CS/COE 1501
Algorithm Implementation

4-Algorithms for Big Integers

Fall 2018

Sherif Khattab

ksm73@pitt.edu

6307 Sennott Square

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Integer multiplication

- Say we have 5 baskets with 8 apples in each
  - How do we determine how many apples we have?
    - Count them all?
      - That would take a while…
    - Since we know we have 8 in each basket, and 5 baskets, lets simply add 8 + 8 + 8 + 8 + 8
      - = 40!
    - This is essentially multiplication!
      - 8 * 5 = 8 + 8 + 8 + 8 + 8

What about bigger numbers?

- Like 1284 * 1583, I mean!
  - That would take way longer than counting the 40 apples!
- Let's think of it like this:
  - 1284 * 1583 = 1284*3 + 1284*80 + 1284*500 + 1284*1000

```
      1284
x     1583
-----------
      3852
+   102720
+   642000
+  1284000
-----------
= 2032572
```

OK, I'm guessing we all knew that...

- … and learned it quite some time ago …
- So why bring it up now? What is there to cover about multiplication
- What is the runtime of this multiplication algorithm?
  - For 2 n-digit numbers:
    - n^2

Yeah, but the processor has a MUL instruction

- Assuming x86
- Given two 32-bit integers, MUL will produce a 64-bit integer in a few cycles
- What about when we need to multiply large ints?
  - VERY large ints?
    - RSA keys should be 2048 bits
  - Back to grade school…

Gradeschool algorithm on binary numbers

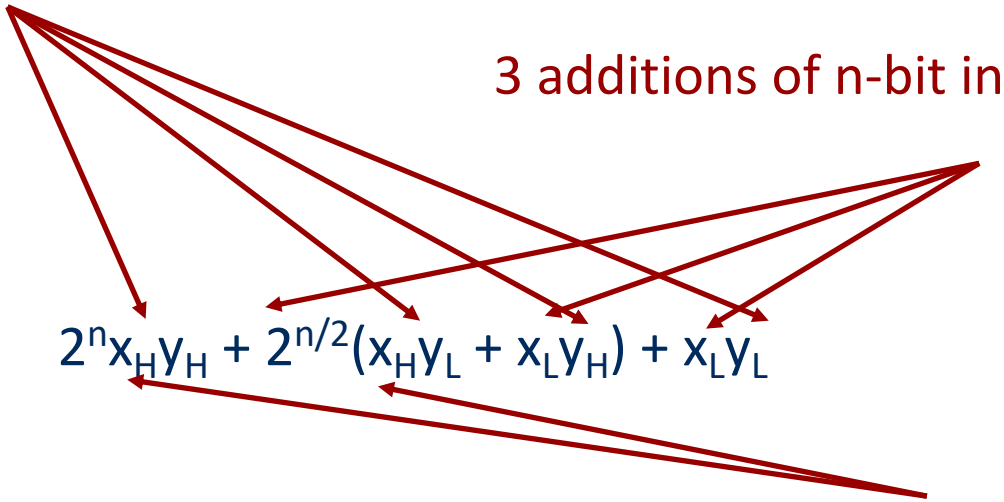
```
          10100000100
x         11000101111
-----------------------
          10100000100
         101000001000
        1010000010000
       10100000100000
      000000000000000
     1010000010000000
    0000000000000000
   00000000000000000
  000000000000000000
 1010000010000000000
1010000010000000000
-----------------------
11111000000111011110 0
```

How can we improve our runtime?

- Let's try to divide and conquer:
  - Break our n-bit integers in half:
    - x = 100101011001000, n = 16
    - Let the high-order bits be x_H = 10010110
    - Let the low-order bits be x_L = 11001000
    - $x = 2^{n/2}x_H + x_L$
    - Do the same for y
    - $x * y = (2^{n/2}x_H + x_L) * (2^{n/2}y_H + y_L)$
    - $x * y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$

So what does this mean?

4 multiplications of n/2 bit integers



Actually 16 multiplications of n/4 bit integers (plus additions/shifts)

Actually 64 multiplications of n/8 bit integers (plus additions/shifts)

...

So what's the runtime???

- Recursion really complicates our analysis…
- We'll use a *recurrence relation* to analyze the recursive runtime
  - Goal is to determine:
    - How much work is done in the current recursive call?
    - How much work is passed on to future recursive calls?
    - All in terms of input size

Recurrence relation for divide and conquer multiplication

- Assuming we cut integers exactly in half at each call
  - I.e., input bit lengths are a power of 2
- Work in the current call:
  - Shifts and additions are $\Theta(n)$
- Work left to future calls:
  - 4 more multiplications on half of the input size

- $T(n) = 4T(n/2) + \Theta(n)$

Soooo… what's the runtime?

- Need to solve the recurrence relation
  - Remove the recursive component and express it purely in terms of n
    - A "cookbook" approach to solving recurrence relations:
      - The master theorem

The master theorem

- Usable on recurrence relations of the following form:

$$T(n) = aT(n/b) + f(n)$$

- Where:
  - a is a constant >= 1
  - b is a constant > 1
  - and f(n) is an asymptotically positive function

Applying the master theorem

$$T(n) = aT(n/b) + f(n)$$

- If $f(n)$ is $O(n^{\log_b(a) - \varepsilon})$:
  - $T(n)$ is $\Theta(n^{\log_b(a)})$
- If $f(n)$ is $\Theta(n^{\log_b(a)})$
  - $T(n)$ is $\Theta(n^{\log_b(a)} \lg n)$
- If $f(n)$ is $\Omega(n^{\log_b(a) + \varepsilon})$ and $(a * f(n/b) <= c * f(n))$ for some $c < 1$:
  - $T(n)$ is $\Theta(f(n))$

Mergesort master theorem analysis

Recurrence relation for mergesort? $T(n) = 2T(n/2) + \Theta(n)$

- a = 2
- b = 2
- f(n) is $\Theta(n)$
- So...
  - $n^{\log_b(a)} = \ldots$
    - $n^{\lg 2} = n$

- If f(n) is $O(n^{\log_b(a) - \varepsilon})$:
  - T(n) is $\Theta(n^{\log_b(a)})$
- If f(n) is $\Theta(n^{\log_b(a)})$
  - T(n) is $\Theta(n^{\log_b(a)} \lg n)$
- If f(n) is $\Omega(n^{\log_b(a) + \varepsilon})$

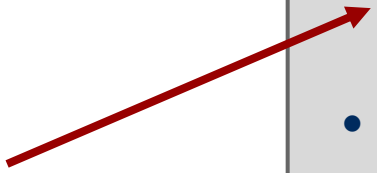
  and (a * f(n/b) <= c * f(n)) for some c < 1:
  - T(n) is $\Theta(f(n))$

