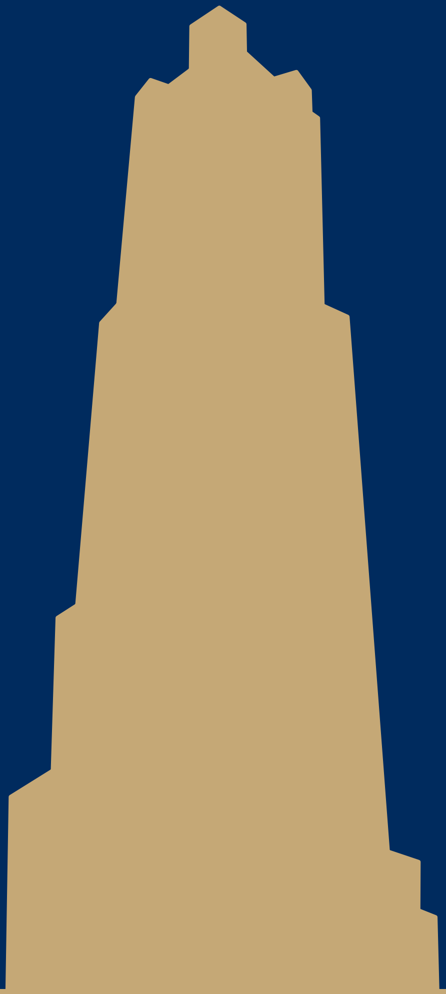
CS/COE 1501

www.cs.pitt.edu/~lipschultz/cs1501/

More Math



Exponentiation

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

```
ans = 1
for i = 1 .. y:
   ans = ans * x
```

- Runtime?
  - It's just a for loop with a single multiplication…

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 is 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}:
  - 134078079299425970995740249982058461274793658205923933777235614437217640300735469768018742981669034276900318581864860508537538828119465699464336490060840 96
    - $= 1.3 * 10^{154}$
- 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
- 8 3GHz cores = 24,000,000,000 cycles/second
  - $(2^{512} / 24000000000) / 3600 * 24 * 365 =$
    - $1.77 * 10^{136}$ years to compute

This is way too long to do exponentiations!

- So how do we do better?
- Let's try divide and conquer!
  - When y is even: $x^y = (x^{(y/2)})^2$
  - When y is odd: $x^y = x * (x^{(y/2)})^2$
- 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:
  - $T(n) = T(n-1) + \Theta((2^{(n-1)} * n)^2)$

squared from the used of the gradeschool algorithm

multiplication input size

Runtime analysis

- Can we use the master theorem?
    - Nope, we don't have a b > 1
- OK, so let's reason it through ...
    - 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 n
    - Halve n until we reach the base case
    - Combine base case results
    - 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

```
res = 1
foreach bit in y:
    res = $res^2$
    if bit == 1:
        res = res * x
```

From most to least significant

Bottom-up exponentiation example

- Consider x^y where x is 3 and y is 43 (computing 3^{43})
- Iterate through the bits of y (43 in binary: 101011)
- res = 1

$\text{res} = 1^2 \quad = 1$

$\quad \text{res} = 1 * x \quad = x$

$\text{res} = x^2 \quad = x^2$

$\text{res} = (x^2)^2 \quad = x^4$

$\quad \text{res} = x^4 * x \quad = x^5$

$\text{res} = (x^5)^2 \quad = x^{10}$

$\text{res} = (x^{10})^2 \quad = x^{20}$

$\quad \text{res} = x^{20} * x \quad = x^{21}$

$\text{res} = (x^{21})^2 \quad = x^{42}$

$\quad \text{res} = x^{42} * x \quad = x^{43}$

Does this solve our problem with mult times?

