Analysis of Algorithms

Input → Algorithm → Output
Scalability

- Scientists often have to deal with differences in scale, from the microscopically small to the astronomically large.
- Computer scientists must also deal with scale, but they deal with it primarily in terms of data volume rather than physical object size.
- **Scalability** refers to the ability of a system to gracefully accommodate growing sizes of inputs or amounts of workload.
Application: Job Interviews

- High technology companies tend to ask questions about **algorithms and data structures** during job interviews.
- Algorithms questions can be short but often require critical thinking, creative insights, and subject knowledge.
  - All the “Applications” exercises in Chapter 1 of the Goodrich-Tamassia textbook are taken from reports of actual job interview questions.


Used with permission under Creative Commons 2.5 License.
Algorithms and Data Structures

- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
  - Typically, an algorithm takes input data and produces an output based upon it.

- A **data structure** is a systematic way of organizing and accessing data.

---

Spring 2020

CMSC 315
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 primarily 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, noting the time needed:
- 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
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
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**
  `method (arg [, arg...])`

- **Return value**
  `return expression`

- **Expressions:**
  - Assignment
  - Equality testing
  - $n^2$ Superscripts and other mathematical formatting allowed
The Random Access Machine (RAM) Model

A **RAM** consists of

- 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

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

Spring 2020

CMSC 315
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

- Example: 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):

Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currentMax \leftarrow A[0]$

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

if $currentMax < A[i]$ then

$currentMax \leftarrow A[i]$

return $currentMax$
Estimating Running Time

- Algorithm `arrayMax` executes $7n - 2$ primitive operations in the worst case, $5n$ in the best 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 `arrayMax`. Then
  
  $a(5n) \leq T(n) \leq b(7n - 2)$

- 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 2 n</th>
<th>time for 4 n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cn\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>2(cn)</td>
<td>4(cn)</td>
</tr>
<tr>
<td>(cn \lg n)</td>
<td>(\sim c n \lg n + c n)</td>
<td>2(c n \lg n + 2c n)</td>
<td>4(c n \lg n + 4c n)</td>
</tr>
<tr>
<td>(cn^2)</td>
<td>(\sim c n^2 + 2c n)</td>
<td>(4c n^2)</td>
<td>16(c n^2)</td>
</tr>
<tr>
<td>(cn^3)</td>
<td>(\sim c n^3 + 3c n^2)</td>
<td>8(c n^3)</td>
<td>64(c 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>

Runtime quadruples when problem size doubles.
Analyzing Recursive Algorithms

- Use a function, T(n), to derive a recurrence relation that characterizes the running time of the algorithm in terms of smaller values of n.

```
Algorithm recursiveMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
    if n = 1 then
        return A[0]
    return max{recursiveMax(A, n - 1), A[n - 1]}
```

\[ T(n) = \begin{cases} 
3 & \text{if } n = 1 \\
T(n - 1) + 7 & \text{otherwise,}
\end{cases} \]
Constant Factors

- The growth rate is minimally 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 $n_0 = 10$
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 ≥ 1 such that 7 n - 2 ≤ c n for n ≥ n_0
  
  this is true for c = 7 and n_0 = 1

- **3 n^3 + 20 n^2 + 5**
  
  3 n^3 + 20 n^2 + 5 is O(n^3)
  
  need c > 0 and n_0 ≥ 1 such that 3 n^3 + 20 n^2 + 5 ≤ c n^3 for n ≥ 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 ≥ 1 such that 3 log n + 5 ≤ c log n for n ≥ 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>

Spring 2020
CMSC 315
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 $O(3n)$”
Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  \( a^{(b+c)} = a^b a^c \)
  \( a^{bc} = (a^b)^c \)
  \( a^b / a^c = a^{(b-c)} \)

- Properties of logarithms:
  \( \log_b(xy) = \log_b x + \log_b y \)
  \( \log_b (x/y) = \log_b x - \log_b y \)
  \( \log_b x^a = a \log_b x \)
  \( \log_b a = \log_x a / \log_x b \)
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
  \[ f(n) \geq c \ g(n) \text{ for } n \geq n_0 \]

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
  \[ c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0 \]
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 \( g(n) \)

**big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically equal to \( g(n) \)
Example Uses of the 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 \( n_0 = 1 \)