Math Review
Why do we need math in a data structures course?

- **To Analyze** data structures and algorithms
  - Deriving formulae for time and memory requirements
  - Will the solution scale?
  - *Quantify* the results
- Proving algorithm correctness
Definition: Let $T(n)$ denote the time take by an algorithm on an input of size $n$.

Examples – how much “time” does each of these algorithms take?

// Assume $A$ is an integer array of size $n$

$\begin{align*}
\text{Algorithm 1 (A,n)} & \quad T(n) \approx n \\
& \quad \text{max} = -\infty \\
& \quad \text{for } (i=1 \text{ to } n) \{} \\
& \quad \quad \text{if } (A[i]>\text{max}) \quad \text{max}=A[i]; \\
& \quad \text{Output } \text{max}; \\
\end{align*}$

$\begin{align*}
\text{Algorithm 2 (A, start, end)} & \quad T(n) \\
& \quad \text{if } (n<2) \quad \text{return} \\
& \quad \quad \text{mid} = \text{floor}(n/2) \\
& \quad \quad \text{if } (\text{condition#1}) \\
& \quad \quad \quad \text{Algorithm 2 (A,1,mid)} \quad T(n/2) \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{Algorithm 2 (A,mid+1,n)} \quad T(n/2) \\
& \quad \Rightarrow T(n) \approx T(n/2) + \text{const.} \\
\end{align*}$

$\begin{align*}
\text{Algorithm 3 (A,n)} & \quad T(n) \\
& \quad \text{if } (n<2) \quad \text{return} \\
& \quad \quad x = \text{floor}(n/2) \\
& \quad \quad \text{Algorithm 3 (A,1,x)} \quad T(n/2) \\
& \quad \quad \text{Algorithm 3 (A,x+1,n)} \quad T(n/2) \\
& \Rightarrow T(n) = 2 \cdot T(n/2) + \text{const.} \\
\end{align*}$

These are not a closed form yet.
If you solve the recurrences, you will get,
$O(\lg n)$ for Algorithm 2, and
$O(n)$ for Algorithm 3.
Example

- Consider Algorithm 1 that divides the input array in half and calls Algorithm 1 recursively on each half

```plaintext
Algorithm1 (A,n)
// A is an integer array of size n
if (n<2) return
x = floor(n/2)
Algorithm1 (A,1,x)
Algorithm1 (A,x+1,n)
```

- What is the running time of Algorithm 1?

\[
T(n) = T(n/2) + T(n/2) + \text{const.}
\]

This is not a closed form yet.
Floors and Ceilings

- \textit{floor}(x), denoted \( \left\lfloor x \right\rfloor \), is the greatest integer \( \leq x \)
- \textit{ceiling}(x), denoted \( \left\lceil x \right\rceil \), is the smallest integer \( \geq x \)
- Normally used to divide input into integral parts: \( \left\lfloor \frac{N}{2} \right\rfloor + \left\lceil \frac{N}{2} \right\rceil = N \)
Exponents

\[ X^A X^B = X^{A+B} \]

\[ \frac{X^A}{X^B} = X^{A-B} \]

\[ (X^A)^B = X^{AB} \]

\[ X^N + X^N = 2X^N \neq X^{2N} \]

\[ 2^N + 2^N = 2^{N+1} \]
Logarithms

\[ \log_X B = A \iff X^A = B \] "logarithm of B base X"

\[ \log_A B = \frac{\log_C B}{\log_C A} ; \quad A, B, C > 0, A \neq 1 \]

\[ \log AB = \log A + \log B ; \quad A, B > 0 \]

\[ \log \frac{A}{B} = \log A - \log B \]

\[ \log A^B = B \log A \]

\[ \log X < X \] for all \( X > 0 \)

\[ \lg A = \log_2 A \]

\[ \ln A = \log_e A ; \quad e = 2.7182... \] "natural logarithm"

Our convention for the course:

\[ \lg n = \log_2 n \]
\[ \log n = \log_{10} n \]
\[ \ln n = \log_e n \]

PS: In Weiss book, \( \log n \rightarrow \log_2 n \)
What is the meaning of the log function?

