

# MA/CSSE 473

## Day 05

**Fibonacci Revisited**  
**Primes**  
**Modular Arithmetic**



# MA/CSSE 473 Day 04

- HW 2 is due tomorrow
- **Student Questions**
- Asymptotic Analysis example: summation
- Continue Algorithm Overview/Review
  - Revisit Fibonacci run-times taking into account non-constant times for addition and multiplication
  - Integer Primality Testing and Factoring
  - Modular Arithmetic



# Asymptotic Analysis Example

- Find a simple big-Theta expression (as a function of  $n$ ) for the following sum
  - when  $0 < c < 1$
  - when  $c = 1$
  - when  $c > 1$
- $f(n) = 1 + c + c^2 + c^3 + \dots + c^n$



# Recap: Arithmetic Run-times

- For operations on two  $n$ -bit numbers:
- Addition:  $\Theta(n)$
- Multiplication:
  - Standard algorithm:  $\Theta(n^2)$
  - "Gauss-enhanced":  $\Theta(n^{1.59})$ , but with a lot of overhead.
- Division (We won't ponder it in detail, but see next slide):  $\Theta(n^2)$



# Algorithm for Integer Division

```
def divide(x, y):  
    """ Input: Two n-bit integers x and y, where y>=1.  
        Output: The quotient and remainder when x is divided by y."""  
    if x == 0:  
        return 0, 0  
    q, r = divide(x / 2, y)  
    q, r = 2 * q, 2 * r  
    if x % 2 == 1:  
        r = r + 1  
    if r >= y:  
        q, r = q + 1, r - y  
    return q, r
```

Try a couple of examples mentally, to convince yourself that it works



# Recap: Fibonacci Numbers

- Straightforward recursive algorithm:

- Correctness is obvious

- $T(N) \geq F(N)$ , which is exponential ( $\approx 2^{0.69N}$ )

- Algorithm with storage to avoid recomputing:

- Again, correctness is obvious

- And the run-time is linear

- until we dig deeper later

```
def fib1(n):  
    if n==0:  
        return 0  
    if n==1:  
        return 1  
    return fib1(n-1) + fib1(n-2)  
  
print fib1(6), fib1(7), fib1(8)
```

```
def fib2(n):  
    nums = [0]*(n+1)  
    nums[0] = 0  
    nums[1] = 1  
    for i in range(2, n+1):  
        nums[i] = nums[i-1] + nums[i-2]  
    return nums[n]
```



# A Creative $O(\log N)$ Algorithm?

- Let  $X$  be the matrix  $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$
- Then  $\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- also  $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, \text{ and } \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$
- How many additions and multiplications of numbers are necessary to multiply two  $2 \times 2$  matrices?
- If  $n = 2^k$ , how many matrix multiplications does it take to compute  $X^n$ ?
  - $O(\log n)$ , it seems.
- But there is a catch!



# Reconsider our Fibonacci algorithms

- Refine  $T(n)$  calculations, (the time for computing the  $n^{\text{th}}$  Fibonacci number) for each of our three algorithms
  - **Recursive (fib1)**
    - We originally had  $T(n) \in \Theta(F(n)) \approx \Theta(2^{0.694n})$
    - We assumed that addition was constant time.
    - Since each addition is  $\Theta(n)$ , the whole thing is  $\Theta(n \cdot F(n))$
  - **Array (fib2)**
    - We originally had  $T(n) \in \Theta(n)$ , because of  $n$  additions.
    - Since each addition is  $\Theta(n)$ , the whole thing is  $\Theta(n^2)$
  - **Matrix multiplication approach (fib3)**
    - We saw that  $\Theta(\log n)$   $2 \times 2$  matrix multiplications work.
    - Let  $M(n)$  be the time required to multiply two  $n$ -bit numbers.  $M(n) = \Theta(n^a)$  for some  $a$  with  $1 \leq a \leq 2$ .
    - It's easy to see that  $T(n) \in O(M(n) \log n)$
    - Can we show that  $T(n) \in O(M(n))$  ?
      - Do it for  $a = 2$  and  $a = \log_2(3)$
      - If the multiplication of numbers is better than  $O(n^2)$ , so is finding the  $n^{\text{th}}$  Fibonacci number.



