Marking

- Four assignments: 15% each
- Midterm: 10%
- Final: 30%

Method of exam delivery has not been decided on.
My office hours

- Monday, Wednesday, and Friday from 2:30 to 3:20+
- On Zoom. Details to be worked out and placed on the course website.

TAs office hours will also be on Zoom. They start next week. Hours TBA.
Analysis of Algorithms
Running Time

- Most algorithms transform input objects into output objects.

- The running time of an algorithm typically grows with the input size.

- Average case time is often difficult to determine.

- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `clock()` to get an accurate measure of the actual running time
- Plot the results
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used, and the same amount of care in implementation.
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, $n$.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm `arrayMax(A, n)`

**Input** array `A` of `n` integers

**Output** maximum element of `A`

1. `currentMax ← A[0]`
2. for `i ← 1 to n − 1` do
   1. if `A[i] > currentMax` then
      1. `currentMax ← A[i]`
3. return `currentMax`
Pseudocode Details

- **Control flow**
  - `if ... then ... [else ...]`
  - `while ... do ...`
  - `repeat ... until ...`
  - `for ... do ...`
  - Indentation replaces braces

- **Method declaration**
  
  Algorithm `method (arg [, arg...])`
  
  Input ...
  
  Output ...

- **Method call**
  
  `var.method (arg [, arg...])`

- **Return value**
  
  `return expression`

- **Expressions**
  
  ← Assignment
  (like `=` in C++)
  
  `=` Equality testing
  (like `==` in C++)
  
  \( n^2 \) Superscripts and other mathematical formatting allowed
The Random Access Machine (RAM) Model

- A CPU
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character

Memory cells are numbered and accessing any cell in memory takes unit time.
Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant \(\approx 1\)
  - Logarithmic \(\approx \log n\)
  - Linear \(\approx n\)
  - N-Log-N \(\approx n \log n\)
  - Quadratic \(\approx n^2\)
  - Cubic \(\approx n^3\)
  - Exponential \(\approx 2^n\)

- In a log-log chart, the slope of the line corresponds to the growth rate.
Functions Graphed Using “Normal” Scale

- $g(n) = 1$
- $g(n) = n \lg n$
- $g(n) = 2^n$
- $g(n) = \lg n$
- $g(n) = n^2$
- $g(n) = n$
- $g(n) = n^3$
Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Examples:
- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

**Algorithm arrayMax(A, n)**

```plaintext
currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
    { increment counter i }
return currentMax
```

<table>
<thead>
<tr>
<th># operations</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2n</td>
<td></td>
</tr>
<tr>
<td>2(n − 1)</td>
<td></td>
</tr>
<tr>
<td>2(n − 1)</td>
<td></td>
</tr>
<tr>
<td>2(n − 1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Total: \(8n − 3\)
Estimating Running Time

- Algorithm *arrayMax* executes $8n - 3$ primitive operations in the worst case. Define:
  - $a =$ Time taken by the fastest primitive operation
  - $b =$ Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of *arrayMax*. Then
  $$a (8n - 3) \leq T(n) \leq b(8n - 3)$$

- Hence, the running time $T(n)$ is bounded by two linear functions
Growth Rate of Running Time

- Changing the hardware/software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax

© 2010 Goodrich, Tamassia
Analysis of Algorithms
## Why Growth Rate Matters

<table>
<thead>
<tr>
<th>if runtime is...</th>
<th>time for (n + 1)</th>
<th>time for (2n)</th>
<th>time for (4n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c \lg n)</td>
<td>(c \lg (n + 1))</td>
<td>(c (\lg n + 1))</td>
<td>(c(\lg n + 2))</td>
</tr>
<tr>
<td>(c n)</td>
<td>(c (n + 1))</td>
<td>(2c n)</td>
<td>(4c n)</td>
</tr>
<tr>
<td>(c n \lg n)</td>
<td>(\sim c n \lg n + c n)</td>
<td>(2c n \lg n + 2cn)</td>
<td>(4c n \lg n + 4cn)</td>
</tr>
<tr>
<td>(c n^2)</td>
<td>(\sim c n^2 + 2c n)</td>
<td><strong>4c n^2</strong></td>
<td>16c n^2</td>
</tr>
<tr>
<td>(c n^3)</td>
<td>(\sim c n^3 + 3c n^2)</td>
<td>8c n^3</td>
<td>64c n^3</td>
</tr>
<tr>
<td>(c 2^n)</td>
<td>(c 2^{n+1})</td>
<td>(c 2^{2n})</td>
<td>(c 2^{4n})</td>
</tr>
</tbody>
</table>

