MA/CSSE 473 Day 05

Factors and Primes
Recursive division algorithm

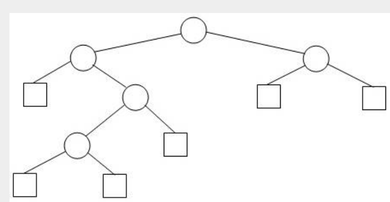
MA/CSSE 473 Day 05

- **Student Questions**
- One more proof by strong induction
- List of review topics I don't plan to cover in class
- Continue Arithmetic Algorithms
  - Toward Integer Primality Testing and Factoring
  - Efficient Integer Division Algorithm
  - Modular Arithmetic intro

Quick look at review topics in textbook

REVIEW THREAD

Another Induction Example
Extended Binary Tree (EBT)



- An Extended Binary tree is either
  - an ***external node***, or
  - an (**internal**) root node and two EBTs T_L and T_R.
- We draw internal nodes as circles and external nodes as squares.
  - Generic picture and detailed picture.
- This is simply an alternative way of viewing binary trees, in which we view the null pointers as “places” where a search can end or an element can be inserted.

A property of EBTs

- **Property** P(N): For any N>=0, any EBT with N internal nodes has _______ external nodes.
- **Proof by strong induction**, based on the recursive definition.
  – A notation for this problem: IN(T), EN(T)
  – Note that, like some other simple examples, this one can also be done without induction.
  – But the purpose of this exercise is practice with strong induction, especially on binary trees.
- What is the crux of any induction proof?
  – Finding a way to relate the properties for larger values (in this case larger trees) to the property for smaller values (smaller trees). **Do the proof now**.

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 later.
  - 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, insert, delete in Binary Search treeTree
  - AVL tree insertion and rebalance
    - We *will* review the analysis of AVL trees.

***Unless you ask me to**

Interlude

Heading toward Primality Testing

Integer Division

Modular arithmetic

Euclid's Algorithm

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 systems
- **More on these problems later**
  - First, some 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?

This idea has many uses

In this course we will use it for encryption and for primality testing

MODULAR ARITHMETIC

Modular arithmetic definitions

- **x modulo N** (written as x % N in many programming languages) 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 ≡ 13 (mod 60),
  253 ≡ 373 (mod 60)

Modular arithmetic properties

- Substitution rule
  - If x ≡ x' (mod N) and y ≡ y' (mod N),
    then x + y ≡ x' + y' (mod N), and xy ≡ x'y' (mod N)
- Associativity
  - x + (y + z) ≡ (x + y) + z (mod N)
- Commutativity
  - xy ≡ yx (mod N)
- Distributivity
  - x(y+z) ≡ 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.
  - Assume that x and y are in the range_____, so x + y is in range ________
  - If the sum is greater than N-1, subtract N.
  - Running 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 by doing regular addition.
  - x and y are in the range **0 to N-1**, so x + y is in range **0 to 2N-2**
  - If the sum is greater than N-1, subtract N, else return x + y
  - Run time is Θ (**k**)
- To **multiply** x and y, begin with regular multiplication, which is quadratic in k.
  - The result is in range **0 to $(N-1)^2$** so has at most **2k** bits.
  - Then compute the remainder when xy dividing by N, quadratic time in k. So entire operation is Θ(**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

```
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 $\Theta(\quad)$
- So the overall algorithm is $\Theta(\quad)$

Modular Exponentiation Algorithm

```
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 $\Theta(\mathbf{k^3})$