- Nope, still squaring res 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

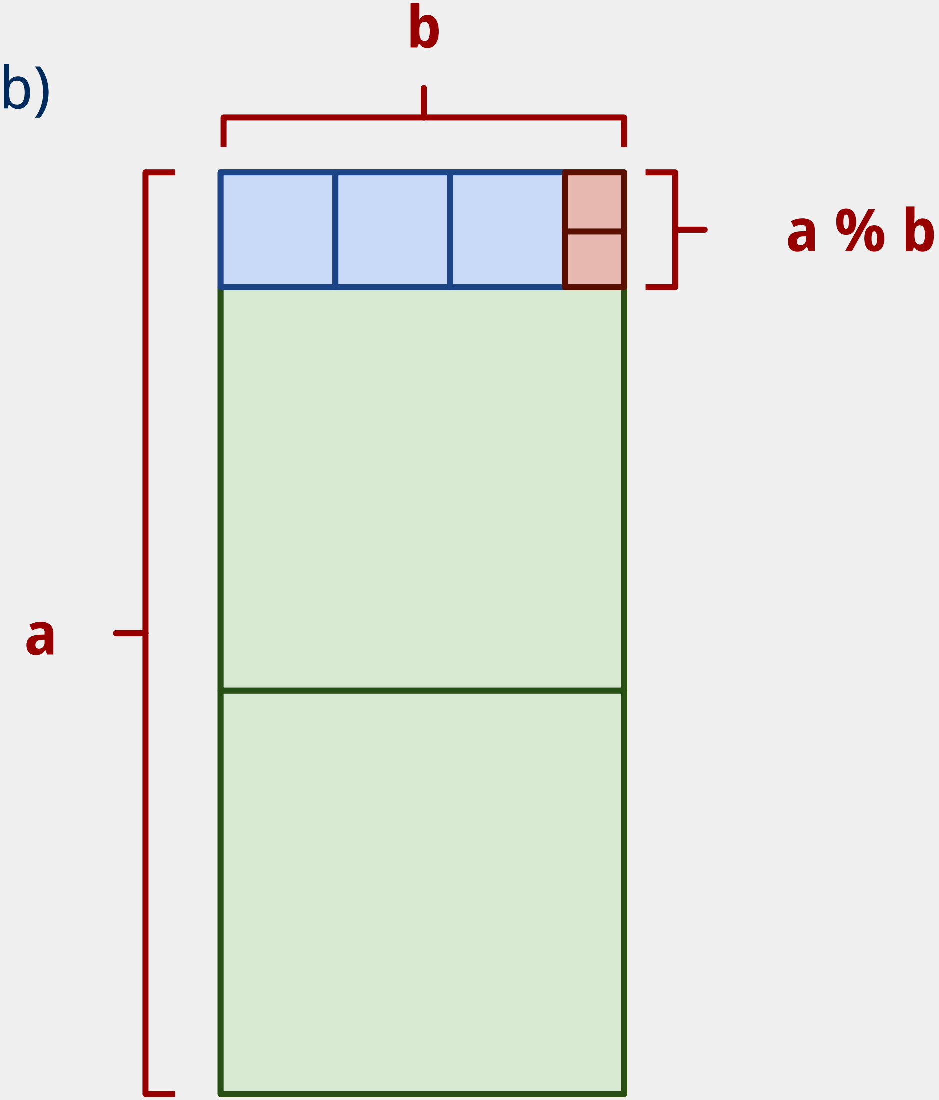
- GCD(a, b)
  - Largest int that evenly divides both a and b
- Easiest approach:
  - BRUTE FORCE

```
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)
  - where a > b
- Repeat until a % b == 0



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

a = b * (a/b) + (a%b)

- = 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

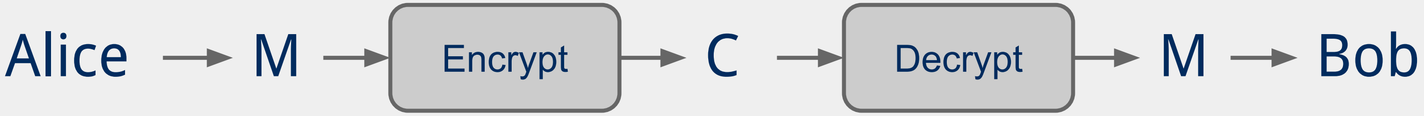
- 3 = 15 - (2 * 6)
- 6 = 21 - 15
  - 3 = 15 - (2 * (21 - 15))
  - = 15 - (2 * 21) + (2 * 15)
  - = (3 * 15) - (2 * 21)
- 15 = 78 - (3 * 21)
  - 3 = (3 * (78 - (3 * 21))) - (2 * 21)
  - = (3 * 78) - (11 * 21)
- 21 = 99 - 78
  - 3 = (3 * 78) - (11 * (99 - 78))
  - = (14 * 78) - (11 * 99)
  - = 99 * -11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



Alice → M → Encrypt → C → Decrypt → M → Bob

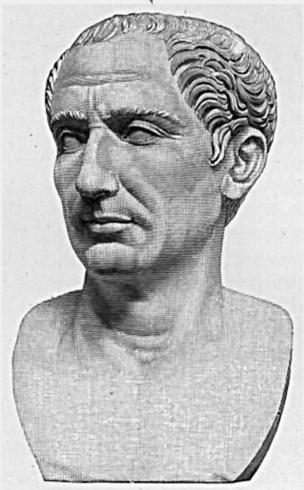
Enter the adversary

- 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 (K) 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

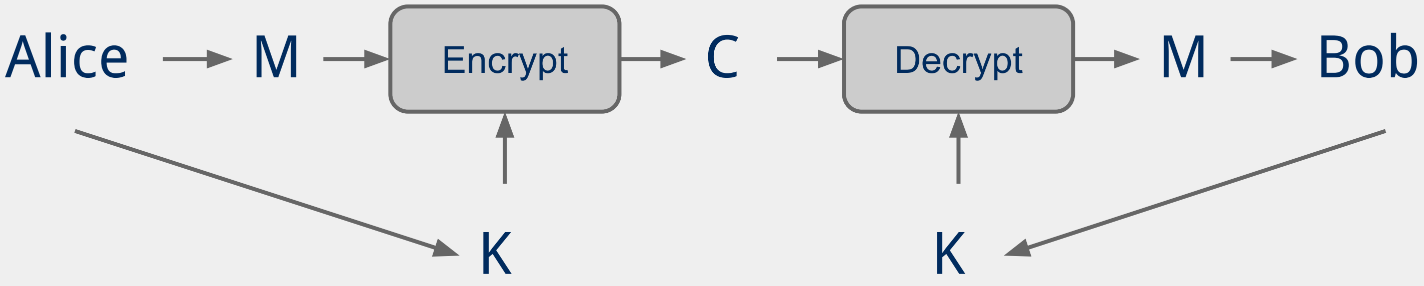
So what is a good cipher?

- 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

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
  - 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.

- Solution next lecture