ANALYSIS OF ALGORITHMS

EVALUATING PROGRAM EFFICIENCY

- What’s the best way to program a solution to a problem?
- Efficiency in terms of
  - Time (running time)
  - Space (memory requirements)
  - Resources (Input/Output such as disk I/O)
  - Energy consumption
- Most analysis focuses on time efficiency
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, noting the time needed:
- Plot the results

```java
1 long startTime = System.currentTimeMillis();    // record the starting time
2 */ (run the algorithm) */
3 long endTime = System.currentTimeMillis();      // record the ending time
4 long elapsed = endTime - startTime;           // compute the elapsed time
```
 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.

- Measured times reported by the system may likely vary from trial to trial, even on the same machine and input.
  - Processes share CPU and memory.

- In order to compare two algorithms, the same hardware and software environments must be used.
EXAMPLE

```java
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j = 0; j < n; j++)
        answer += c;
    return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder();
    for (int j = 0; j < n; j++)
        sb.append(c);
    return sb.toString();
}
```

repeat("*", 40)

"*****************************************

Figure 4.1: Chart of the results of the timing experiment from Code Fragment 4.2, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.
GOALS OF ANALYZING THE EFFICIENCY OF ALGORITHMS

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.

2. Is performed by studying a high-level description of the algorithm without need for implementation.

3. Takes into account all possible inputs.
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
F.Y.I.: 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:
- Assigning a value to a variable
- Following an object reference
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of an array by index
- Calling a method
- Returning from a method
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
F.Y.I.: 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 of pseudocode (for the mathematical game fizz buzz):

<table>
<thead>
<tr>
<th>Fortran style pseudo code</th>
<th>Pascal style pseudo code</th>
<th>C style pseudo code:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>program fizzbuzz</strong></td>
<td><strong>procedure fizzbuzz</strong></td>
<td><strong>void function fizzbuzz</strong></td>
</tr>
<tr>
<td>Do i = 1 to 100</td>
<td>For i := 1 to 100 do</td>
<td>For (i = 1; i &lt;= 100; i++) {</td>
</tr>
<tr>
<td>set print_number to true</td>
<td>set print_number to true;</td>
<td>set print_number to true;</td>
</tr>
<tr>
<td>If i is divisible by 3</td>
<td>If i is divisible by 3 then</td>
<td>If i is divisible by 3</td>
</tr>
<tr>
<td>print &quot;Fizz&quot;</td>
<td>print &quot;Fizz&quot;;</td>
<td>print &quot;Fizz&quot;;</td>
</tr>
<tr>
<td>set print_number to false</td>
<td>set print_number to false;</td>
<td>set print_number to false;</td>
</tr>
<tr>
<td>If i is divisible by 5</td>
<td>If i is divisible by 5 then</td>
<td>If i is divisible by 5</td>
</tr>
<tr>
<td>print &quot;Buzz&quot;</td>
<td>print &quot;Buzz&quot;;</td>
<td>print &quot;Buzz&quot;;</td>
</tr>
<tr>
<td>set print_number to false</td>
<td>set print_number to false;</td>
<td>set print_number to false;</td>
</tr>
<tr>
<td>If print_number, print i</td>
<td>If print_number, print i;</td>
<td>If print_number, print i;</td>
</tr>
<tr>
<td>print a newline</td>
<td>print a newline;</td>
<td>print a newline;</td>
</tr>
<tr>
<td>end do</td>
<td>end</td>
<td>}</td>
</tr>
</tbody>
</table>

http://en.wikipedia.org/wiki/Pseudocode
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 `=`
  - Superscripts and other mathematical formatting allowed $n^2$
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.

```java
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0]; // assume first entry is biggest (for now)
    for (int j=1; j < n; j++) { // consider all other entries
        if (data[j] > currentMax) // if data[j] is biggest thus far...
            currentMax = data[j]; // record it as the current max
    }
    return currentMax;
}
```

- Step 3: 2 ops, 4: 2 ops, 5: 2n ops,
  6: 2n ops, 7: 0 to n ops, 8: 1 op
ESTIMATING RUNNING TIME

Focus on the growth rate of the running time as a function of the input size $n$, taking a “big-picture” approach.

