Acknowledgement: Parts of these course notes are based on notes from courses given by Jennifer Welch at Texas A&M University.
Mathematical Foundations

The initial part of this course focuses on giving us the mathematical foundations we need to use in the rest of the course. These are covered by Chapters 3-5 in the text:

- **Chap. 3, Growth of Functions** (Asymptotic Analysis)
  - we will review in class (Math 302 material)

- **Chap. 4, Recurrences**
  - we will review in class (Math 302 material)

- **Chap. 5, Probabilistic Analysis and Randomized Algorithms**
  - this should also be largely review, but we will cover parts of it as they are needed.
Asymptotic Analysis

Main Idea: We are interested in the work (running time) \textit{in the limit} as the input size grows to infinity

- focus on calculating running time in terms of its rate of growth with increasing problem size
  - disregard multiplicative constants
  - identify leading terms (of highest order)

Example: an algorithm with running time of order $n^2$ will “eventually” (i.e., for sufficiently large $n$) run slower than one with running time of order $n$, which in turn will eventually run slower than one with running time of order $\log n$.

- asymptotic analysis in terms of “Big Oh”, “Theta”, and “Omega” are the tools we will use to make these notions precise

Note: Our conclusions will only be valid “in the limit” or “asymptotically”. That is, they may not hold true for small values of $n$. (You will explore this issue in the programming assignments.)
“Big Oh” – Upper Bounding Running Time

**Definition:** \( g(n) \in O(f(n)) \) if \( \exists c > 0 \) and \( n_0 > 0 \) such that

\[
g(n) \leq cf(n)
\]

for all \( n \geq n_0 \) (often written as \( g(n) = O(f(n)) \)).

**Intuition:**

- \( g(n) \in O(f(n)) \) means \( g(n) \) is “less than or equal to” \( f(n) \) when we ignore small values of \( n \) and constant multiples.
- \( g(n) \) is *eventually* trapped below *some* constant multiple of \( f(n) \)
- *some* constant multiple of \( f(n) \) is an *upper bound* for \( g(n) \) (for large enough \( n \))

**Useful Way to Show “Big Oh” Relationships:**

\[
g(n) \in O(f(n)) \quad \text{iff} \quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = c
\]

for some constant \( c \geq 0 \).

... and L’Hopital’s Rule is useful for doing this...

If \( \lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty \), then (assuming \( f'(n) \) and \( g'(n) \) exist),

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{g'(n)}{f'(n)}
\]
Examples: “Big Oh”

The complexities for *insertion sort* are:

- **worst-case:** \( w(n) = \frac{1}{2}n^2 - \frac{1}{2}n \)
- **average-case:** \( a(n) = \frac{1}{4}n^2 + \frac{3}{4}n - 1 - \ln(n + 1) + \ln 2 \)
- **best-case:** \( b(n) = n - 1 \)

1. is \( b(n) = O(n) \)? (\( f(n) = n \), \( g(n) = b(n) \))

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n - 1}{n} = \lim_{n \to \infty} 1 - \frac{1}{n} = 1
\]

so the answer is yes!

2. is \( w(n) = O(n) \)?

3. is \( w(n) = O(n^2) \)?

4. is \( a(n) = O(n^2) \)?
“Omega” – Lower Bounding Running Time

**Definition:** $g(n) \in \Omega(f(n))$ if $\exists c > 0$ and $n_0 > 0$ such that

$$g(n) \geq cf(n)$$

for all $n \geq n_0$ (often written as $g(n) = \Omega(f(n))$).

**Intuition:**
- $g(n) \in \Omega(f(n))$ means $g(n)$ is “greater than or equal to” $f(n)$ when we ignore small values of $n$ and constant multiples.
- $g(n)$ is *eventually* trapped above *some* constant multiple of $f(n)$
- *Some* constant multiple of $f(n)$ is a *lower bound* for $g(n)$ (for large enough $n$)

**Useful Way to Show “Omega” Relationships:**

$$g(n) \in \Omega(f(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

for some constant $c \geq 0$.

... and again, L’Hopital’s Rule is useful for doing this.
Examples: “Omega”

The complexities for *insertion sort* are:

- **worst-case:** \( w(n) = \frac{1}{2}n^2 - \frac{1}{2}n \)
- **average-case:** \( a(n) = \frac{1}{4}n^2 + \frac{3}{4}n - 1 - \ln(n + 1) + \ln 2 \)
- **best-case:** \( b(n) = n - 1 \)

1. is \( b(n) = \Omega(n) \)? \((f(n) = n, g(n) = b(n))\)
   
   \[
   \lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n}{n - 1} = \lim_{n \to \infty} 1 + \frac{1}{n - 1} = 1
   \]

   so the answer is yes!

