CS/COE 1501

cs.pitt.edu/~bill/1501/

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
Assuming 512 bit operands, $2^{512}$:
- $134078079299425970995740249982058461274793658205923$
- $933777235614437217640300735469768018742981669034276$
- $900318581864860508537538828119465699464336490060840$
- $96$

Assume we can do multiplications in 1 cycle…
- Which we can’t as we discussed last lecture
And further that these operations are completely parallelizable
- 16 cores at 4 GHz = 64,000,000,000 cycles/second
  - $(2^{512} / 640000000000) / (60 \times 60 \times 24 \times 365) =$
    - $6.64 \times 10^{135}$ years to compute
So how do we do better?

Let’s try divide and conquer!

\[ x^y = (x^{(y/2)})^2 \]

- ...when \( y \) is even; \( (x^{(y/2)})^2 \times x \) when \( y \) is odd

Analyzing a recursive approach:

- Base case?
  - When \( y = 1 \), \( x^y \) is \( x \); when \( y = 0 \), \( x^y \) is 1

- Runtime?
Building another recurrence relation

- \( x^y = (x^{(y/2)})^2 = x^{(y/2)} \times x^{(y/2)} \)
  - Similarly, \((x^{(y/2)})^2 \times x = x^{(y/2)} \times x^{(y/2)} \times x\)

- 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: $x \times x$
  - $n$ bit operands
  - Result size?
    - $2n$
- These results are then multiplied: $x^2 \times x^2$
  - $2n$ bit operands
  - Result size?
    - $4n$ bits
- ...
- $x^{(y/2)} \times x^{(y/2)}$?
  - $(y / 2) \times n$ bit operands = $2^{(n-1)} \times n$ bit operands
  - Result size? $y \times n$ bits = $2^n \times n$ bits
Our recurrence relation looks like:

\[ T(n) = T(n-1) + \Theta((2^{(n-1)} \times n)^2) \]
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?
    ■ \( \log(y) \)
  ○ Further, we know the max value of \( y \)
    ■ Relative to \( n \), that is:
      ● \( 2^n \)
  ○ So, we have, at most \( \log(y) = \log(2^n) = n \) recursions
But we need to do expensive mult in each call

- We need to do $\Theta((2^{(n-1)} \times n)^2)$ 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 $x^y$

```python
ans = 1
foreach bit in y:
    ans = ans^2
    if bit == 1:
        ans = ans * x
```

From most to least significant
Consider $x^y$ where $y$ is 43 (computing $x^{43}$)

- Iterate through the bits of $y$ (43 in binary: 101011)
- $\text{ans} = 1$

\[
\begin{align*}
\text{ans} &= 1^2 = 1 \\
\text{ans} &= 1 \times x = x \\
\text{ans} &= x^2 = x^2 \\
\text{ans} &= (x^2)^2 = x^4 \\
\text{ans} &= x^4 \times x = x^5 \\
\text{ans} &= (x^5)^2 = x^{10} \\
\text{ans} &= (x^{10})^2 = x^{20} \\
\text{ans} &= x^{20} \times x = x^{21} \\
\text{ans} &= (x^{21})^2 = x^{42} \\
\text{ans} &= x^{42} \times x = x^{43}
\end{align*}
\]
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 $g$ that evenly divides both $a$ and $b$ ($g|a$ and $g|b$)
  - Recall that $g|a$ if $\exists i \in \mathbb{Z}$ such that $g \times i = a$

- **Easiest approach:**
  - **BRUTE FORCE**
    
    
    
    ```
    i = min(a, b)
    while(a % i != 0 || b % i != 0):
        i--
    ```

- **Runtime?**
  - $\Theta(\text{min}(a, b))$
  - Linear!
    - … in the value of min(a, b)…
  - Exponential in n
    - Assuming $a, b$ are n-bit integers
Euclid’s algorithm

- \( \text{GCD}(a, b) = \text{GCD}(b, a \% b) \)
Euclidean example 1

- $\text{GCD}(30, 24)$
  - $= \text{GCD}(24, 30 \mod 24)$
  - $= \text{GCD}(24, 6)$
  - $= \text{GCD}(6, 24 \mod 6)$
  - $= \text{GCD}(6, 0)$...
    - Base case! Overall GCD is 6
Euclidean example 2

- $\text{GCD}(99, 78) = 99 = 78 \times 1 + 21$
- $\text{GCD}(78, 21) = 78 = 21 \times 3 + 15$
- $\text{GCD}(21, 15) = 21 = 15 \times 1 + 6$
- $\text{GCD}(15, 6) = 15 = 6 \times 2 + 3$
- $\text{GCD}(6, 3) = 6 = 3 \times 2 + 0$
- $= 3$
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
In addition to the GCD, the Extended Euclidean algorithm (XGCD) produces values x and y such that:

- \( \text{GCD}(a, b) = i = ax + by \)

**Examples:**

- \( \text{GCD}(30, 24) = 6 = 30 \times 1 + 24 \times -1 \)
- \( \text{GCD}(99, 78) = 3 = 99 \times -11 + 78 \times 14 \)

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

CRYPTOGRAPHY