Acknowledgement:
These slides are adapted from slides provided with *Data Structures and Algorithms in C++*
Goodrich, Tamassia and Mount (Wiley, 2004)

**Analysis of Algorithms**

Data Structures and Algorithms

**Motivation**

- **What to do with algorithms?**
  - Programmer needs to develop a working solution
  - Client wants problem solved efficiently
  - Theoretician wants to understand

- **Why analyze algorithms?**
  - To compare different algorithms for the same task
  - To predict performance in a new environment
  - To set values of algorithm parameters

**Outline and Reading**

- Running time (§4.2)
- Pseudo-code
- Counting primitive operations (§4.2.2)
- Asymptotic notation (§4.2.3)
- Asymptotic analysis (§4.2.4)
- Case study (§4.2.5)

**Running Time**

- We are interested in the design of "good" data structures and algorithms.
- Measure of "goodness":
  - Running time (most important)
  - Space usage
- The running time of an algorithm typically grows with the input size, and is affected by other factors:
  - Hardware environments: processor, memory, disk.
  - Software environments: OS, compiler.
- **Focus:** input size vs. running time.
Experimental Studies

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

Measure Actual Running Time

```java
//generate input data

//begin timing
clock_t k=clock();
clock_t start;
do {
    start = clock();
    while (start == k);
}while(true); //begin at new tick

//Run the test _num_itr times
for(int i=0; i<_num_itr; ++i) {
    //run the test once
}

//end timing
clock_t end = clock();

//calculate elapsed time
double elapsed_time = double(end - start) / double(CLOCKS_PER_SEC);
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult and time consuming
- 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
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
- Goal: characterizes running time as a function of the input size \( n \)
Pseudocode Details

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

- **Method declaration**
  - Method call
    - `var.method (arg [, arg...])`
  - Return value
    - `return expression`

- **Expressions**
  - Assignment (like = in C++/Java)
  - Equality testing (like == in C++/Java)
  - \( n^2 \) Superscripts and other mathematical formatting allowed

Example: find max element of an array

```
Algorithm arrayMax(A, n)
Input array A of n integers
Output maximum element of A

currentMax ← A[0]
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]

return currentMax
```

Pseudocode

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

**Algorithm arrayMax(A, n)**

**Input** array A of n integers

**Output** maximum element of A

```
currentMax ← A[0]
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]