2. is \( w(n) = \Omega(n) \)?

3. is \( w(n) = \Omega(n^2) \)?

4. is \( a(n) = \Omega(n^2) \)?
“Theta” – Tightly Bounding Running Time

**Definition:** $g(n) \in \Theta(f(n))$ if $\exists c_1, c_2 > 0$ and $n_0 > 0$ such that

$$c_1 f(n) \leq g(n) \leq c_2 f(n)$$

for all $n \geq n_0$ (often written as $g(n) = \Theta(f(n))$).

**Intuition:**

- $g(n) \in \Theta(f(n))$ means $g(n)$ is “equal to” $f(n)$ when we ignore small values of $n$ and constant multiples.
- $g(n)$ is *eventually* trapped between *two* constant multiples of $f(n)$

**Useful Way to Show “Theta” Relationships:**

- The easiest way is to show both a “Big Oh” and an “Omega” relationship
- Can also use limits as before:

$$g(n) \in \Theta(f(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = c$$

for some constant $c > 0$ (note strictly greater than zero).
Examples: “Theta”

The complexities for *insertion sort* are:

- **worst-case:** \( w(n) = \frac{1}{2}n^2 - \frac{1}{2}n \)
- **average-case:** \( a(n) = \frac{1}{4}n^2 + \frac{3}{4}n - 1 - \ln(n + 1) + \ln 2 \)
- **best-case:** \( b(n) = n - 1 \)

1. is \( b(n) = \Theta(n) \)? (\( f(n) = n \), \( g(n) = b(n) \))
We have already seen that \( b(n) = O(n) \) and \( b(n) = \Omega(n) \), so the answer is yes!

We can also do it from scratch:

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n - 1}{n} = \lim_{n \to \infty} 1 - \frac{1}{n} = 1
\]
since \( 1 > 0 \), the answer is yes!

2. is \( w(n) = \Theta(n) \)?

3. is \( w(n) = \Theta(n^2) \)?

4. is \( a(n) = \Theta(n^2) \)?
Useful Properties for Asymptotic Analysis

We will use asymptotic analysis to make statements like:

- “An algorithm has worst-case running time $O(g(n))$” – which means there is a constant $c$ s.t. for every $n$ big enough, every execution on an input of size $n$ takes at most $cg(n)$ time.

- “An algorithm has worst-case running time $\Omega(g(n))$” – which means there is a constant $c$ s.t. for every $n$ big enough, at least one execution on an input of size $n$ takes at least $cg(n)$ time.

Some useful properties:

- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$ (transitive).
  
  **intuition:** if $f(n) \leq g(n)$ and $g(n) \leq h(n)$, then $f(n) \leq h(n)$

- also holds for $\Omega$ and $\Theta$

- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
  
  **intuition:** $f(n) \leq g(n)$ iff $g(n) \geq f(n)$,

- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
  
  **intuition:** $f(n) = g(n)$ iff $g(n) = f(n)$,

- $O(f(n) + g(n)) = O(\max(f(n), g(n)))$, e.g. $O(n^3 + n) = O(n^3)$
  
  $\Omega(f(n) + g(n)) = \Omega(\max(f(n), g(n)))$
  
  $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$
Little Oh and Little Omega

‘Little Oh’ and ‘Little Omega’ are used to denote strict upperbounds and lowerbounds, respectively (O and Ω bounds are not necessarily strict)

**Definition:** $g(n) \in o(f(n))$ if for every $c > 0$, there exists some $n_0 > 0$ such that for all $n \geq n_0 \quad g(n) < cf(n)$.

**Intuition:**
- $g(n) \in o(f(n))$ means $g(n)$ is “less than” any constant multiple of $f(n)$ when we ignore small values of $n$
- $g(n)$ is *eventually* trapped below *any* constant multiple of $f(n)$

**Definition:** $g(n) \in \omega(f(n))$ if for every $c > 0$, there exists some $n_0 > 0$ such that for all $n \geq n_0 \quad g(n) > cf(n)$.

