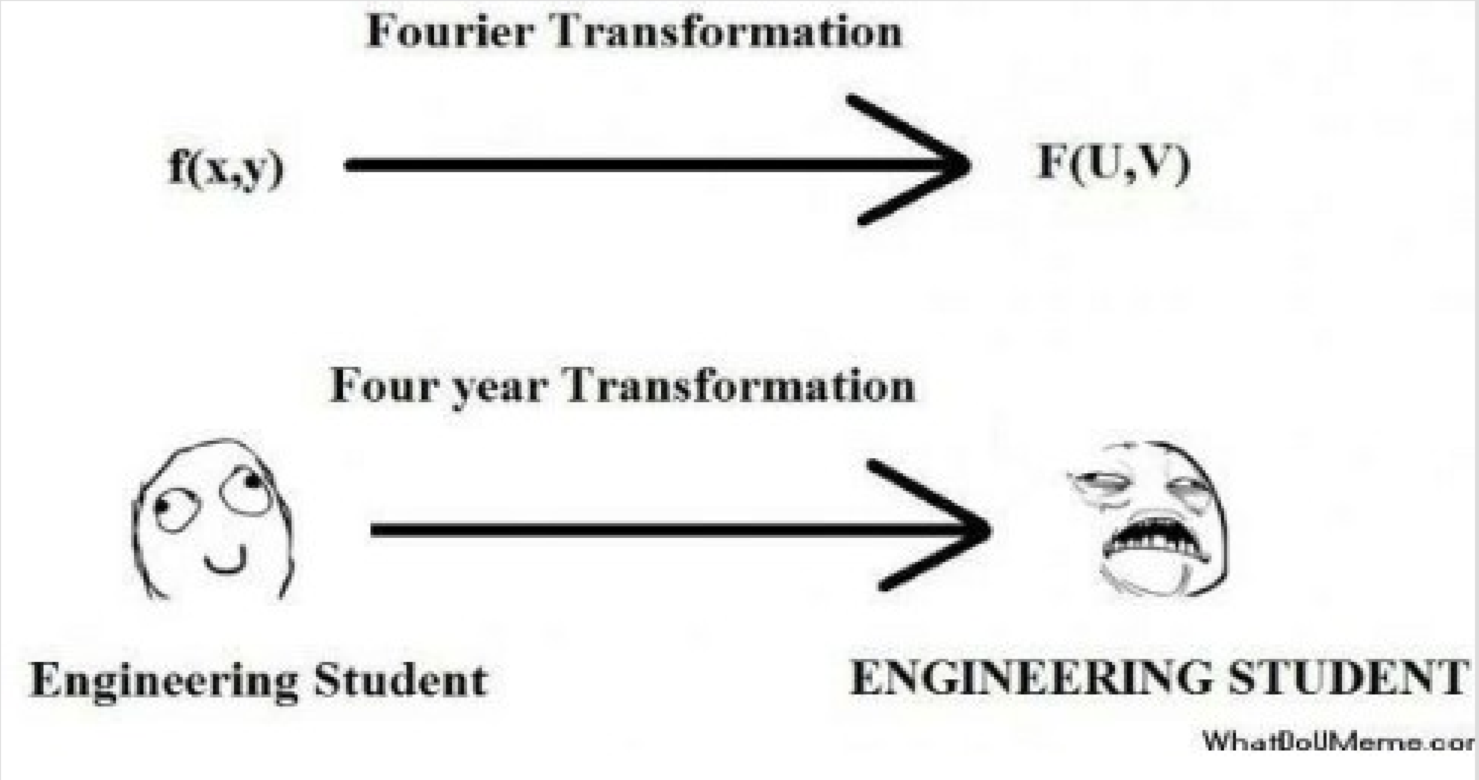
Fast Fourier Transform



Announcements

HW 3 posted, due Sunday

Fast Fourier Transform

Suppose....