return currentMax
```

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant execution time

Examples:
- Performing an arithmetic operation
- Comparing two numbers
- 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)

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

# of operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[0]</code></td>
<td>1</td>
</tr>
<tr>
<td><code>A[i]</code></td>
<td>( n )</td>
</tr>
<tr>
<td><code>currentMax = A[i]</code></td>
<td>( n )</td>
</tr>
<tr>
<td><code>i = i + 1</code></td>
<td>( n − 1 )</td>
</tr>
<tr>
<td><code>if A[i] &gt; currentMax</code></td>
<td>1</td>
</tr>
<tr>
<td><code>currentMax = A[i]</code></td>
<td>( n − 1 )</td>
</tr>
<tr>
<td><code>i = i + 1</code></td>
<td>( n − 1 )</td>
</tr>
<tr>
<td><code>return</code></td>
<td>1</td>
</tr>
</tbody>
</table>

Total \( 8n - 2 \)
Worst case analysis

- Average case analysis is difficult for many problems:
  - Probability distribution of inputs.
- We focus on the worst case analysis:
  - Easier
  - If an algorithm does well in the worst-case, it will perform well on all cases.

Estimating Running Time

- Algorithm `arrayMax` executes \(8n - 2\) 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
  \[
  a(8n - 2) \leq T(n) \leq b(8n - 2)
  \]
- 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`.

Growth Rates

- Growth rates of functions:
  - Linear = \(n\)
  - Quadratic = \(n^2\)
  - Cubic = \(n^3\)
Growth Rates

- Growth rates of functions:
  - Linear = \( n \)
  - Quadratic = \( n^2 \)
  - Cubic = \( n^3 \)

- In a log-log chart, the slope of the line corresponds to the growth rate of the function.

Constant Factors

- The growth rate is not affected by constant factors or lower-order terms.

Examples

- \( 10^5n + 10^5 \) is a linear function
- \( 10^3n^2 + 10^3n \) is a quadratic function

Big-Oh Notation Example

- 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 Notation Example (cont.)

- 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.

![Graph showing the comparison of various functions including $n^2$, $100n$, $10n$, and $n$.]

More Big-Oh Examples

- **7n-2**
  7n-2 is $O(n)$
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq cn$ 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 $3n^3 + 20n^2 + 5 \leq cn^3$ 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 $3 \log n + 5 \leq c \log n$ 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 “$f(n)$ 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>

Classes of Functions

- Let $\{g(n)\}$ denote the class (set) of functions that are $O(g(n))$
- We have $\{n\} \subset \{n^2\} \subset \{n^3\} \subset \{n^4\} \subset \{n^5\} \subset \ldots$
- where the containment is strict
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
  1. We find the worst-case number of primitive operations executed as a function of the input size
  2. We express this function with big-Oh notation
- Example:
  1. We determine that algorithm \( \text{arrayMax} \) executes at most \( 8n - 2 \) primitive operations
  2. We say that algorithm \( \text{arrayMax} \) "runs in \( O(n) \) time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  1. Constant = 1
  2. Logarithmic = \( \log n \)
  3. Linear = \( n \)
  4. N-Log-N = \( n \log n \)
  5. Quadratic = \( n^2 \)
  6. Cubic = \( n^3 \)
  7. Exponential = \( 2^n \)
Asymptotic Analysis

<table>
<thead>
<tr>
<th>Running Time</th>
<th>Maximum Problem Size (n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 second</td>
<td>1 minute</td>
</tr>
<tr>
<td>20nlogn</td>
<td>4,096</td>
</tr>
<tr>
<td>2n^2</td>
<td>707</td>
</tr>
<tr>
<td>n^4</td>
<td>31</td>
</tr>
<tr>
<td>2^n</td>
<td>19</td>
</tr>
</tbody>
</table>

Caution: $10^{100}n$ vs. $n^2$

Algorithm Analysis

Computing Prefix Averages

- We 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 $X$
  \[ A[i] = (X[0] + X[1] + \ldots + X[i]) / (i + 1) \]
- Problem: compute the array $A$ of prefix averages of another array $X$
- Applications in economics and statistics

Prefix Averages (Quadratic)

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>prefixAverages1($X, n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>array $X$ of $n$ integers</td>
</tr>
<tr>
<td>Output</td>
<td>array $A$ of prefix averages of $X$ #operations</td>
</tr>
<tr>
<td></td>
<td>$n$ integers</td>
</tr>
<tr>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td>for $i$ 0 to $n - 1$ do</td>
<td></td>
</tr>
<tr>
<td>$s \leftarrow X[0]$</td>
<td></td>
</tr>
<tr>
<td>for $j$ 1 to $i$ do</td>
<td></td>
</tr>
<tr>
<td>$s \leftarrow s + X[j]$</td>
<td></td>
</tr>
<tr>
<td>$A[i] \leftarrow s / (i + 1)$</td>
<td></td>
</tr>
<tr>
<td>return $A$</td>
<td></td>
</tr>
</tbody>
</table>

Arithmetic Progression

- The running time of $\text{prefixAverages1}$ is \( O(1 + 2 + \ldots + n) \)
  or \( O(\frac{n(n+1)}{2}) \)
- Thus, the algorithm $\text{prefixAverages1}$ runs in $O(n^2)$ time
Prefix Averages (Linear)

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

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X #operations
    A ← new array of n integers
    s ← 0
    for i ← 0 to n – 1 do
        s ← s + X[i]
        A[i] ← s / (i + 1)
    return A
```

Algorithm prefixAverages2 runs in $O(n)$ time

Relatives of Big-Oh, Intuition for Asymptotic Notation

**Big-Oh**
- $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$

**Big-Omega**
- $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$
  - $f(n)$ 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 $n \geq n_0$

**Big-Theta**
- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$
  - $f(n)$ 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 $n \geq n_0$