**Intuition:**
- $g(n) \in \omega(f(n))$ means $g(n)$ is “greater than” any constant multiple of $f(n)$ when we ignore small values of $n$
- $g(n)$ is *eventually* trapped above *any* constant multiple of $f(n)$

**Showing “Little Oh and Little Omega” Relationships:**

$g(n) \in o(f(n)) \quad$ iff $\quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$

$g(n) \in \omega(f(n)) \quad$ iff $\quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$
Divide-and-Conquer Algorithms

The divide-and-conquer paradigm (Ch 2)

- **divide** the problem into a number of subproblems
- **conquer** the subproblems (solve them)
- **combine** the subproblem solutions to get the solution to the original problem

**Note:** often the “conquer” step is done **recursively**

**Recursive algorithm:** to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.
- usually the subproblems are smaller in size than the ‘parent’ problem
- divide-and-conquer algorithms are often recursive

**Example: Merge Sort**

- **divide** the $n$-element sequence to be sorted into two $\frac{n}{2}$-element sequences
- **conquer:** sort the subproblems, recursively using merge sort
- **combine:** merge the resulting two sorted $\frac{n}{2}$-element sequences
Analyzing Divide-and-Conquer Algorithms

When an algorithm contains a recursive call to itself, its running time can often be described by a **recurrence equation** which describes the overall running time on a problem of size \( n \) in terms of the running time on smaller inputs.

For divide-and-conquer algorithms, we get recurrences that look like:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c \\
aT(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases}
\]

where

- \( a \) = the number of subproblems we break the problem into
- \( n/b \) = the size of the subproblems (in terms of \( n \))
- \( D(n) \) is the time to divide the problem of size \( n \) into the subproblems
- \( C(n) \) is the time to combine the subproblem solutions to get the answer for the problem of size \( n \)

**Example: Merge Sort**

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq c \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

- \( a = 2 \) (two subproblems)
- \( n/b = n/2 \) (each subproblem has size approx \( n/2 \))
- \( D(n) = \Theta(1) \) (just compute midpoint of array)
- \( C(n) = \Theta(n) \) (merging can be done by scanning sorted subarrays)
Solving Recurrences

There are 3 general methods for solving recurrences (Ch. 4)

1. **Iteration: Convert to Summation:** convert the recurrence into a summation (by expanding some terms) and then bound the summation

2. **Substitution: Guess & Verify:** guess a solution and verify it is correct with an inductive proof

3. **Apply “Master Theorem”:** if the recurrence has the form

   \[ T(n) = aT(n/b) + f(n) \]

   then there is a formula that can (often) be applied.

**Simplifications:** there are two simplifications we apply that won’t affect asymptotic analysis

- ignore floors and ceilings (justification in text)
- assume base cases are constant, i.e., \( T(n) = \Theta(1) \) for \( n \) small enough
Solving Recurrences: Iteration (convert to summation)

Example: \( T(n) = 4T\left(\frac{n}{2}\right) + n \)

\[
T(n) = 4T\left(\frac{n}{2}\right) + n \\
= 4\left(\frac{n}{2} + 4T\left(\frac{n}{4}\right)\right) + n \\
= 16T\left(\frac{n}{4}\right) + 2n + n \\
= 16\left(\frac{n}{4} + 4T\left(\frac{n}{8}\right)\right) + 2n + n \\
= 64T\left(\frac{n}{8}\right) + 4n + 2n + n \\
= 4^\log n T(1) + \ldots + 4n + 2n + n \\
= c4^\log n + n \sum_{k=0}^{\log n-1} 2^k \\
= cn^\log 4 + n \left(\frac{2^{\log n - 1}}{2 - 1}\right) \\
= cn^2 + n(n^{\log 2} - 1) \\
= cn^2 + n(n - 1) \\
= cn^2 + n^2 - n \\
= \Theta(n^2)
\]

Intuitive Help: Can represent this as a recursion tree and identify computation with each node/level in the tree.

- root represents computation \((D(n) + C(n))\) at top level of recursion
- node at level \(i\) represents subproblem at level \(i\) in the recursion
- height of tree is number of levels in the recursion
- \( T(n) = \text{sum of all nodes in the tree} \)
Solving Recurrences: Substitution (guess and verify)

This method involves

- guessing form of solution
- use mathematical induction to find the constants and verify solution
- use to find an upper or a lower bound (do both to obtain a tight bound)

**Example:** $T(n) = 4T(n/2) + n$ (upper bound)

*guess* $T(n) = O(n^3)$ and try to show $T(n) \leq cn^3$ for some $c > 0$ (we’ll have to find $c$)

*basis*?

*assume* $T(k) \leq ck^3$ for $k < n$, and prove $T(n) \leq cn^3$

$$
T(n) = 4T(n/2) + n \\
\leq 4(c(n/2)^3) + n \quad /\text{by inductive hypothesis}/ \\
= \frac{c}{2}n^3 + n \\
= cn^3 - (\frac{c}{2}n^3 - n) \\
\leq cn^3
$$

where the last step holds if $c \geq 2$ and $n \geq 1$.