  - Being $\Theta(n)$ means f(n) is $\Theta(n^{\log_b(a)})$
  - $T(n) = \Theta(n^{\log_b(a)} \lg n) = \Theta(n^{\lg 2} \lg n) = \Theta(n \lg n)$

For our divide and conquer multiplication approach

$$T(n) = 4T(n/2) + \Theta(n)$$

- $a = 4$
- $b = 2$
- $f(n)$ is $\Theta(n)$
- So...
  - $n^{\log_b(a)} = \ldots$
    - $n^{\lg 4} = n^2$



- If $f(n)$ is $O(n^{\log_b(a) - \varepsilon})$:
  - $T(n)$ is $\Theta(n^{\log_b(a)})$
- If $f(n)$ is $\Theta(n^{\log_b(a)})$
  - $T(n)$ is $\Theta(n^{\log_b(a)} \lg n)$
- If $f(n)$ is $\Omega(n^{\log_b(a) + \varepsilon})$

  and $(a * f(n/b) <= c * f(n))$ for some $c < 1$:
  - $T(n)$ is $\Theta(f(n))$

  - Being $\Theta(n)$ means $f(n)$ is polynomially smaller than n^2
  - $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\lg 4}) = \Theta(n^2)$

@#$%^&*

- Leaves us back where we started with the grade school algorithm…
  - Actually, the overhead of doing all of the dividing and conquering will make it slower than grade school

SO WHY EVEN BOTHER?

- Let's look for a smarter way to divide and conquer
- Look at the recurrence relation again to see where we can improve our runtime:

$$T(n) = 4T(n/2) + \Theta(n)$$

Can we reduce the number of subproblems?

Can we reduce the subproblem size?

Can we reduce the amount of work done by the current call?

Karatsuba's algorithm

- By reducing the number of recursive calls (subproblems), we can improve the runtime
- $x * y = 2^n x_H y_H + 2^{n/2}(x_H y_L + x_L y_H) + x_L y_L$

  M1 M2 M3 M4

- We don't actually need to do both M2 and M3
  - We just need the sum of M2 and M3
    - If we can find this sum using only 1 multiplication, we decrease the number of recursive calls and hence improve our runtime

Karatsuba craziness

- $M1 = x_h y_h$; $M2 = x_h y_l$; $M3 = x_l y_h$; $M4 = x_l y_l$;
- The sum of all of them can be expressed as a single mult:
  - $M1 + M2 + M3 + M4$
  - $= x_h y_h + x_h y_l + x_l y_h + x_l y_l$
  - $= (x_h + x_l) * (y_h + y_l)$
- Lets call this single multiplication M5:
  - $M5 = (x_h + x_l) * (y_h + y_l) = M1 + M2 + M3 + M4$
- Hence, $M5 - M1 - M4 = M2 + M3$
- So: $x * y = 2^n M1 + 2^{n/2}(M5 - M1 - M4) + M4$
  - Only 3 multiplications required!
  - At the cost of 2 more additions, and 2 subtractions

Karatsuba runtime

- To get M5, we have to multiply (at most) n/2 + 1 bit ints
  - Asymptotically the same as our other recursive calls
- Requires extra additions and subtractions…
  - But these are all $\Theta(n)$
- So, the recurrence relation for Karatsuba's algorithm is:
  - $T(n) = 3T(n/2) + \Theta(n)$
    - Which solves to be $\Theta(n^{\lg 3})$
      - Asymptotic improvement over grade school algorithm!
        - For large n, this will translate into practical improvement

Large integer multiplication in practice

- Can use a hybrid algorithm of grade school for large operands,

  Karatsuba's algorithm for VERY large operands

  - Why are we still bothering with grade school at all?

Is this the best we can do?

- The Schönhage–Strassen algorithm
  - Uses Fast Fourier transforms to achieve better asymptotic runtime
    - O(n log n log log n)
    - Fastest asymptotic runtime known from 1971-2007
      - Required n to be astronomical to achieve practical improvements to runtime
        - Numbers beyond $2^{2^{15}}$ to $2^{2^{17}}$
- Fürer was able to achieve even better asymptotic runtime in 2007
  - n log n $2^{O(\log^* n)}$
  - No practical difference for realistic values of n

Exponentiation

- x^y
- Can easily compute with a simple algorithm:

```
ans = 1
i = y
while i > 0:
   ans = ans * x
   i--
```

- Runtime?

Just like with multiplication, let's consider large integers...

- Runtime = # of iterations * cost to multiply
- Cost to multiply was covered in the last lecture
- So how many iterations?
  - Single loop from 1 to y, so linear, right?
    - What is the size of our input?
      - n
        - The *bitlength* of y…
    - So, linear in the *value* of y…
      - But, increasing n by 1 doubles the number of iterations
  - $\Theta(2^n)$
    - Exponential in the *bitlength* of y

This is RIDICULOUSLY BAD

- Assuming 512 bit operands, 2^{512}:
  - 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096
- Assuming we can do mults in 1 cycle…
  - Which we *can't* as we learned last lecture
- And further that these operations are completely parallelizable
- 16 4GHz cores = 64,000,000,000 cycles/second
  - $(2^{512} / 64000000000) / (3600 * 24 * 365) =$
    - $6.64 * 10^{135}$ years to compute

This is way too long to do exponentiations!

- So how do we do better?
- Let's try divide and conquer!
- $x^y = (x^{(y/2)})^2$
  - When y is even, $(x^{(y/2)})^2$ * x when y is odd
- Analyzing a recursive approach:
  - Base case?
    - When y is 1, x^y is x; when y is 0, x^y is 1
  - Runtime?

Building another recurrence relation

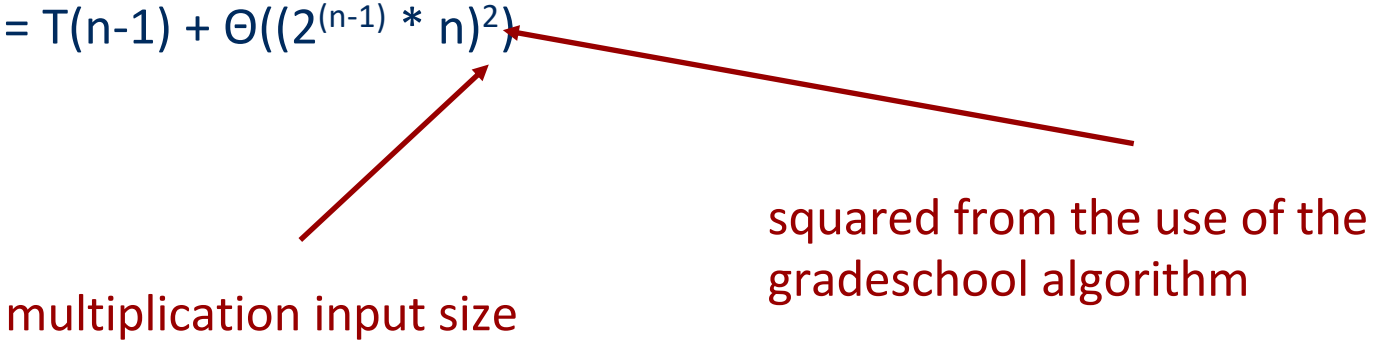
- $x^y = (x^{(y/2)})^2 = x^{(y/2)} * x^{(y/2)}$
  - Similarly, $(x^{(y/2)})^2 * x = x^{(y/2)} * x^{(y/2)} * x$
- So, our recurrence relation is:
  - $T(n) = T(n-1) + ?$
    - How much work is done per call?
    - 1 (or 2) multiplication(s)
      - Examined runtime of multiplication last lecture
      - But how big are the operands in this case?