### Observations:
- The runtime quadruples when the problem size doubles.
Comparison of Two Algorithms

- Insertion sort is $n^2 / 4$
- Merge sort is $2n \log n$

Sort a million items?
- Insertion sort takes roughly 70 hours
- Merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it’s 40 minutes versus less than 0.5 seconds
Constant Factors

- The growth rate is not affected by constant factors or lower-order terms.
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function
Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2) n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
Big-Oh Example

- Example: the function $n^2$ is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since $c$ must be a constant
More Big-Oh Examples

- **7n-2**
  7n-2 is $O(n)$
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- **$3n^3 + 20n^2 + 5$**
  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$

- **$3 \log n + 5$**
  $3 \log n + 5$ is $O(\log n)$
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$
Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.

- The statement \( f(n) \text{ is } O(g(n)) \) means that the growth rate of \( f(n) \) is no more than the growth rate of \( g(n) \).

- We can use the big-Oh notation to rank functions according to their growth rate.

<table>
<thead>
<tr>
<th></th>
<th>( f(n) \text{ is } O(g(n)) )</th>
<th>( g(n) \text{ is } O(f(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g(n) \text{ grows more} )</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( f(n) \text{ grows more} )</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\( \text{© 2010 Goodrich, Tamassia} \)
Big-Oh Rules

- If is $f(n)$ a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say “$2n$ is $O(n)$” instead of “$2n$ is $O(n^2)$”
- Use the simplest expression of the class
  - Say “$3n + 5$ is $O(n)$” instead of “$3n + 5$ is $O(3n)$”
Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation.

- To perform the asymptotic analysis:
  - We find the worst-case number of primitive operations executed as a function of the input size.
  - We express this function with big-Oh notation.

- Example:
  - We determine that algorithm \textit{arrayMax} executes at most $8n - 2$ primitive operations.
  - We say that algorithm \textit{arrayMax} “runs in $O(n)$ time”.

- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The $i$-th prefix average of an array $X$ is average of the first $(i + 1)$ elements of $X$:
  \[ A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1} \]
- Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.
Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition.

Algorithm `prefixAverages1(X, n)`

**Input** array `X` of `n` integers

**Output** array `A` of prefix averages of `X`

1. `A ← new array of `n` integers`
2. `for i ← 0 to n − 1 do`
   1. `s ← X[0]`
   2. `for j ← 1 to i do`
      1. `s ← s + X[j]`
      2. `A[i] ← s / (i + 1)`
3. `return A`

#operations:
- `n`
- `n`
- `n`
- `1 + 2 + ... + (n − 1)`
- `1 + 2 + ... + (n − 1)`
- `n`
- `1`
The running time of \texttt{prefixAverages1} is \(O(1 + 2 + \ldots + n)\).

The sum of the first \(n\) integers is \(n(n + 1)/2\).
- There is a simple visual proof of this fact.

Thus, algorithm \texttt{prefixAverages1} runs in \(O(n^2)\) time.
Prefix Averages (Linear)

The following algorithm computes prefix averages in linear time by keeping a running sum.

**Algorithm** \(\text{prefixAverages2}(X, n)\)

**Input** array \(X\) of \(n\) integers

**Output** array \(A\) of prefix averages of \(X\)

\[
\begin{align*}
A & \leftarrow \text{new array of } n \text{ integers} \\
s & \leftarrow 0 \\
\text{for } i & \leftarrow 0 \text{ to } n - 1 \text{ do} \\
& \quad s \leftarrow s + X[i] \\
& \quad A[i] \leftarrow s / (i + 1) \\
\text{return } A
\end{align*}
\]

Algorithm \(\text{prefixAverages2}\) runs in \(O(n)\) time
Math you need to Review

- Summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

- properties of logarithms:
  \[ \log_b(xy) = \log_b x + \log_b y \]
  \[ \log_b (x/y) = \log_b x - \log_b y \]
  \[ \log_b xa = a \log_b x \]
  \[ \log_b a = \log_x a / \log_x b \]

- properties of exponentials:
  \[ a^{(b+c)} = a^b a^c \]
  \[ a^{bc} = (a^b)^c \]
  \[ a^b / a^c = a^{(b-c)} \]
  \[ b = a^{\log_a b} \]
  \[ b^c = a^{c \log_a b} \]
Relatives of Big-Oh

big-Omega

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

big-Theta

- $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
Intuition for Asymptotic Notation

**Big-Oh**
- \( f(n) \) is \( O(g(n)) \) if \( f(n) \) is asymptotically \textit{less than or equal} to \( g(n) \)

**big-Omega**
- \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically \textit{greater than or equal} to \( g(n) \)

**big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically \textit{equal} to \( g(n) \)
Example Uses of the Relatives of Big-Oh

- *5n^2 is \Omega(n^2)*

  \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \).

