Recap

- What should you have learned?
  - Basic java programming skills
    - Object-oriented programming
    - Classes and objects
    - Inheritance, exception handling, generics
    - Java class library
  - Basic data structures and their applications
    - Linear data structure: linked list, array, stack, queue
    - Hierarchical data structures: tree and heap
Wrap up

- What are you going to learn in the rest of this semester?
  - Algorithms
    - Analysis of algorithms
    - Brute force, divide and conquer, dynamic programming
    - Sorting
  - Advanced data structures
    - Hash table
    - Map and dictionary
    - Graph
Analysis of Algorithms

How good is your program?
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 `System.currentTimeMillis()` 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.
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
Pseudo code

- 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 the max element of an array

Algorithm $arrayMax(A, n)$

Input array $A$ of $n$ integers

Output the maximum element of $A$

$\text{currentMax} \leftarrow A[0]$

for $i \leftarrow 1$ to $n - 1$ do

if $A[i] > \text{currentMax}$ then

$\text{currentMax} \leftarrow A[i]$

return $\text{currentMax}$
Pseudo code

Example: find the max element of an array

Algorithm \textit{arrayMax}(A, n)
\begin{itemize}
  \item \textbf{Input} array $A$ of $n$ integers
  \item \textbf{Output} the maximum element of $A$
\end{itemize}
\begin{itemize}
  \item currentMax $\leftarrow A[0]$
  \item for $i \leftarrow 1$ to $n - 1$ do
    \begin{itemize}
    \item if $A[i] >$ currentMax then
      \begin{itemize}
        \item currentMax $\leftarrow A[i]$
      \end{itemize}
    \end{itemize}
  \item return currentMax
\end{itemize}

Find the min element of an array

Algorithm \textit{arrayMin}(A, n)
\begin{itemize}
  \item \textbf{Input} array $A$ of $n$ integers
  \item \textbf{Output} the minimum element of $A$
\end{itemize}
\begin{itemize}
  \item currentMin $\leftarrow A[0]$
  \item for $i \leftarrow 1$ to $n - 1$ do
    \begin{itemize}
    \item if $A[i] <$ currentMin then
      \begin{itemize}
        \item currentMin $\leftarrow A[i]$
      \end{itemize}
    \end{itemize}
  \item return currentMin
\end{itemize}
Pseudo code

Find the min element of an array

Algorithm \textit{arrayMin}(A, n)
\begin{itemize}
  \item \textbf{Input} array \textit{A} of \textit{n} integers
  \item \textbf{Output} the minimum element of \textit{A}
\end{itemize}

\texttt{currentMin} \leftarrow A[0]
\textbf{for} \textit{i} \leftarrow 1 \textbf{to} \textit{n} - 1 \textbf{do}
  \textbf{if} \textit{A}[\textit{i}] < \texttt{currentMin} \textbf{then}
    \texttt{currentMin} \leftarrow \textit{A}[	extit{i}]
\textbf{return} \texttt{currentMin}

Sum all the elements of an array

Algorithm \textit{arraySum}(A, n)
\begin{itemize}
  \item \textbf{Input} array \textit{A} of \textit{n} integers
  \item \textbf{Output} sum of all the elements of \textit{A}
\end{itemize}

\texttt{currentSum} \leftarrow 0
\textbf{for} \textit{i} \leftarrow 0 \textbf{to} \textit{n} - 1 \textbf{do}
  \texttt{currentSum} \leftarrow \texttt{currentSum} + \textit{A}[	extit{i}]
\textbf{return} \texttt{currentSum}
**Pseudo code**

Sum all the elements of an array

Multiply all the elements of an array

---

**Algorithm arraySum**\((A, n)\)

Input array \(A\) of \(n\) integers

Output sum of all the elements of \(A\)

\[
\text{currentSum} \leftarrow 0 \\
\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
\quad \text{currentSum} \leftarrow \text{currentSum} + A[i] \\
\text{return currentSum}
\]

---

**Algorithm arrayMultiply**\((A, n)\)

Input array \(A\) of \(n\) integers

Output Multiply all the elements of \(A\)

\[
\text{current} \leftarrow 1 \\
\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \\
\quad \text{current} \leftarrow \text{current} \times A[i] \\
\text{return current}
\]
Pseudo code Details

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

- Method declaration
  
  Algorithm method (arg [, arg...])
  
  Input ...
  
  Output ...
Pseudo code Details

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

- Return value
  
  ```
  return expression
  ```

- Expressions
  
  $\leftarrow$ Assignment
    
    (like $=$ in Java)
  
  $=$ Equality testing
    
    (like $==$ in Java)

  $n^2$ 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 $\approx 2^n$
Functions Graphed Using “Normal” Scale

- $g(n) = 1$
- $g(n) = \lg n$
- $g(n) = n$
- $g(n) = n^2$
- $g(n) = n^3$
- $g(n) = n \lg n$
- $g(n) = 2^n$
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 \texttt{arrayMax}(A, n)

\texttt{currentMax} \leftarrow A[0]

\texttt{for } i \leftarrow 1 \texttt{ to } n - 1 \texttt{ do}

\hspace{1em} \text{if } A[i] > \texttt{currentMax} \texttt{ then}

\hspace{2em} \texttt{currentMax} \leftarrow A[i]