Determining work done per call

- Base case returns x
  - n bits
- Base case results are multiplied: x * x
  - n bit operands
  - Result size?
    - 2n
- These results are then multiplied: $x^2 * x^2$
  - 2n bit operands
  - Result size?
    - 4n bits
- …
- $x^{(y/2)} * x^{(y/2)}$?
  - (y / 2) * n bit operands = $2^{(n-1)}$ * n bit operands
  - Result size? y * n bits = 2^n * n bits

Multiplication input size increases throughout

- Our recurrence relation looks like:
  - $T(n) = T(n-1) + \Theta((2^{(n-1)} * n)^2)$



Runtime analysis

- Can we use the master theorem?
  - Nope, we don't have a b > 1
- OK, then…
  - How many times can y be divided by 2 until a base case?
    - lg(y)
  - Further, we know the max value of y
    - Relative to n, that is:
      - 2^n
  - So, we have, at most lg(y) = lg(2^n) = n recursions

But we need to do expensive mult in each call

- We need to do $\Theta((2^{(n-1)} * n)^2)$ work in just the root call!
  - Our runtime is dominated by multiplication time
    - Exponentiation quickly generates HUGE numbers
    - Time to multiply them quickly becomes impractical

Can we do better?

- We go “top-down” in the recursive approach
  - Start with y
  - Halve y until we reach the base case
  - Square base case result
  - Continue combining until we arrive at the solution
- What about a “bottom-up” approach?
  - Start with our base case
  - Operate on it until we reach a solution

A bottom-up approach

- To calculate x^y

```
ans = 1
foreach bit in y:
        ans = ans²
        if bit == 1:
                ans = ans * x
```

From most to least significant

Bottom-up exponentiation example

- Consider x^y where y is 43 (computing x^{43})
- Iterate through the bits of y (43 in binary: 101011)
- ans = 1

ans = 1^2	= 1	
ans = 1 * x	= x	
ans = x^2		= x^2
ans = $(x^2)^2$		= x^4
ans = x^4 * x	= x^5	
ans = $(x^5)^2$		= x^{10}
ans = $(x^{10})^2$		= x^{20}
ans = x^{20} * x	= x^{21}	
ans = $(x^{21})^2$	= x^{42}	
ans = x^{42} * x	= x^{43}	

Does this solve our problem with mult times?

- Nope, still squaring ans everytime
  - We'll have to live with huge output sizes
- This does, however, save us recursive call overhead
  - Practical savings in runtime

Greatest Common Divisor

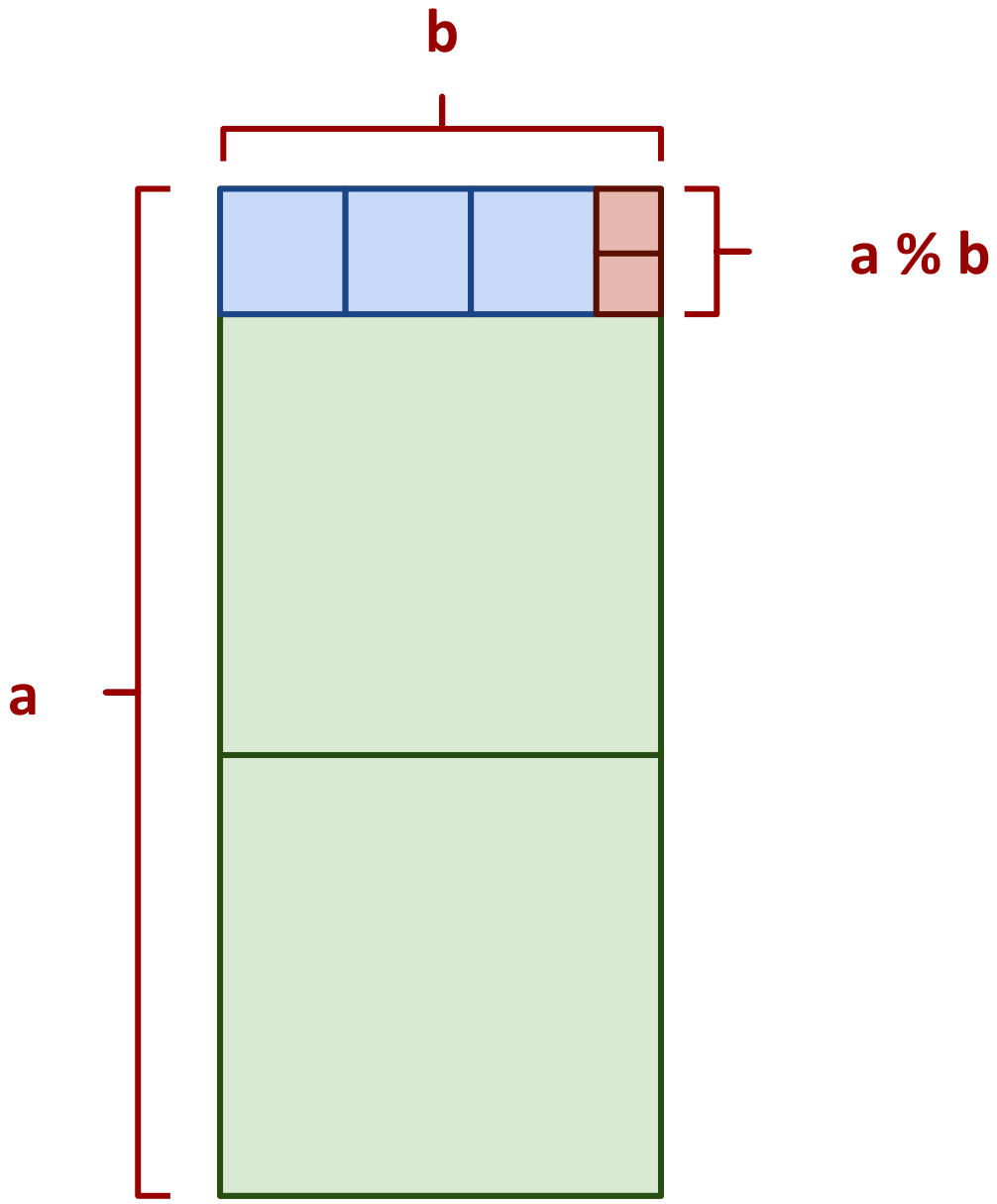
- GCD(a, b)
  - Largest int that evenly divides both a and b
- Easiest approach:
  - BRUTE FORCE

```
i = min(a, b)
while(a % i != 0 || b % i != 0):
        i--
```