$A(x) = (x+1)$

$B(x) = (x^2 - 2x + 3)$

The A(x)*B(x) will be degree 3 (thus 4 coefficients)

So 4 points needed on A(x) and B(x)

Fast Fourier Transform

To do this we buffer some "0" coefficients:

$A(x) = (x+1) = (0x^3 + 0x^2 + x + 1)$

So coefficients (from power 0) = [1 1 0 0]

From this we can run FFT

Fast Fourier Transform

$A(x) = A^{[0]}(x^2) + x*A^{[1]}(x^2)$

$A(1) = A^{[0]}(1) + A^{[1]}(1)$

$A(i) = A^{[0]}(-1) + i*A^{[1]}(-1)$

$A(-1) = A^{[0]}(1) + -1*A^{[1]}(1)$

$A(-i) = A^{[0]}(-1) + -i*A^{[1]}(-1)$

... so we need to find $A^{[0]}$ and $A^{[1]}$ at x=1 and x=-1

Fast Fourier Transform

$A^{[0]}(x)$ = coefficients [1 0] = $1 + 0*x$

$A^{[0]}(x) = A^{[0][0]}(x^2) + x*A^{[0][1]}(x^2)$

$A^{[0][0]}(x)$ = coefficients [1]

... so $A^{[0][0]}(x) = 1$ (... an easy poly)

Likewise $A^{[0][1]}(x) = 0$

Fast Fourier Transform

$A^{[0]}(x) = A^{[0][0]}(x^2) + x*A^{[0][1]}(x^2)$

$A^{[0][0]}(x) = 1$

$A^{[0][1]}(x) = 0$

$A^{[0]}(1) = A^{[0][0]}(1^2) + 1*A^{[0][1]}(1^2)$

$= 1 + 1*0 = 1$

$A^{[0]}(-1) = A^{[0][0]}((-1)^2) + -1*A^{[0][1]}((-1)^2)$

$= 1 + -1*0 = 1$

Fast Fourier Transform

$A^{[1]}(x)$ = coeffecients [1 0] = 1 + 0*x

... this is identical to A[0](x),
so we repeat this and get:

$A^{[1]}(1) = 1$
$A^{[1]}(-1) = 1$

Fast Fourier Transform

$$A(1) = A^{[0]}(1) + A^{[1]}(1)$$
$$= 1 + 1 = 2$$
$$A(i) = A^{[0]}(-1) + i*A^{[1]}(-1)$$
$$= 1 + i*1$$
$$A(-1) = A^{[0]}(1) + -1*A^{[1]}(1)$$
$$= 1 + -1*1 = 0$$
$$A(-i) = A^{[0]}(-1) + -i*A^{[1]}(-1)$$
$$= 1 + -i * 1 = 1-i$$

Fast Fourier Transform

Thus $A(x) = 1+x$ in the point-value representation is:

(1, 2)

(i, 1+i)

(-1, 0)

(-i, 1 - i)

Can verify by plugging in for x

Fast Fourier Transform

Now we do the same thing for B(x)...

$B(x) = 0*x^3 + (x^2 - 2x + 3)$

$= \text{coefficients } [3\ -2\ 1\ 0]$

$B(x) = B^{[0]}(x^2) + x*B^{[1]}(x^2)$

$B^{[0]}(x) = \text{coef } [3\ 1] = 3 + x$

$B^{[1]}(x) = \text{coef } [-2\ 0] = -2$

Fast Fourier Transform

$B^{[0]}(x)$ = coef [3 1] = 3 + x

$B^{[0]}(x) = B^{[0][0]}(x^2) + x*B^{[0][1]}(x^2)$

$B^{[0][0]}(x)$ = coef [3] = 3 (for any x)