For example, lg 1024
Example

- How many times to halve an array of length n until its length is 1?

```python
def KeepHalving(n):
    i = 0
    while n != 1:
        i = i + 1
        n = floor(n/2)
    return i
```

What will be the value of i?
Factorials

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
(n-1)! & \text{if } n > 0 
\end{cases} \]

\[ n! < n^n \]

\[ n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right) \quad \text{Stirling's approximation} \]

n! == how many ways to order a set of n elements?
Modular Arithmetic

\[ A \mod N = A - N \cdot \lfloor A / N \rfloor \]

\[(A \mod N) = (B \mod N) \implies A \equiv B \pmod{N}\]

"A is congruent to B modulo N"

E.g., \(81 \equiv 61 \equiv 1 \pmod{10}\)

If \(A \equiv B \pmod{N}\)

Then \(A + C \equiv B + C \pmod{N}\)

and \(AD \equiv BD \pmod{N}\)

Basis of most encryption schemes:
(MESSAGE mod KEY)
Series

- General \[ \sum_{i=0}^{N} f(i) = f(0) + f(1) + \ldots + f(N) \]

- Linearity \[ \sum_{i=0}^{N} (cf(i) + g(i)) = c \sum_{i=0}^{N} f(i) + \sum_{i=0}^{N} g(i) \]

- Arithmetic series \[ \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \]
Series

- **Geometric series**

\[
\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}
\]

\[
\sum_{i=0}^{N} A^i \leq \sum_{i=0}^{\infty} A^i = \frac{1}{1 - A}; \text{ if } 0 < A < 1
\]

**Example**

How many nodes in a complete binary tree of depth \(D\)?

\[
A = 2, \ N = D = 2 \quad \Rightarrow \quad \frac{(2^{2+1} - 1)}{(2 - 1)} = 7
\]
Proofs

- What do we want to prove?
  - Properties of a data structure always hold for all operations
  - Algorithm’s running time / memory will never exceed some threshold
  - Algorithm will always be correct
  - Algorithm will always terminate

- Techniques
  - Proof by induction
  - Proof by counterexample
  - Proof by contradiction
Proof by Induction

- **Goal:** Prove some hypothesis is true
- **Three-step process**
  1. **Base case:** Show hypothesis is true for some initial conditions
  2. **Inductive hypothesis:** Assume hypothesis is true for all values ≤ k
  3. **Inductive step:** Show hypothesis is true for next larger value (typically k+1)

**Variation:**
Ind/hyp: All values < k,
Ind/step: show for value = k
Inductive Proof: Example

- Prove arithmetic series
  \[ \sum_{i=1}^{N} i = \frac{N(N + 1)}{2} \]

- Base case: Show true for \( N = 1 \)
  \[ \sum_{i=1}^{1} i = 1 = \frac{1(1 + 1)}{2} \quad \Rightarrow \text{Base case verified} \]
Example (cont.)

**Ind/Hyp:** Assume true for all \( N \leq k \)

**Ind/Step:** Now see if it is true for \( N = k+1 \)

\[
\sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^{k} i
\]

\[
= (k+1) + \frac{k(k+1)}{2}
\]

\[
= \frac{2(k+1) + k(k+1)}{2}
\]

\[
= \frac{(k+1)(k+2)}{2}
\]
More Examples for Induction Proofs

- Prove the geometric series

\[ \sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \]

- Prove that the number of nodes \( N \) in a complete binary tree of depth \( D \) is \( 2^{D+1} - 1 \)
Proof by Counterexample

Prove hypothesis is not true by giving an example that doesn’t work

- Example: $2^N > N^2$?
- Proof: $N=2$ (or 3 or 4)

- Proof by example?
- Proof by lots of examples?
- Proof by all possible examples?
  - Empirical proof
  - Hard when input size and contents can vary arbitrarily

Cpt S 223. School of EECS, WSU
Another Example for a proof by Counterexample

Given N cities and costs for traveling between each pair of cities, a "least-cost tour" is one which visits every city exactly once with the least cost.