- Runtime?
  - Θ(min(a, b))
  - Linear!
    - In *value* of min(a, b)...
  - Exponential in n
    - Assuming a, b are n-bit integers

Euclid’s algorithm

- GCD(a, b) = GCD(b, a % b)



Euclidean example 1

- GCD(30, 24)
  - = GCD(24, 30 % 24)
- = GCD(24, 6)
  - = GCD(6, 24 % 6)
- = GCD(6, 0)...
  - Base case! Overall GCD is 6

Euclidean example 2

- = GCD(99, 78)
  - 99 = 78 * 1 + 21
- = GCD(78, 21)
  - 78 = 21 * 3 + 15
- = GCD(21, 15)
  - 21 = 15 * 1 + 6
- = GCD (15, 6)
  - 15 = 6 * 2 + 3
- = GCD(6, 3)
  - 6 = 3 * 2 + 0
- = 3

Analysis of Euclid's algorithm

- Runtime?
  - Tricky to analyze, has been shown to be linear in n
    - Where, again, n is the number of bits in the input

Extended Euclidean algorithm

- In addition to the GCD, the Extended Euclidean algorithm (XGCD) produces values x and y such that:
  - GCD(a, b) = i = ax + by
- Examples:
  - GCD(30,24) = 6 = 30 * 1 + 24 * -1
  - GCD(99,78) = 3 = 99 * -11 + 78 * 14
- Can be done in the same linear runtime!

Extended Euclidean example

- = GCD(99, 78)
  - 99 = 78 * 1 + 21
- = GCD(78, 21)
  - 78 = 21 * 3 + 15
- = GCD(21, 15)
  - 21 = 15 * 1 + 6
- = GCD (15, 6)
  - 15 = 6 * 2 + 3
- = GCD(6, 3)
  - 6 = 3 * 2 + 0
- = 3

- 3 = 15 - (2 * 6)
- 6 = 21 - 15
  - 3 = 15 - (2 * (21 - 15))
  - = 15 - (2 * 21) + (2 * 15)
  - = (3 * 15) - (2 * 21)
- 15 = 78 - (3 * 21)
  - 3 = (3 * (78 - (3 * 21)))
    - (2 * 21)
  - = (3 * 78) - (11 * 21)
- 21 = 99 - 78
  - 3 = (3 * 78) - (11 * (99 - 78))
  - = (14 * 78) - (11 * 99)
  - = 99 * -11 + 78 * 14

OK, but why?

- This and all of our large integer algorithms will be handy when we look at algorithms for implementing…

CRYPTOGRAPHY

Introduction to crypto

- Cryptography - enabling secure communication in the presence of third parties
  - Alice wants to send Bob a message without anyone else being able to read it

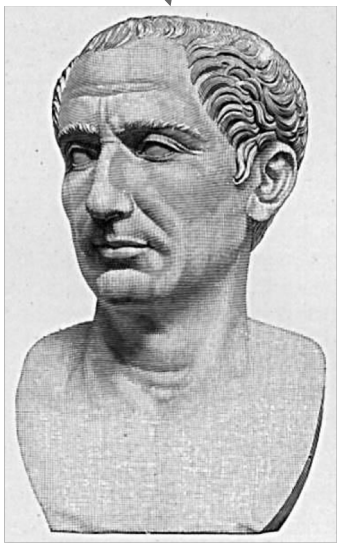
Enter the adversary

- Consider the adversary to be anyone that could try to eavesdrop on Alice and Bob communicating
  - People in the same coffee shop as Alice or Bob as they talk over WiFi
  - Admins operating the network between Alice and Bob
    - And mirroring their traffic to the NSA…
- Will have access to:
  - The *ciphertext*
    - The encrypted message
  - The encryption algorithm
    - At least Alice and Bob should assume the adversary does
- The key material is the only thing Bob knows that the adversary does not

Cryptography has been around for some time

- Early, classic encryption scheme:
  - Caesar cipher:
    - "Shift" the alphabet by a set amount
    - Use this shifted alphabet to send messages
    - The "key" is the amount the alphabet is shifted

Yes, that Caesar



Alphabet

ABCDEFGHIJKLMNOPQRSTUVWXYZ

XYZABCDEFGHIJKLMNOPQRSTUVW

Shift 3

By modern standards, incredibly easy to crack

- BRUTE FORCE
  - Try every possible shift
    - 25 options for the English alphabet
    - 255 for ASCII
- OK, let's make it harder to brute force
  - Instead of using a shifted alphabet, let's use a random permutation of the alphabet
    - Key is now this permutation, not just a shift value
  - R size alphabet means R! possible permutations!

By modern standards, incredibly easy to crack

- Just requires a bit more sophisticated of an algorithm
- Analyzing encrypted English for example
  - Sentences have a given structure
  - Character frequencies are skewed
  - Essentially playing Wheel of Fortune

So what is a good cipher?

- One-time pads
  - List of one-time use keys (called a *pad*) here
- To send a message:
  - Take an unused pad
  - Use modular addition to combine key with message
    - For binary data, XOR
  - Send to recipient
- Upon receiving a message:
  - Take the next pad
  - Use modular subtraction to combine key with message
    - For binary data, XOR
  - Read result
- Proven to provide perfect secrecy