# FACTORING and PRIMALITY

- Two important problems
  - FACTORING: Given a number  $N$ , express it as a product of its prime factors
  - PRIMALITY: Given a number  $N$ , determine whether it is prime
- Conclusions
  - Factoring is hard
    - The best algorithms so far require time that is exponential in the number of bits of  $N$
  - Primality testing is relatively easy
  - A strange disparity for these closely related problems
  - Exploited by cryptographic algorithms
- More on these problems later



# Modular arithmetic definitions

- **x modulo N** is the remainder when x is divided by N. I.e.,
  - If  $x = qN + r$ , where  $0 \leq r < n$  (**q and r are unique!**),
  - then **x modulo N** is equal to r.
- x and y are **congruent modulo N**, which is written as  $x \equiv y \pmod{N}$ , if and only if N divides (x-y).
  - i.e., there is an integer k such that  $x - y = kN$ .
  - In a context like this, **a divides b** means "divides with no remainder", i.e. "a is a factor of b."
  - Example:  $253 \equiv 13 \pmod{60}$



# Modular arithmetic properties

- Substitution rule
  - If  $x \equiv x' \pmod{N}$  and  $y \equiv y' \pmod{N}$ ,  
then  $x + y \equiv x' + y' \pmod{N}$ , and  $xy \equiv x'y' \pmod{N}$
- Associativity
  - $x + (y + z) \equiv (x + y) + z \pmod{N}$
- Commutativity
  - $xy \equiv yx \pmod{N}$
- Distributivity
  - $x(y+z) \equiv xy + yz \pmod{N}$



# Modular Addition and Multiplication

- To **add** two integers  $x$  and  $y$  modulo  $N$  (where  $n = \lceil \log N \rceil$  (the number of bits in  $N$ ), begin with regular addition.
  - $x$  and  $y$  are in the range \_\_\_\_\_, so  $x + y$  is in range \_\_\_\_\_
  - If the sum is greater than  $N-1$ , subtract  $N$ .
  - Run time is  $\Theta( )$
- To **multiply**  $x$  and  $y$ , begin with regular multiplication, which is quadratic in  $n$ .
  - The result is in range \_\_\_\_\_ and has at most \_\_\_\_\_ bits.
  - Compute the remainder when dividing by  $N$ , quadratic time. So entire operation is  $\Theta( )$



# Modular Addition and Multiplication

- To **add** two integers  $x$  and  $y$  modulo  $N$  (where  $n = \lceil \log N \rceil$ , begin with regular addition.
  - $x$  and  $y$  are in the range **0 to  $N-1$** , so  $x + y$  is in range **0 to  $2N-1$**
  - If the sum is greater than  $N-1$ , subtract  $N$ .
  - Run time is  $\Theta(n)$
- To **multiply**  $x$  and  $y$ , begin with regular multiplication, which is quadratic in  $n$ .
  - The result is in range 0 to  **$(N-1)^2$**  and has at most  **$2n$**  bits.
  - Then compute the remainder when dividing by  $N$ , quadratic time in  $n$ . So entire operation is  $\Theta(n^2)$



# Modular Exponentiation

- In some cryptosystems, we need to compute  $x^y$  modulo  $N$ , where all three numbers are several hundred bits long. Can it be done quickly?
- Can we simply take  $x^y$  and then figure out the remainder modulo  $N$ ?
- Suppose  $x$  and  $y$  are only 20 bits long.
  - $x^y$  is at least  $(2^{19})^{(2^{19})}$ , which is about 10 million bits long.
  - Imagine how big it will be if  $y$  is a 500-bit number!
- To save space, we could repeatedly multiply by  $x$ , taking the remainder modulo  $N$  each time.
  - If  $y$  is 500 bits, then there would be  $2^{500}$  multiplications.
  - This algorithm is exponential in the length of  $y$ .
  - Ouch!