**Hypothesis:** Any sub-path within any least-cost tour will also be a least-cost tour for those cities included in the sub-path.

Is this hypothesis true?
Proof by counterexample

- **Counterexample**
  - Cost \((A\rightarrow B\rightarrow C\rightarrow D)\) = 40 (optimal)
  - Cost \((A\rightarrow B\rightarrow C)\) = 30
  - Cost \((A\rightarrow C\rightarrow B)\) = 20

Conclusion: Least cost tours don’t necessarily contain smaller least cost tours
Proof by Contradiction

1. Start by assuming that the hypothesis is false

2. Show this assumption could lead to a contradiction (i.e., some known property is violated)

3. Therefore, hypothesis must be true
Example for proof by contradiction

Single pair shortest path problem

- Given N cities and costs for traveling between each pair of cities, find the least-cost path to go from city X to city Y

**Hypothesis:** A least-cost path from X to Y contains least-cost paths from X to every city on the path

- E.g., if X→C1→C2→C3→Y is a least-cost path from X to Y, then
  - X→C1→C2→C3 must be a least-cost path from X to C3
  - X→C1→C2 must be a least-cost path from X to C2
  - X→C1 must be a least-cost path from X to C1

**Conclusion:** Least cost paths should contain smaller least cost paths starting at the source
Proof by contradiction..

- Let P be a least-cost path from X to Y
- Now, assume that the hypothesis is false:
  - ==> there exists C along X->Y path, such that, there is a **better path** from X to C than the one in P
  - ==> So we could replace the subpath from X to C in P with this lesser-cost path, to create a new path P’ from X to Y
  - ==> Thus we now have a better path from X to Y
    - i.e., cost(P’) < cost(P)
  - ==> But this violates the fact that P is a least-cost path from X to Y
    - (hence a contradiction!)
- Therefore, the original hypothesis must be true
Mathematical Recurrence vs. Recursion

A recursive function or a recursive formula is defined in terms of itself

Example:

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
(n-1)! & \text{if } n > 0 
\end{cases} \]

Factorial (n)
if n = 0
then return 1
else return (n * Factorial (n-1))
Basic Rules of Recursion

- Base cases
  - Must always have some base cases, which can be solved without recursion

- Making progress
  - Recursive calls must always make progress toward a base case

- Design rule
  - Assume all recursive calls work

- Compound interest rule
  - Try not to duplicate work by solving the same instance of a problem in separate recursive calls
Example

- Fibonacci numbers
  - $F(0) = 1$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

Fibonacci (n)
if (n ≤ 1)
then return 1
else return (Fibonacci (n-1) + Fibonacci (n-2))

recursive code
So, is there a better way to write the Fibonacci code?

Example (cont.)

- *Computation tree* for: Fibonacci (5)

- Runtime for the recursive code (previous slide):
  - is proportional to the size of the tree
  - and that is a lot wasteful.
  - Why?
Running time for Fibonacci(n)?

- Show that the running time $T(n)$ of Fibonacci(n) is exponential in $n$
- Use mathematical induction
  - We can show that $T(n) < \left(\frac{5}{3}\right)^n$ for $n\geq1$
- Actually, this gives only an *upper bound* for $T(n)$
  - We also need to prove that $T(n)$ is at least exponential
Solving recurrences

- Example:
  ```
  Algo1(A,1,n)
  // A is an integer array of size n
  if(n<2) return;
  x = floor(n/2)
  Algo1(A,1,x)
  Algo1(A,x+1,n)
  ```

- How much time does Algo1 take?
  - Express time as a function of n (input size)
  - Let $T(n)$ be the time taken by Algo1 on an input size $n$
  - Then, $T(n) = 1 + T(n/2) + T(n/2)$
  - $= 2T(n/2) + 1$
Solving recurrences...