One-time pad example

Encoding:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Message: H E L L O

Pad: Q J C W T

	7	4	11	11	14	
+	16	9	2	22	19	(mod 26)
	23	13	13	7	7	

Encrypted Message: X N N H H

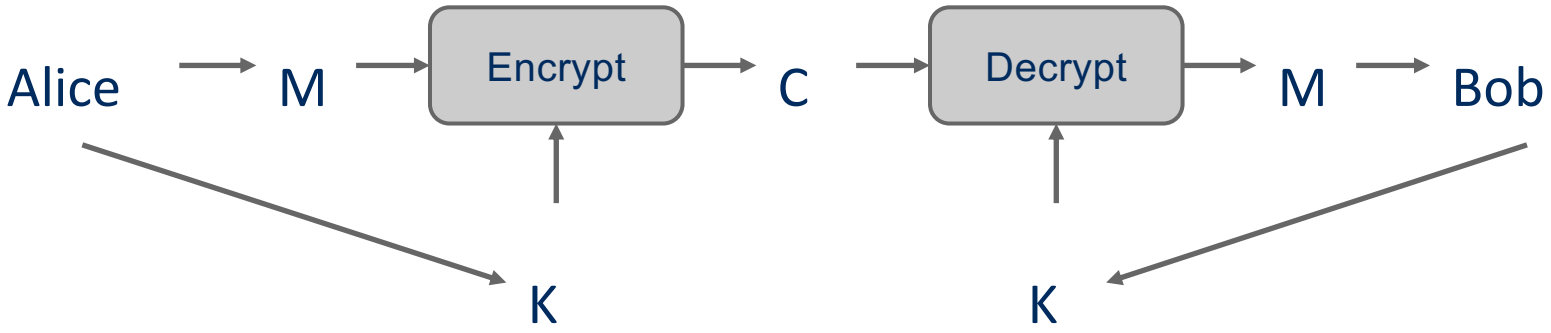
	23	13	13	7	7	
-	16	9	2	22	19	(mod 26)
	7	4	11	11	14	

H E L L O

Difficulties with one-time pads

- Pads must be truly random
- Both sender and receiver must have a matched list of pads in the appropriate order
- Once you run out of pads, no more messages can be sent

Symmetric ciphers



- E.g., DES, AES, Blowfish
- Users share a single *key*
  - Numbers of a given bitlength (e.g., 128, 256)
  - Key is used to encrypt/decrypt many messages back and forth
- Encryptions/decryptions will be fast
  - Typically linear in the size the input
- Ciphertext should appear random
- Best way to recover plaintext should be a brute force attack on the encryption key
  - Which we have shown to be infeasible for 128bit AES keys

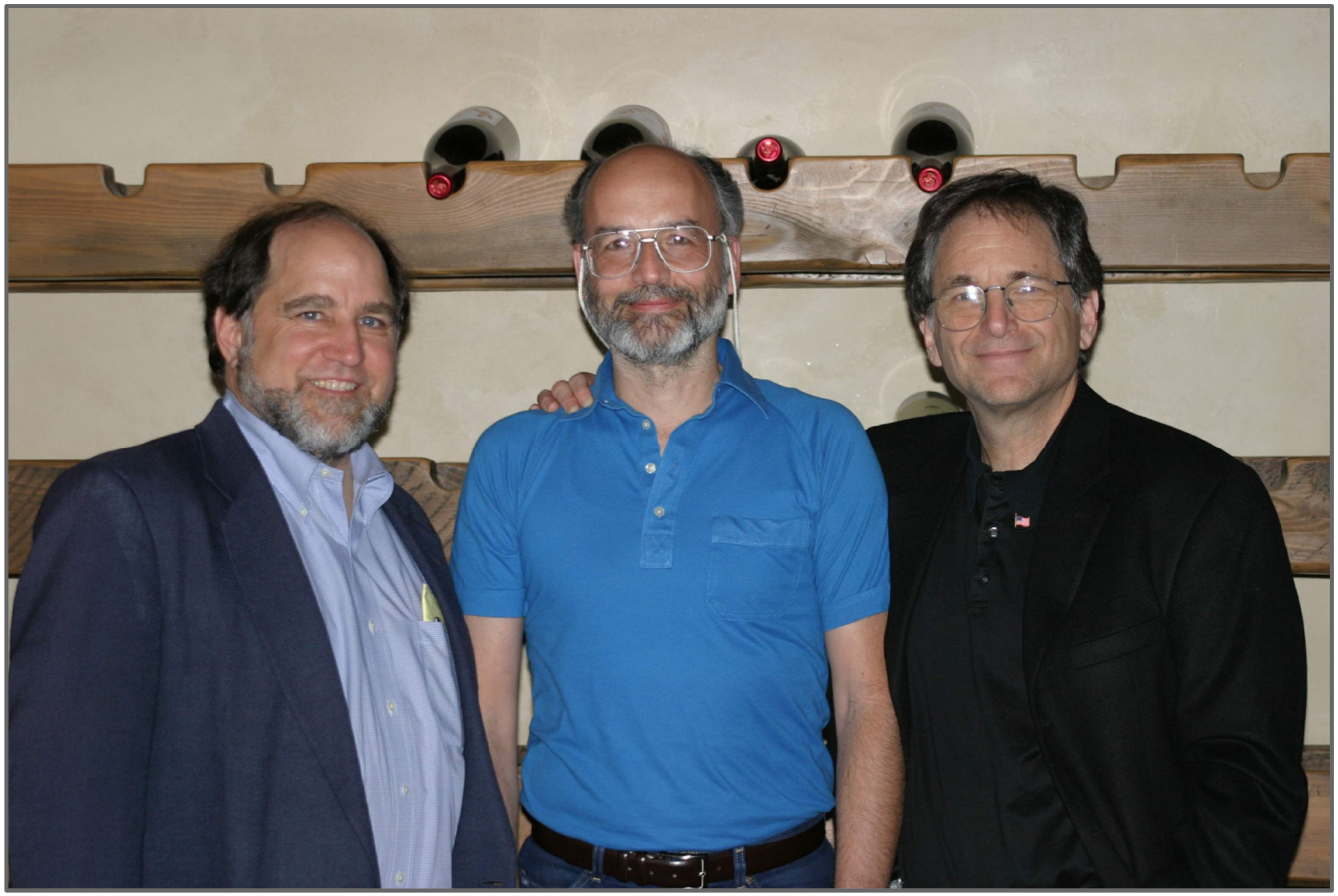
Problems with symmetric ciphers

- Alice and Bob have to both know the same key
  - How can you securely transmit the key from Alice to Bob?
- Further, if Alice also wants to communicate with Charlie, her and Charlie will need to know the same key, a different key from the key Alice shares with Bob
  - Alice and Danielle will also have to share a different key…
  - etc.

Enter public-key encryption

- Each user has their own pair of keys
  - A *public* key that can be revealed to anyone
  - A *private* key that only they should know
- How does this solve our problem?
  - Public key can simply be published/advertised
    - Posted repositories of public keys
    - Added to an email signature
  - Each user is responsible only for their own keypair
- Let's look at a public-key crypto scheme in detail...

RSA



RSA Cryptosystem in-depth

- What are RSA keypairs?
- How messages encrypted?
- How are messages decrypted?
- How are keys generated?
- Why is it secure?

RSA keypairs

- *Public* key is two numbers, which we will call n and e
- *Private* key is a single number we will call d
- The length of n in bits is the key length
  - I.e., 2048 bit RSA keys will have a 2048 bit n value
    - Note that "n" will be used to indicate the RSA public key component for our discussion of RSA...

