CS/COE 1501

www.cs.pitt.edu/~nlf4/cs1501/

More Math

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:
  - $T(n) = T(n-1) + \Theta((2^{(n-1)} * n)^2)$

multiplication input size

squared from the used of the gradeschool algorithm

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

$$\begin{aligned}
ans &= 1^2 && = 1 \\
ans &= 1 * x && = x \\
ans &= x^2 && = x^2 \\
ans &= (x^2)^2 && = x^4 \\
ans &= x^4 * x && = x^5 \\
ans &= (x^5)^2 && = x^{10} \\
ans &= (x^{10})^2 && = x^{20} \\
ans &= x^{20} * x && = x^{21} \\
ans &= (x^{21})^2 && = x^{42} \\
ans &= x^{42} * x && = x^{43}
\end{aligned}$$

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

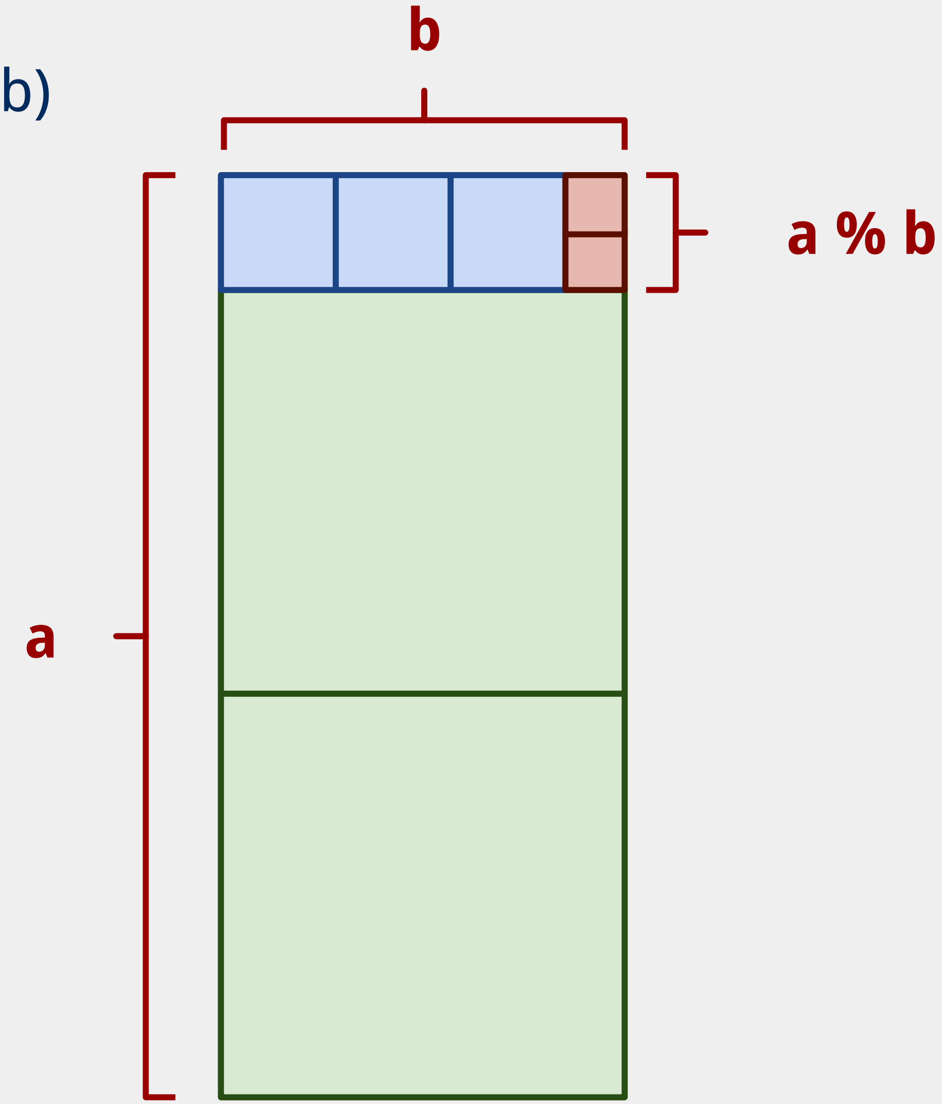
- GCD(a, b)
  - Largest int that evenly divides both a and b
- Easiest approach:
  - BRUTE FORCE

```
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