  Let \( c = 5 \) and \( n_0 = 1 \).

- *5n^2 is \Omega(n)*

  \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \).

  Let \( c = 1 \) and \( n_0 = 1 \).

- *5n^2 is \Theta(n^2)*

  \( f(n) \) is \( \Theta(g(n)) \) if it is \( \Omega(n^2) \) and \( O(n^2) \). We have already seen the former, for the latter recall that \( f(n) \) is \( O(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \).

  Let \( c = 5 \) and \( n_0 = 1 \).
Algorithm Analysis –
Tom’s Rules of Thumb

- Start by defining a function that represents the time for the thing you are trying to analyze.
  - Often $T(n)$
  - Be sure to state what $n$ is.
  - Time is worst-case if not specified.

```c
quicksort(A, i, j) {
    ...
}
```

Let $T(n)$ be the time to complete quicksort on $n$ array elements, where $n = j-i+1$.

```c
bubblesort(A) {
    ...
}
```

Let $S(n)$ be the time to complete bubblesort on an array of $n$ elements.
Algorithm Analysis –
Tom’s Rules of Thumb

- Work from inner blocks of (pseudo-)code to outer blocks.

quicksort(A, i, j) {
    pivot = A[i];
    for(k = i+1 to j) {
        if(pivot > A[k]) {
            ...
        } else {
            ...
        }
    }
    ...
}

Start with these

Then do this

Then this ...
Assignments and Function Calls

- An assignment with no function calls is $O(1)$.
- A function call to a known algorithm (not the one you are analyzing) takes the known time for that algorithm.

```plaintext
def foo(A, n):
    ...
    k = (j+91)/3;  \quad \text{O(1)}
    m = \max(A, n);  \quad \text{O(n)}  \quad \text{(finding maximum of an array takes linear time)}
    p = (j - 17) + \max(B, n)  \quad \text{O(1)} + \text{O(n)} = \text{O(n)}
    ...
```

© 2019 Shermer  Analysis of Algorithms  36
Recursive Function Calls

- A function call to the algorithm you are analyzing takes (the function you defined) (some function of n) time.
- For example, $T(n/2)$

```plaintext
foo(A, n) {
  ...
  foo(A, n-2);
}
```

Define $S(n)$ to be time taken by foo with second parameter $n$

```plaintext
bar(A, i, j) {
  m = (i + j) / 2;
  bar(A, i, m);
  ...
}
```

Define $T(n)$ to be time taken by bar with $n = j-i+1$
Conditionals

- Add up the time in each branch of the conditional.
- The conditional takes the time taken by the condition, plus the maximum of the branch times.

```plaintext
foo(A, n) {
...
if( p < A[i] ) {
    ...
    O(1)
    O(n)

    O(n^2)
}
else {
    ...
}
}
```

The whole if statement takes time

\[ O(1) + \max(O(n), O(n^2)) = O(n^2) \]
Loops

- Add up the time in the body of the loop.
- Determine how many times \( t \) the loop will be executed, as a function of your \( n \). Use worst-case estimate.
- The time for the loop is \( t \times (\text{time for body}) \)

```java
for(i = 1 to n ) {
    n iterations
    ...
    O(n)
}
```

The whole for loop takes time
\( n \times O(n) = O(n^2) \)

```java
i = 0;
while(p < A[i]) {
    n iterations
    ...
    i++;
}
```
Triangular Loops

- Triangular loops have an inner index that counts up to the outer index.
- Assume the inner loops have the same number of iterations as the outer loop.

```plaintext
for(i = 1 to n) {
  n iterations
  ...
  for(j = 1 to i) {
    n iterations
      ...
  }
}
```

- This also works for more than two nested loops
At the end

- Set (the function of \( n \) you defined) = the summed-up cost of the entire algorithm.
- Reminder: Along the way, don’t absorb \( T(...) \) factors into the big-Oh notation.
- If you end up with a recurrence, solve it.
At the end

```c
int find(A, i, j, target) {
    if( i == j) {
        if(A[i] == target)
            return i
        else
            return -1
    } else
        m = (i + j) / 2
    f = find(A, i, m, target)
    if(f > 0) {
        return f
    } else
        return find(A, m+1, j, target)
}
```

Let $T(n)$ be the time for find where $n = j-i+1$.

$T(n) = O(1) + 2T(n/2)$

$T(n)$ is $O(n)$