Encryption

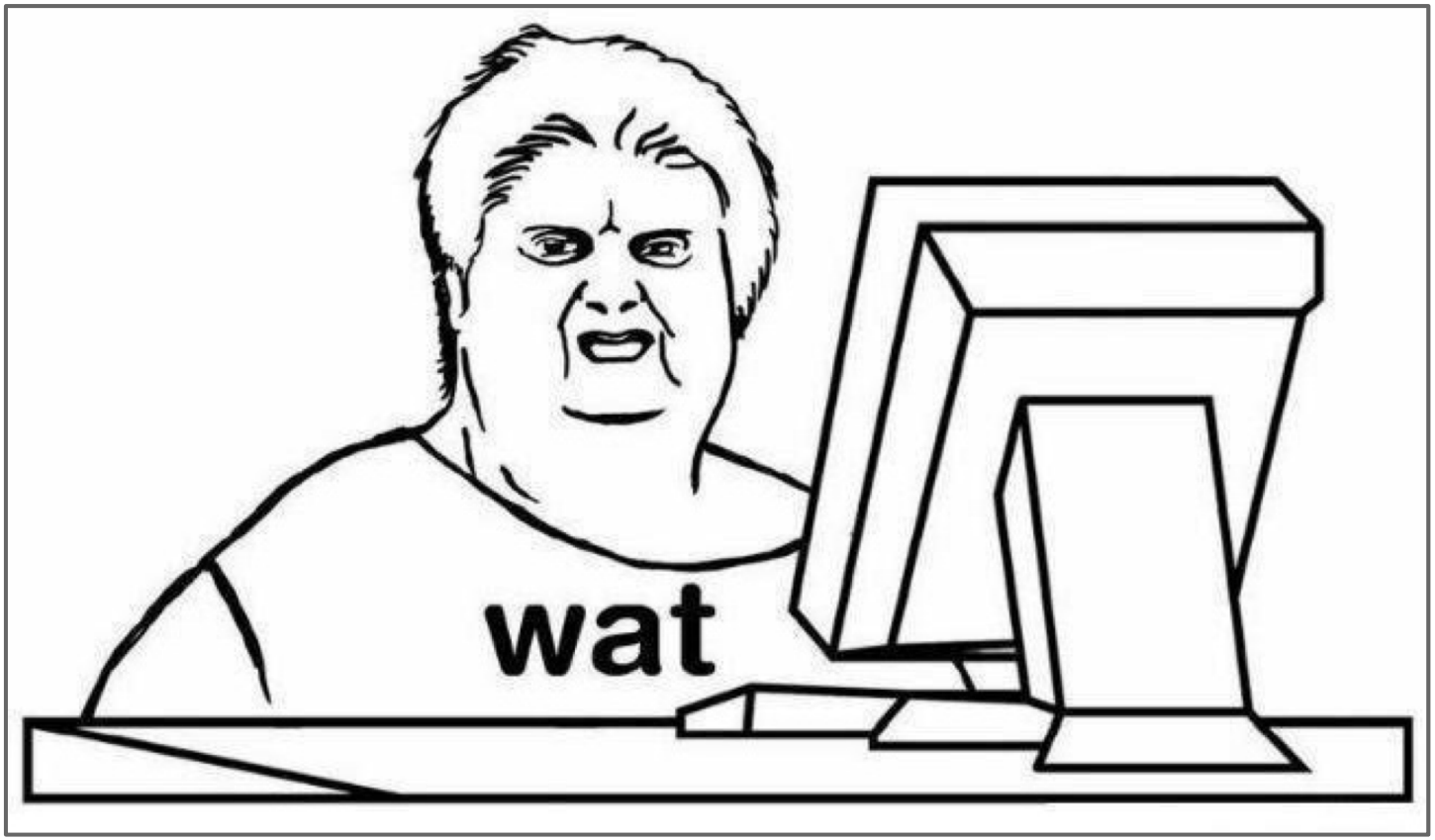
Say Alice wants to send a message to Bob

1. Looks up Bob's public key
2. Convert the message into an integer: m
3. Compute the ciphertext c as:
    - $c = m^e \pmod{n}$
4. Send c to Bob

Decryption

Bob can simply:

1. Compute m as:
    a. $m = c^d \pmod{n}$
2. Convert m into Alice's message


wat

n, e, and d need to be carefully generated

1. Choose two prime numbers p and q
2. Compute n = p * q
3. Compute $\phi(n)$
   - $\phi(n) = \phi(p) * \phi(q) = (p - 1) * (q - 1)$
4. Choose e such that
   - $1 < e < \phi(n)$
   - $GCD(e, \phi(n)) = 1$
     - I.e., e and $\phi(n)$ are co-prime
5. Determine d as $d = e^{-1} \bmod(\phi(n))$

What the ϕ?

- Here, we mean ϕ to be Euler's totient
- ϕ(n) is a count of the integers < n that are co-prime to n
  - I.e., how many k are there such that:
    - 1 <= k <= n AND GCD(n, k) = 1
- p and q are prime..
  - Hence, ϕ(p) = p - 1 and ϕ(q) = q -1
- Further, ϕ is multiplicative
  - Since p and q are prime, they are co-prime, so
    - ϕ(p) * ϕ(q) = ϕ(p * q) = ϕ(n)
      - I won't detail the proof here...

OK, now what about multiplicative inverses mod ϕ(n)?

- $d = e^{-1} \bmod(\phi(n))$
- Means that $d = 1/e \bmod(\phi(n))$
- Means that $e * d = 1 \pmod{\phi(n)}$
- Now, *this* can be equivalently stated as $e * d = z * \phi(n) + 1$
  - For some z
- Can further restate this as: $e * d - z * \phi(n) = 1$
- Or similarly: $1 = \phi(n) * (-z) + e * d$
- How can we solve this?
  - Hint: recall that we know $GCD(\phi(n), e) = 1$

Use extended Euclidean algorithm!

- GCD(a, b) = i = ax + by
- Let:
  - a = $\phi(n)$
  - b = e
  - x = -z
  - y = d
  - i = 1
- GCD($\phi(n)$, e) = 1 = $\phi(n)$ * (-z) + e * d
- We can compute d in linear time!

RSA keypair example notes

- p and q must be prime
- $n = p * q$
- $\phi(n) = (p - 1) * (q - 1)$
- Choose e such that
  - $1 < e < \phi(n)$ and $GCD(e, \phi(n)) = 1$
- Solve $XGCD(\phi(n), e) = 1 = \phi(n) * (-z) + e * d$
- Compute the ciphertext c as:
  - $c = m^e \pmod n$
- Recover m as:
  - $m = c^d \pmod n$

OK, but how does $m^{ed} = m \bmod n$?

- Feel free to look up the proof using Fermat's little theorem
  - Knowing this proof is **NOT** required for the course
  - Knowing how to generate RSA keys and encrypt/decrypt **IS**
- For this course, we'll settle with our example showing that it *does* work

Why is RSA secure?