$B^{[0][1]}(x)$ = coef [1] = 1

Evaluate B[0](x) at 2 points as 2 coef, so we use w_2^0 and w_2^1, so 1 and -1

Fast Fourier Transform

$B^{[0]}(x) = B^{[0][0]}(x^2) + x*B^{[0][1]}(x^2)$

$B^{[0][0]}(x) = 3$

$B^{[0][1]}(x) = 1$

$B^{[0]}(1) = B^{[0][0]}(1^2) + 1*B^{[0][1]}(1^2)$

$= 3 + 1*1 = 4$

$B^{[0]}(-1)= B^{[0][0]}((-1)^2) + -1*B^{[0][1]}((-1)^2)$

$= 3 * -1*1 = 2$

Fast Fourier Transform

$B^{[1]}(x) = B^{[1][0]}(x^2) + x*B^{[1][1]}(x^2)$

$B^{[1]}(x) = \text{coef } [-2\ 0]$

$B^{[1][0]}(x) = -2 = \text{coef } [-2]$

$B^{[1][1]}(x) = 0 = \text{coef}[0]$

$B^{[1]}(1) = B^{[1][0]}(1^2) + 1*B^{[1][1]}(1^2)$

$= -2 + 1*0 = -2$

$B^{[1]}(-1)= B^{[1][0]}((-1)^2) + -1*B^{[1][1]}((-1)^2)$

$= -2 * -1*0 = -2$

Fast Fourier Transform

$B(1) = B^{[0]}(1) + 1*B^{[1]}(1)$
$= 4 + -2 = 2$

$B(i) = B^{[0]}(-1) + i*B^{[1]}(-1)$
$= 2 + i*-2 = 2 - 2i$

$B(-1) = B^{[0]}(1) + -1*B^{[1]}(1)$
$= 4 + -1*-2 = 6$

$B(-i) = B^{[0]}(-1) + -i*B^{[1]}(-1)$
$= 2 + -i * -2 = 2 + 2i$

Fast Fourier Transform

$B(x) = (x^2 - 2x + 3)$

Thus B(x) in point-value notation is:

(1, 2)

(i, 2 - 2i)

(-1, 6)

(-i, 2 + 2i)

Fast Fourier Transform

A(x)	B(x)	C(x)
(1, 2)	(1, 2)	(1, 2*2)
(i, 1+i)	(i, 2 - 2i)	(i, (i+1)(2-2i))
(-1, 0)	(-1, 6)	(-1, 0*6)
(-i, 1 - i)	(-i, 2 + 2i)	(-i, (i-i)(2+2i))

... and then we do this whole thing in reverse...