\hspace{1em} \{ \text{ increment counter } i \} \\

\texttt{return } \texttt{currentMax}

\begin{tabular}{l|c}

\hline

# operations & \\

\hline

2 & 2 \\

2n & 2n \\

2(n - 1) & 2(n - 1) \\

2(n - 1) & 2(n - 1) \\

1 & 1 \\

\hline

Total & 8n - 2 \\

\hline

\end{tabular}
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

<table>
<thead>
<tr>
<th>Algorithm arrayMultiply(A, n)</th>
<th>#operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>current ← 1</td>
<td></td>
</tr>
<tr>
<td>for i ← 0 to n - 1 do</td>
<td></td>
</tr>
<tr>
<td>current ← current*A[i]</td>
<td>2(n+1)</td>
</tr>
<tr>
<td>{ increment counter i }</td>
<td>3n</td>
</tr>
<tr>
<td>return current</td>
<td>2n</td>
</tr>
</tbody>
</table>

Total 7n+4 => O(n)
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>ArrayAverage(A, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>current ← 0</td>
<td></td>
</tr>
<tr>
<td>for i ← 0 to n - 1 do</td>
<td></td>
</tr>
<tr>
<td>current ← current + A[i]</td>
<td></td>
</tr>
<tr>
<td>{ increment counter i }</td>
<td></td>
</tr>
<tr>
<td>return current/n</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2(n+1)</td>
</tr>
<tr>
<td>3n</td>
</tr>
<tr>
<td>2n</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Total: \(7n+5\) \(\Rightarrow O(n)\)
Estimating Running Time

- Algorithm \textit{arrayMax} executes $8n - 2$ primitive operations in the worst case. Define:
  \[ a = \text{Time taken by the fastest primitive operation} \]
  \[ b = \text{Time taken by the slowest primitive operation} \]

- Let $T(n)$ be worst-case time of \textit{arrayMax}. Then
  \[ a \ (8n - 2) \leq T(n) \leq b(8n - 2) \]

- 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`
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>( cn )</td>
<td>( c (n + 1) )</td>
<td>( 2cn )</td>
<td>( 4cn )</td>
</tr>
<tr>
<td>( cn \lg n )</td>
<td>(~ c n \lg n + cn)</td>
<td>( 2cn \lg n + 2cn)</td>
<td>( 4cn \lg n + 4cn)</td>
</tr>
<tr>
<td>( cn^2 )</td>
<td>(~ c n^2 + 2cn)</td>
<td>( 4cn^2 )</td>
<td>( 16cn^2 )</td>
</tr>
<tr>
<td>( cn^3 )</td>
<td>(~ c n^3 + 3cn^2)</td>
<td>( 8cn^3 )</td>
<td>( 64cn^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>

runtime quadruples when problem size doubles
Comparison of Two Algorithms

insertion sort is \( \frac{n^2}{4} \)

merge sort is \( 2n \lg n \)

sort a million items?

insertion sort takes roughly 70 hours
while
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

\[ f(n) \leq cg(n) \text{ for } n \geq n_0 \]

- 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 Example

- 7n-2 is O(n)
  - need c > 0 and n₀ ≥ 1 such that 7n-2 ≤ c•n for n ≥ n₀
  - this is true for c = 7 and n₀ = 1
- 3n³ + 20n² + 5 is O(n³)
  - need c > 0 and n₀ ≥ 1 such that 3n³ + 20n² + 5 ≤ c•n³ for n ≥ n₀
  - this is true for c = 4 and n₀ = 21
- 3 log n + 5 is O(log n)
  - need c > 0 and n₀ ≥ 1 such that 3 log n + 5 ≤ c•log n for n ≥ n₀
  - this is true for c = 8 and n₀ = 2
Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement “$f(n)$ 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>$f(n)$ is $O(g(n))$</th>
<th>$g(n)$ is $O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n)$ grows more</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$f(n)$ grows more</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Big-Oh Rules

- If $f(n)$ is 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 $O(3n)$”
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.
Asymptotic Algorithm Analysis

- Example:
  - We determine that algorithm *arrayMax* executes at most $8n - 2$ primitive operations
  - We say that algorithm *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 the average of the first $(i + 1)$ elements of $X$:

  $$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.
Exercise

- Implement prefixAverage

- **Input:**
  - Get n integers from a txt file
  - The first integer indicates the number of integers (the size of X)

- **Output:**
  - Print out a sequence of integers
  - The ith integer indicates the average of the first i+1 input numbers (starting from the second input)

Input: 4 1 2 3 5
Output: 1 2 2
Prefix Average (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

A ← new array of n integers

for i ← 0 to n - 1 do

s ← X[0]

for j ← 1 to i do

s ← s + X[j]

A[i] ← s / (i + 1)

return A

#operations

n

n

n

1 + 2 + ... + (n - 1)

1 + 2 + ... + (n - 1)

n

1
Arithmetic Progression

- The running time of `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 `prefixAverages1` runs in $O(n^2)$ time
The following algorithm computes prefix averages in linear time by keeping a running sum.

Algorithm `prefixAverages2` runs in $O(n)$ time.

**Algorithm prefixAverages2**($X$, $n$)

- **Input** array $X$ of $n$ integers
- **Output** array $A$ of prefix averages of $X$
- $A \leftarrow$ new array of $n$ integers
- $s \leftarrow 0$
- for $i \leftarrow 0$ to $n - 1$ do
  - $s \leftarrow s + X[i]$
  - $A[i] \leftarrow s / (i + 1)$
- return $A$
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 \textbf{less than or equal} to $g(n)$

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

- **big-Theta**
  - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically \textbf{equal} to $g(n)$
Examples of Using Relatives of Big-Oh

- **5n^2 is Ω(n^2)**
  - \( f(n) \) is \( Ω(g(n)) \) 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 Ω(n)**
  - \( f(n) \) is \( Ω(g(n)) \) 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 Θ(n^2)**
  - \( f(n) \) is \( Θ(g(n)) \) if it is \( Ω(n^2) \) 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 \( n_0 = 1 \)
Coming Up…

- Big O: Read Text Book 4
- Divide and Conquer/Sorting: Read Text Book 11