- 4 avenues of attack on the math of RSA were identified in the original paper:
  - Factoring n to find p and q
  - Determining $\phi(n)$ without factoring n
  - Determining d without factoring n or learning $\phi(n)$
  - Learning to take e^{th} roots modulo n

Factoring n

- To the best of our knowledge, this is *hard*
  - A 768 bit RSA key was factored one time using the best currently known algorithm
    - Took 1500 CPU years
      - 2 years of real time on hundreds of computers
    - Hence, large keys are pretty safe
      - 2048 bit keys are a pretty good bet for now

What about determining $\phi(n)$ without factoring n?

- Would allow us to easily compute d because $ed = 1 \bmod \phi(n)$
- Note:
  - $\phi(n) = n - p - q + 1$
    - $\phi(n) = n - (p + q) + 1$
    - $(p + q) = n + 1 - \phi(n)$
  - $(p + q) - (p - q) = 2q$
  - Now we just need $(p - q)$...
    - $(p - q)^2 = p^2 - 2pq + q^2$
    - $(p - q)^2 = p^2 + 2pq + q^2 - 4pq$
    - $(p - q)^2 = (p + q)^2 - 4pq$
    - $(p - q)^2 = (p + q)^2 - 4n$
    - $(p - q) = \sqrt{((p + q)^2 - 4n)}$
- If we can figure out $\phi(n)$ efficiently, we could factor n efficiently!

Determining d without factoring n or learning ϕ(n)?

- If we know, d, we can get a multiple of ϕ(n)
  - ed = 1 mod ϕ(n)
  - ed = kϕ(n) + 1
    - For some k
  - ed - 1 = kϕ(n)
- It has been shown that n can be efficiently factored using any multiple of ϕ(n)
  - Hence, this would provide another efficient solution to factoring!

Learning to take e^{th} roots modulo n

- Conjecture was made in 1978 that breaking RSA would yield an efficient factoring algorithm
  - To date, it has been not been proven or disproven

This all leads to the following conclusion

- Odds are that breaking RSA efficiently implies that factoring can be done efficiently.
- Since factoring is probably hard, RSA is probably safe to use.

Implementation concerns

- Encryption/decryption:
  - How can we perform efficient exponentiations?
- Key generation:
  - We can do multiplication, XGCD for large integers
  - What about finding large prime numbers?

Efficient exponentiation for RSA

Does this solve our problems?

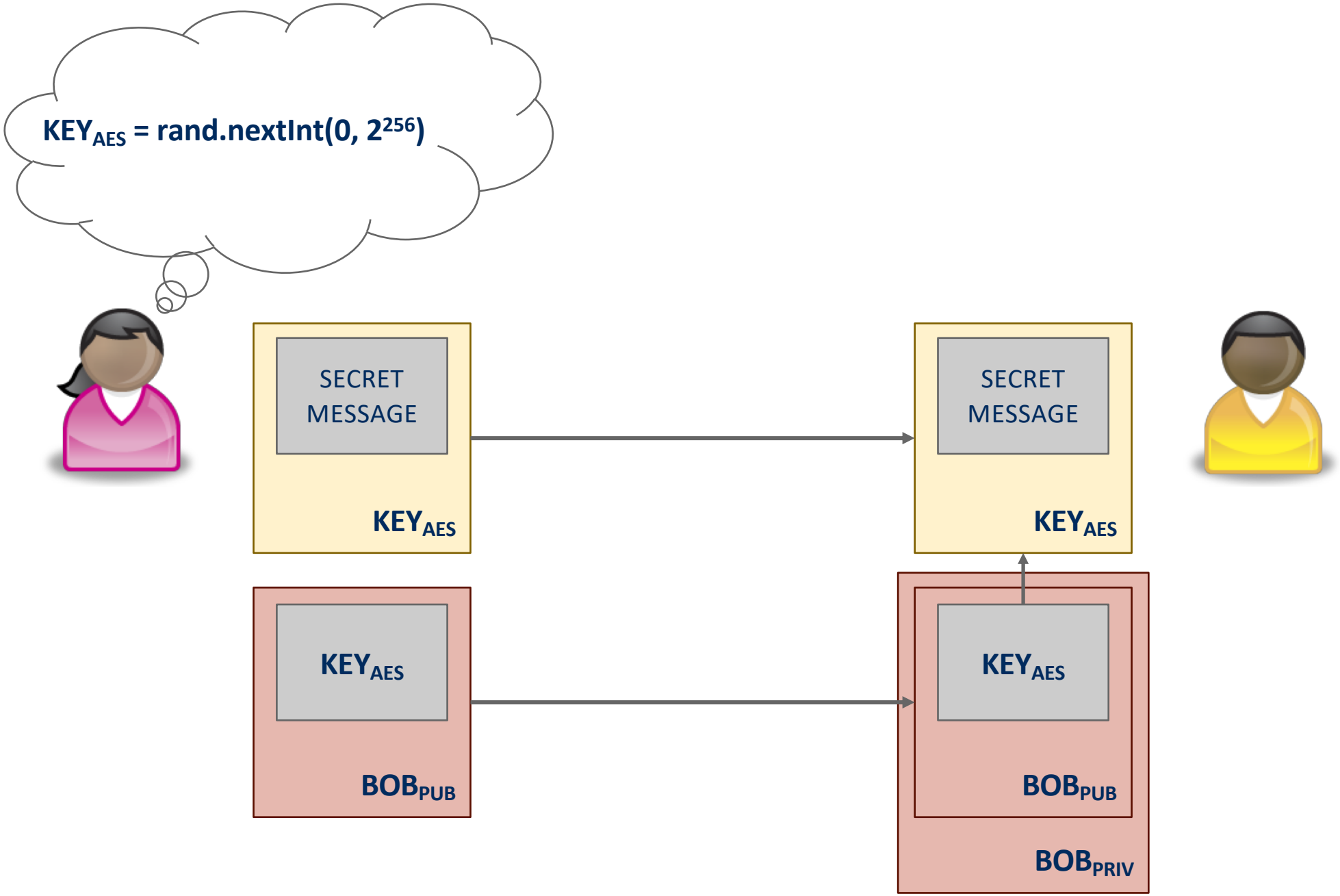
```
ans = 1
foreach bit in y:
        ans = (ans^2 mod n)
        if bit == 1:
                ans = (ans * x
mod n)x
```

- How can we improve runtime for RSA exponentiations?
  - Don't actually need x^y
    - Just need (x^y mod n)