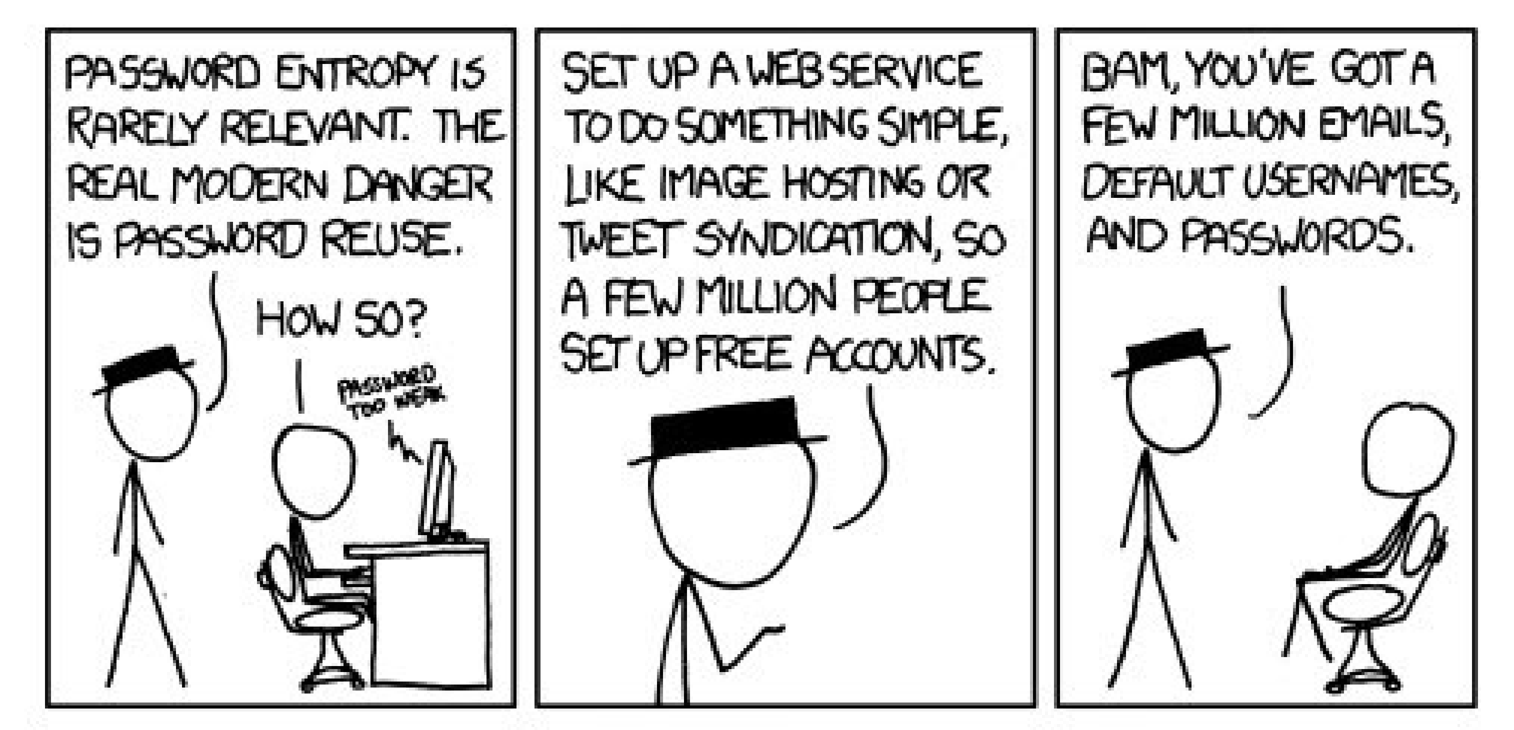
Fast Fourier Transform

When going in reverse only differences:

Rather than going 1 to i to -1 to -i...
Go other way: 1 to -i to -1 to i

Then divide all coefficients by n at the end

Prime numbers (cryptography)



RSA Encryption

RSA person A has two keys:

P_A = public key

S_A = secret key (private key)

The key is that these functions are inverse, namely for some message M:

$$P_A(S_A(M)) = S_A(P_A(M)) = M$$

RSA Encryption

Thus, if person B wants to send a secret message to person A, they do:

1. Encrypt the message using public key: $C = P_A(M)$
2. Then A can decrypt it using the secret key: $M = S_A(C)$

RSA Encryption

If A does not share S_A, no one else knows the proper way to decrypt C

$P_A(P_A(M)) \neq M$

... and ...

S_A not easily computable from P_A

(more on this next week)

RSA Encryption

RSA algorithm:

1. Select two large primes p, q ($p \neq q$)
2. Let n = p * q
3. Let e be: $\gcd(e, (p - 1)*(q - 1)) = 1$
4. Let d be: $e*d \bmod (p-1)*(q-1) = 1$
(use “extended euclidean” in book)
5. Public key: P = (e, n)
6. Secret key: S = (d, n)

RSA Encryption

Specifically:

$P_A(M) = M^e \bmod n$

$S_A(C) = C^d \bmod n$

A key assumption is that $M < n$, as we want: $M \bmod n = M$

Pick large p,q or encode per byte

RSA Encryption

Example: p=7, q=11... n = p*q = 77

e=13 (does not need to be prime) as gcd(13,(7-1)(11-1))=gcd(13,60) = 1

d=37 as 13*37 mod 60 = 1

If M = 20 (a byte), then C = 20^{13} mod 77 = 69

C = 71, 71^{37} mod 77 = 20

RSA Encryption + CRT

Computing large powers can require a lot of processor power

Can more efficiently get the result with Chinese remainder theorem: (backwards)
Have: number mod product
Want: smaller system of equations

RSA Encryption + CRT

Using CRT:

$m1 = C^{d \bmod p-1} \bmod p$ // less compute

$m2 = C^{d \bmod q-1} \bmod q$ // much smaller

$qI = q^{-1} \bmod p$

$h = qI * (m1 - m2)$

$m = m2 + h*q$

(see: rsa.cpp)