- **Recurrence:**
  \[
  T(n) = 2T(n/2) + 1
  \]
  \[
  T(1) = 1
  \] (base case)

- **Solution:**
  \[
  T(n) = 2T(n/2) + 1
  \]
  \[
  = 2[2T(n/2^2) + 1] + 1
  \]
  \[
  = 2^2T(n/2^2) + 2 + 1
  \]
  \[
  = 2^3T(n/2^3) + 2^2 + 2 + 1
  \]
  \[
  \vdots \quad (k \text{ steps})
  \]
  \[
  = 2^kT(n/2^k) + 2^{k-1} + \ldots + 2^2 + 2 + 1
  \]
  For termination, \( n/2^k = 1 \) \( \Rightarrow \) \( k = \log n \)
  \[
  T(n) = 2^\log nT(1) + n-1
  \]
  \[
  = 2n-1
  \]

This is the closed form for \( T(n) \)
Ponder this

1. Do constants matter for asymptotic analysis?

2. Recurrence vs. Recursion
   - A recurrence *need not* always be implemented using recursion
   - How?
Notion of a “recursion” as a function calling a function (same or not)

Recursive Function Calls

Code structure: (guess)?

M() {
    A()
    A()
    B()
    C()
}
A() {
    B()
}
B() {
    D()
    E()
}
C() {
    D()
    E()
}
D() {
    F()
}
E() {
    F()
}
F() {
}

Call tree:

Call sequence:

M
A
A
B
B
B
D
D
E
E

M
A
A
B
B
B
D
D
E
E

Cpt S 223. School of EECS, WSU
Why is iterative code more desirable than tail recursive code?

Refer to the note on tail recursion on the lecture notes web page.

Tail Recursion \( \Rightarrow \) Iteration

A(n) {
  ...
  A(n-1)
}

The result of A(n-1) is NOT needed/used in A(n)

A(n) {
  for(i=n;i>=0;i--){
    ...
  }
}

Tail recursive code

Last recursive call replaced with while() or for() loop

Iteration
Tower of Hanoi

**Goal:** Move all disks from peg A to peg B using peg C

**Rules:**
1. Move one disk at a time
2. Larger disks cannot be placed above smaller disks

Invented by a French Mathematician Edouard Lucas, 1883

**Question:** What is the minimum number of moves necessary to solve the problem?
Tower of Hanoi: Algorithm

- **A Recursive Algorithm:**
  1. First, move the top n-1 disks, “recursively”, from A to C (using B)
  2. Move n\textsuperscript{th} disk (i.e., largest & bottom-most in A) from A to B
  3. Then, move all the n-1 disks, “recursively”, from C to B (using A)
Recursive Algorithm for Tower of Hanoi (pseudocode)

- Move (n: disk, A, B, C)
- **PRE:** n disks on A; B and C unaffected
- **POST:** n disks on B; A and C unaffected
- BEGIN
  - IF n=0 THEN RETURN
  - Move (n-1, A,C,B)
  - Move $n^{th}$ disk from A to B directly
  - Move (n-1,C,B,A)
- END

Tail Recursion
Tower of Hanoi: Analysis

- Let $T(n)$ = minimum number of moves required to solve the problem

- **Analysis:**
  - $T(1)=1$ ➔ Base case
  - $T(n) = 2.T(n-1)+1$ ➔ recurrence
  - Solving this yields $T(n)=2^n-1$ (how?)
  - In the original Tower of Hanoi problem, $n=8$ & so $T(n)=255$ (which is fine!)

- For Tower of Brahma, $n=64$
  - $2^{64}-1$ moves made by a priest in a temple
  - Assuming each move takes 1 second, this would take $5,000,000,000$ centuries to complete
  - So lots of time before the world ends!
Summary

- Floors, ceilings, exponents, logarithms, series, and modular arithmetic
- Proofs by mathematical induction, counterexample and contradiction
- Recursion
- Solving recurrences
- Tools to help us analyze the performance of our data structures and algorithms
Try it out yourself