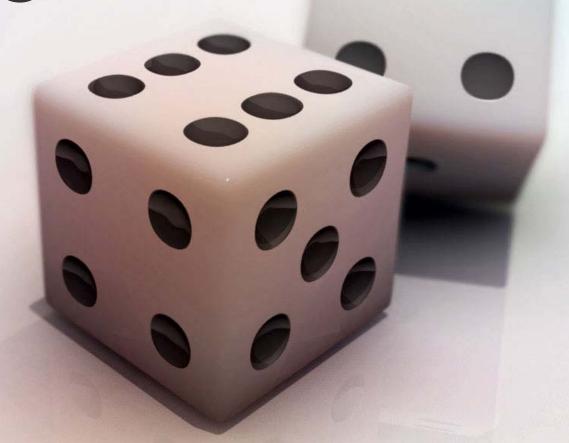# MA/CSSE 473 Day 05

**Factors and Primes
Recursive division
algorithm**

MA/CSSE 473 Day 05

- HW 2 due tonight, 3 is due Monday

- **Student Questions**

- Asymptotic Analysis example: summation

- Review topics I don't plan to cover in class

- Continue Algorithm Overview/Review

  - Integer Primality Testing and Factoring

  - Modular Arithmetic intro

  - Euclid's Algorithm

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 $c > 1$
- $f(n) = 1 + c + c^2 + c^3 + \ldots + c^n$

Quick look at review topics in textbook

REVIEW THREAD

Textbook Topics I Won't Cover in Class

- **Chapter 1 topics** that I will not discuss in detail unless you have questions. They should be review For some of them, there will be review problems in the homework
  - Sieve of Eratosthenes (all primes less than n)
  - Algorithm Specification, Design, Proof, Coding
  - Problem types : sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, numerical problems
  - Data Structures: ArrayLists, LinkedLists, trees, search trees, sets, dictionaries,

Textbook Topics I Won't Cover*

- Chapter 2
  - Empirical analysis of algorithms should be review
  - I believe that we have covered everything else in the chapter except amortized algorithms and recurrence relations
  - We will discuss amortized algorithms
  - Recurrence relations are covered in CSSE 230 and MA 375. We'll review particular types as we encounter them.

*Unless you ask me to

Textbook Topics I Won't Cover*

- Chapter 3 - Review
  - Bubble sort, selection sort, and their analysis
  - Sequential search and simple string matching

*Unless you ask me to

Textbook Topics I Won't Cover*

- Chapter 4 - Review
  - Mergesort, quicksort, and their analysis
  - Binary search
  - Binary Tree Traversal Orders (pre, post, in, level)

*Unless you ask me to

Textbook Topics I Won't Cover*

- Chapter 5 - Review
  - Insertion Sort and its analysis
  - Search, insertion, delete in Binary Tree
  - AVL tree insertion and rebalance

*Unless you ask me to

Efficient Fibonacci 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 $\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = X \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = X^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}, and \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = X^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$

- How many additions and multiplications of numbers are necessary to multiply two 2x2 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!

Hidden because not ready for prime time yet.

- Refine T(n) calculations, (the time for computing the nth 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 $\Theta(\log n)$ 2x2 matrix multiplications give us F_n.
    - Let M(k) be the time required to multiply two k-bit numbers. $M(k) \in \Theta(k^a)$ for some a with $1 \le a \le 2$.
    - It's easy to see that $T(n) \in O(Ml(n) \log n)$
    - 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 nth Fibonacci number.

Integer Division

Modular arithmetic

Euclid's Algorithm

Heading toward Primality Testing

ARITHMETIC THREAD

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
- **Where we will go with this eventually**
  - Factoring is hard
    - The best algorithms known so far require time that is exponential in the number of bits of N
  - Primality testing is comparatively easy
  - A strange disparity for these closely-related problems
  - Exploited by cryptographic algorithms
- **More on these problems later**
  - First, more math and computational background…

Recap: Arithmetic Run-times

- For operations on two k-bit numbers:

- Addition: $\Theta(k)$

- Multiplication:

  - Standard algorithm: $\Theta(k^2)$

  - "Gauss-enhanced": $\Theta(k^{1.59})$, but with a lot of overhead.

- Division (We won't ponder it in detail, but see next slide): $\Theta(k^2)$

Algorithm for Integer Division

```python
def divide(x, y):
    """ Input: Two non-negative 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)    # max recursive calls:
    q, r = 2 * q, 2 * r         #    number of bits in x
    if x % 2 == 1:
        r = r + 1
    if r >= y:                  # note that all of the multiplications
        q, r = q + 1, r - y     # and divisions are by 2:
    return q, r                 #       simple bit shifts
```

Let's work through divide(19, 4).

Analysis?

Modular arithmetic definitions

- **x modulo N** is the remainder when x is divided by N. I.e.,
  - If x = qN + r, where 0 ≤ 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≡y (mod 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: $253 \equiv 13$ (mod 60)

Modular arithmetic properties

- Substitution rule

  - If $x \equiv x'$ (mod N) and $y \equiv y'$ (mod N),
    then $x + y \equiv x' + y'$ (mod N), and $xy \equiv x'y'$ (mod N)

- Associativity

  - $x + (y + z) \equiv (x + y) + z$ (mod N)

- Commutativity

  - $xy \equiv yx$ (mod N)

- Distributivity

  - $x(y+z) \equiv xy + yz$ (mod N)

Modular Addition and Multiplication

- To **add** two integers x and y modulo N (where k = $\lceil$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 Θ ( )
- To **multiply** x and y modulo N, begin with regular multiplication, which is quadratic in k.
  - The result is in range _____ and has at most _____ bits.
  - Compute the remainder when dividing by N, quadratic time. So entire operation is Θ( )

Modular Addition and Multiplication

- To **add** two integers x and y modulo N (where k = $\lceil \log N \rceil$, 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 Θ (**k**)

- 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 **2k** bits.
  - Then compute the remainder when dividing by N, quadratic time in k. So entire operation is $\Theta(\mathbf{k^2})$

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} bit multiplications.
  - This algorithm is exponential in the length of y.
  - Ouch!

Modular Exponentiation Algorithm

```python
def modexp(x, y, N):
    if y==0:
        return 1
    z = modexp(x, y/2, N)
    if y%2 == 0:
        return (z*z) % N
    return (x*z*z) % N
```

- Let k be the maximum number of bits in x, y, or N
- The algorithm requires at most ___ recursive calls
- Each call is Θ(   )
- So the overall algorithm is Θ(   )

Modular Exponentiation Algorithm

```python
def modexp(x, y, N):
    if y==0:
        return 1
    z = modexp(x, y/2, N)
    if y%2 == 0:
        return (z*z) % N
    return (x*z*z) % N
```

- Let n be the maximum number of bits in x, y, or N
- The algorithm requires at most **k** recursive calls
- Each call is $\Theta(\mathbf{k^2})$
- So the overall algorithm is $\Theta(\mathbf{k^3})$