Primes

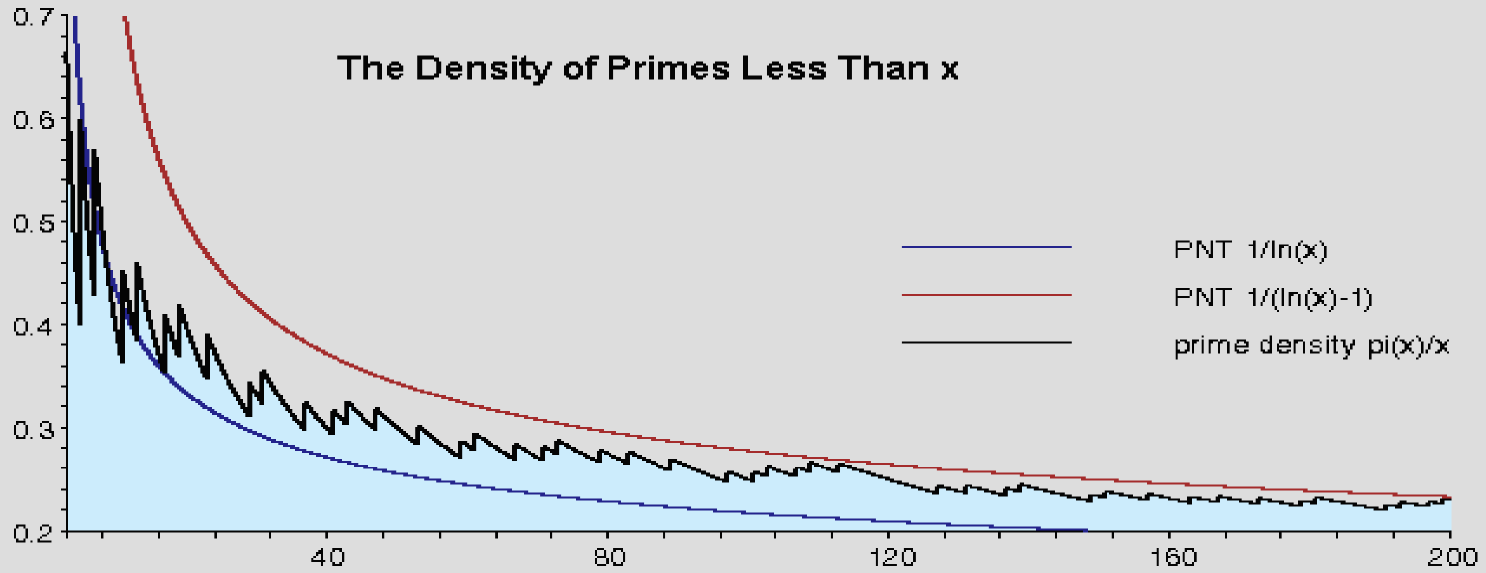
RSA (and many other applications) require large prime numbers

We need to find these efficiently (not brute force!)

The common methods are actually probabilistic (no guarantee)

Primes

First, are there actually large primes?



Density of primes around x is about 1/ln(x) (i.e. 3 per 100 when $x=10^{10}$)

Prime finding

To find them, we just make a smart guess then check if it really is prime

Smart guess:
last digit not: 2, 4, 5, 6, 8 or 0

This eliminates 60% of numbers!

Prime finding

Both of these methods use Fermat's theorem, for a prime p:

$$a^{p-1} \mod p = 1, \forall a \in \mathbb{Z}$$

So we simply check if:

2^{p-1} mod p == 1

If this is, probably prime

Prime finding

This simplistic method works surprisingly well:

Error rate less than 0.2%

(if around 512 bit range, 1 in 10^{20})

Has two major issues:

1. More accurate for large numbers
2. Carmichael numbers(e.g. 561, rare)

Prime finding

Computation time also goes up with number size

Carmichael numbers are composite, but have: $a^{p-1} \bmod p = 1$ for all a

These are quite rare though (only 255 less than 100,000,000)

Miller-Rabin primality test

Again, we will basically test Fermat's theorem but with a twist

We let: n-1 = u * 2^t, for some u and t

Then compute: $a^{n-1} \mod n == 1$

As: $a^{u \cdot 2^t} \mod n == 1$

(more efficient, as we can square it)

Miller-Rabin primality test

```
Witness(a, n)
find (t,u) such that t≥1 and n-1=u*2^t
x_0 = a^u mod n
for i = 1 to t
    x_i = x^2_{i-1} mod n
    if x_i == 1 and x_{i-1} ≠ 1 and x_{i-1} ≠ n-1
        return true
if x_i ≠ 1
    return true
return false
```

Miller-Rabin primality test

If Witness returns true, the number is composite

If Witness returns false, there is a 50% probability that it is a prime

Thus testing “s” different values of “a” (range 0 to n-1) gives error 2^{-s}

Composites

To find composites of n takes (we think) O(sqrt(n))

This is the same asymptotic running time as brute force

(i.e. n%2 ==0, n%3==0, ...)

Composites

Many security systems depend on the fact that factoring nubmers is (we think) a hard problem

In RSA, if you could factor n into p and q, anyone can get private key

However, no one has been able to prove that this is hard

Composites

The book does give an algorithm to compute composites

Similar to security hashing:
(finding hash collision)

Still O(sqrt(n))
(smaller coefficient)