We find values of $c$ and $n_0$ by determining when $\frac{c}{2}n^3 - n \geq 0$

**Useful Tricks:** are in text (e.g., subtract lower order term, change of variables)
Practice: Substitution (guess and verify)

**Problem 1:** Give an upper bound for \( T(n) = 2T(n/2) + n \)

*guess* \( T(n) = O(n) \) and try to show \( T(n) \leq cn \) for some \( c > 0 \) (you have to find \( c \))

*basis*?

*assume* \( T(k) \leq ck \) for \( k < n \), and prove \( T(n) \leq cn \)

\[
T(n) = 2T(n/2) + n \\
\leq 2(cn^2) + n \quad \text{/**by inductive hypothesis**/} \\
= cn + n \\
= O(n) \quad \text{/**WRONG!**/}
\]

**Question:** What is wrong with the above proof?

**Problem 2:** Show \( T(n) = 2T(n/2) + n \) is \( \Omega(n \log n) \) using the substitution method.
Solving Recurrences: The Master Method

The master method provides a ‘cookbook’ method for solving recurrences of a certain form.

**Master Theorem:** Let \( a \geq 1 \) and \( b > 1 \) be constants, let \( f(n) \) be a function, and let \( T(n) \) be defined on nonnegative integers as:

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n),
\]

Then, \( T(n) \) can be bounded asymptotically as follows:

1. \( T(n) = \Theta(n^{\log_b a}) \) \quad if \( f(n) = O(n^{\log_b a-\epsilon}) \) for some constant \( \epsilon > 0 \)
2. \( T(n) = \Theta(n^{\log_b a} \log n) \) \quad if \( f(n) = \Theta(n^{\log_b a}) \)
3. \( T(n) = \Theta(f(n)) \) \quad if \( f(n) = \Omega(n^{\log_b a+\epsilon}) \) for some constant \( \epsilon > 0 \)
   and if \( af\left(\frac{n}{b}\right) \leq cf(n) \) for some constant \( c < 1 \)
   and all sufficiently large \( n \).

**Intuition:** compare \( f(n) \) with \( \Theta(n^{\log_b a}) \)

- case 1: \( f(n) \) is **polynomially** smaller than \( \Theta(n^{\log_b a}) \)
- case 2: \( f(n) \) is **asymptotically** equal to \( \Theta(n^{\log_b a}) \)
- case 3: \( f(n) \) is **polynomially** larger than \( \Theta(n^{\log_b a}) \)

What is \( \log_b a \)? The number of times we divide \( a \) by \( b \) to reach \( O(1) \).
Solving Recurrences: Master Method

**Example:** $T(n) = 9T(n^\frac{1}{3}) + n$
- $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_b a} = n^{\log_3 9} = n^2$
- compare $f(n) = n$ with $n^{\log_b a} = n^2$
  - $n = O(n^{2-\varepsilon})$ (f(n) is polynomially smaller than $n^{\log_b a}$)
- case 1 applies: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

**Example:** $T(n) = T(\frac{2}{3}n) + 1$
- $a = 1$, $b = \frac{3}{2}$, $f(n) = 1$, $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- compare $f(n) = 1$ with $n^{\log_b a} = 1$
  - $1 = \Theta(1)$ (f(n) is asymptotically equal to $n^{\log_b a}$)
- case 2 applies: $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$

**Example:** $T(n) = 3T(n^{\frac{1}{4}}) + n \log n$
- $a = 3$, $b = 4$, $f(n) = n \log n$, $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$
- compare $f(n) = n \log n$ with $n^{\log_b a} = n^{0.793}$
  - $n \log n = \Omega(n^{0.793+\varepsilon})$ (f(n) is polynomially larger than $n^{\log_b a}$)
- case 3 **might** apply: need to check ‘regularity’ of f(n)
  - find $c < 1$ s.t. $a f(n^{\frac{1}{4}}) \leq c f(n)$ for large enough $n$
  - i.e., $3n^{\frac{3}{4}} \log \frac{n}{4} \leq cn \log n$ which is true for $c = \frac{3}{4}$
- case 3 applies: $T(n) = \Theta(f(n)) = \Theta(n \log n)$

**Problem 1.** $T(n) = 4T(n^{\frac{1}{2}}) + n^2$

**Problem 2.** $T(n) = 4T(n^{\frac{1}{2}}) + \frac{n^2}{\log n}$