Still slower (generally) than symmetric encryption

- If only we could have the speed of symmetric encryption without the key distribution woes!
  - What if we transmitted symmetric crypto keys with RSA?
    - RSA Envelopes!
- Going back to Alice and Bob
  - Alice generates a random AES key
  - Alice encrypts her message using AES with this key
  - Alice encrypts the key using Bob's RSA public key
  - Alice sends the encrypted message and encrypted key to Bob
  - Bob decrypts the AES key using his RSA private key
  - Bob decrypts the message using the AES key

RSA Envelope example



Prime testing option 1: BRUTE FORCE

- Try all possible factors of x
  - 1 .. sqrt(x)
    - aka 1 .. sqrt($2^{size(x)}$)
      - For a total of $2^{(size(x)/2)}$ factor checks
- A factor check should take about the same amount of time as multiplication
  - $size(x)^2$
- So our runtime is $\Theta(2^{(size(x)/2)} * size(x)^2)$

Option 2: A probabilistic approach

- Need a method test : Z×Z → {T, F}
  - If test(x, a) = F, x is composite based on the witness a
  - If test(x, a) = T, x is probably prime based on the witness a
- To test a number x for primality:
  - Randomly choose a witness a
    - if test(x, a) = F, x is composite
    - if test(x, a) = T, loop
- Possible implementations of test(x, a):
  - Miller-Rabin, Fermat's, Solovay–Strassen

often probability ≈ 1/2

k repetitions leads to probability that x is composite ≈ $1/2^k$

Another fun use of RSA...

- Notice that encrypting and decrypting are inverses
  - $m^{ed} = m^{de} \pmod{n}$
- We can “decrypt” the message first with a private key
- Then recover the message by “encrypting” with a public key
- Note that anyone can recover the message
  - However, they know the message must have come from the owner of the private key
    - Using RSA this way creates a *digital signature*

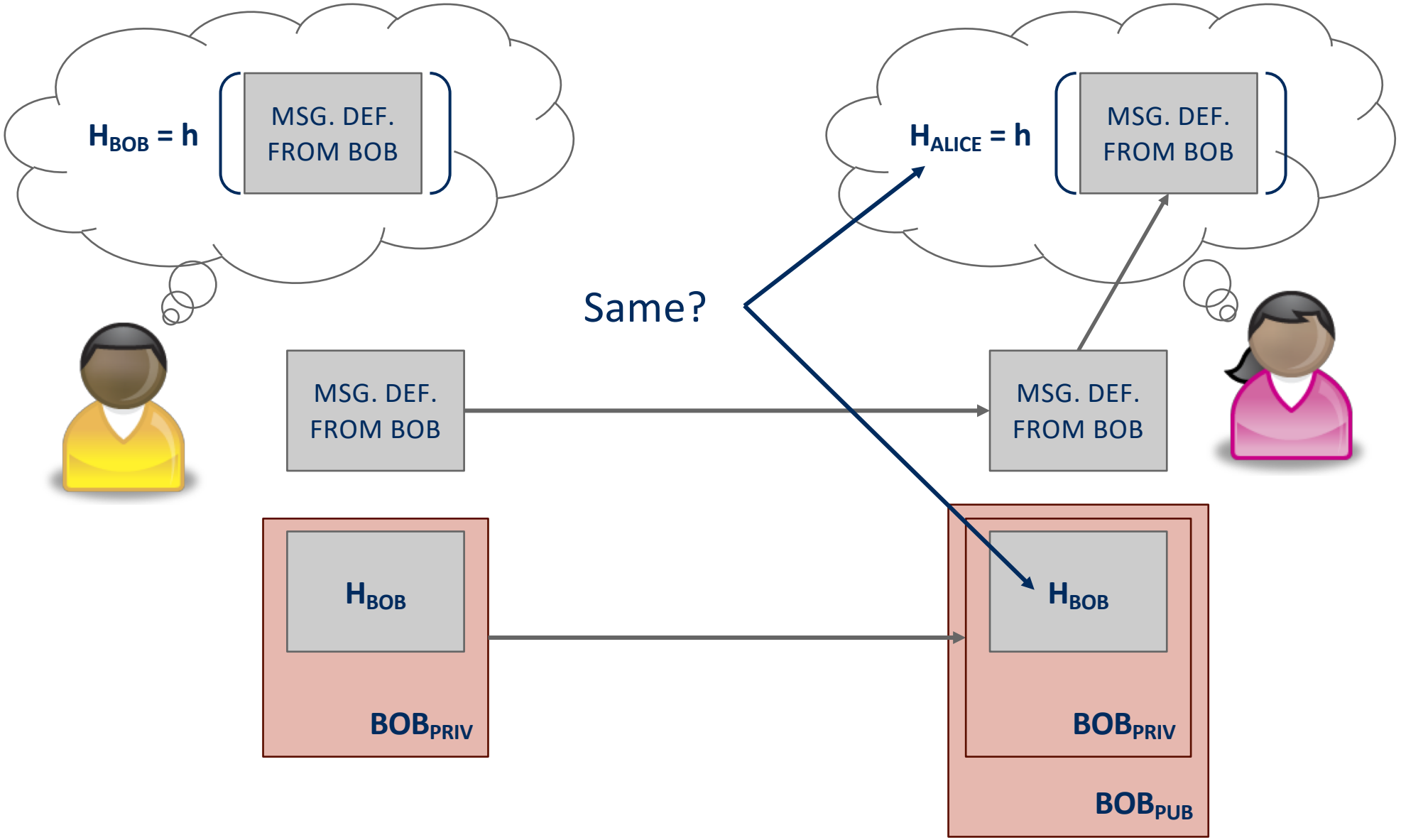
Do RSA signatures need to be slow?

- We encrypted symmetric crypto keys before to speed things up…
  - We'll need another crypto primitive to help out here
  - Cryptographically secure hash functions

Hashing for security (similarities)

- Cryptographically secure hash functions share properties with the hash functions we've already talked about:
  - Map from some input domain to a limited output range
    - Though output ranges are much larger here
      - For modern algorithms 224-512 bit output sizes
  - Time required to compute the hash is proportional to the size of the item being hashed
    - Though, practically, cryptographic hash functions are more expensive

Now just sign a hash of the message!



What about collisions?

- If Bob signs a hash of the message “I’ll see you at 7”
- It could appear that Bob signed any message whose hash collides with “I’ll see you at 7”...
- If h(“I’ll see you at 7”) == h(“I’ll see you after I rob the bank”), Bob could be in alot of trouble
- An attack like this helped the Flame malware to spread
- This is also the reason Google is aiming to deprecate SHA-1

Hashing for security (differences)

- This is why cryptographically secure hash functions must support additional properties:
  - It should be infeasible to find two different messages with the same hash value
  - It should be infeasible to recover a message from its hash
    - Should require a brute force approach
  - Small changes to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value

Public key isn’t perfect

What do you when a private key is compromised?

Final note about crypto

NEVER IMPLEMENT YOUR OWN CRYPTO

Use a trusted and tested library.