- Algorithm `arrayMax` executes $5n + 5$ primitive operations in the worst case, $4n + 5$ 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
  
  $a (4n + 5) \leq T(n) \leq b(5n + 5)$

- Hence, the running time $T(n)$ is bounded by two linear functions
Changing the hardware/software environment
+ Affects $T(n)$ by a constant factor, but
+ Does not alter the growth rate of $T(n)$

$$a \ (4n + 5) \leq T(n) \leq b(5n + 5)$$

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>$\sim cn \lg n + cn$</td>
<td>$2cn \lg n + 2cn$</td>
<td>$4cn \lg n + 4cn$</td>
</tr>
<tr>
<td>$cn^2$</td>
<td>$\sim cn^2 + 2cn$</td>
<td>$4cn^2$</td>
<td>$16cn^2$</td>
</tr>
<tr>
<td>$cn^3$</td>
<td>$\sim cn^3 + 3cn^2$</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**
F.Y.I.: 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 $\approx b^n$

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

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \cdots + a_dn^d$$

Polynomials
THE SEVEN FUNCTIONS GRAPHED USING “NORMAL” SCALE

- **g(n) = 1**
  - Constant

- **g(n) = \log n**
  - Logarithmic

- **g(n) = n**
  - Linear

- **g(n) = n \log n**
  - N-Log-N

- **g(n) = n^2**
  - Quadratic

- **g(n) = 2^n**
  - Exponential

- **g(n) = n^3**
  - Cubic

Slide by Matt Stallmann included with permission.
log-log chart
Running times of loops with increasing terms

\[ \sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b) \]

where \( a \) and \( b \) are integers and \( a \leq b \)

**Proposition 4.3:** For any integer \( n \geq 1 \), we have:

\[ 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}. \]

Loop for which each iteration takes a multiplicative factor longer than the previous one.

**Proposition 4.5:** For any integer \( n \geq 0 \) and any real number \( a \) such that \( a > 0 \) and \( a \neq 1 \), consider the summation

\[ \sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n \]

(remembering that \( a^0 = 1 \) if \( a > 0 \). This summation is equal to

\[ \frac{a^{n+1} - 1}{a - 1}. \]
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
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
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
BIG-OH NOTATION

Focus on the growth rate of the running time as a function of the input size \( n \), taking a “big-picture” approach.

Let \( f(n) \) and \( g(n) \) be functions mapping positive integers to positive real numbers. We say that \( f(n) \) is \( O(g(n)) \) if there is a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that

\[
f(n) \leq c \cdot g(n), \quad \text{for} \quad n \geq n_0.
\]

This definition is often referred to as the “big-Oh” notation, for it is sometimes pronounced as “\( f(n) \) is big-Oh of \( g(n) \).”

Example: \( 2n + 10 \) is \( O(n) \)

\[
+ \quad 2n + 10 \leq cn
\]

\[
+ \quad (c - 2) \cdot n \geq 10
\]

\[
+ \quad n \geq 10/(c - 2)
\]

+ 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 n₀ ≥ 1 such that 7 n - 2 ≤ c n for n ≥ n₀
  this is true for c = 7 and n₀ = 1

- **3 n³ + 20 n² + 5**
  3 n³ + 20 n² + 5 is O(n³)
  need c > 0 and n₀ ≥ 1 such that 3 n³ + 20 n² + 5 ≤ c n³ for n ≥ n₀
  this is true for c = 4 and n₀ = 21

- **3 log n + 5**
  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
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 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

**Proposition 4.8:** If \( f(n) \) is a polynomial of degree \( d \), that is,

\[
f(n) = a_0 + a_1 n + \cdots + a_d n^d,
\]

and \( a_d > 0 \), then \( f(n) \) is \( O(n^d) \).

- 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.

Example:
- 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.
COMPARATIVE ANALYSIS

- What’s considered a better algorithm?
  + Asymptotically faster algorithm that solves the same problem.
  + NOTE: Although we ignore constants, large constants can be effect the time significantly in real programs.

- What’s considered fast?
  + Generally, algorithms that runs within $O(n \log n)$ is considered fast.
  + If we need to divide algorithms to tractable and intractable, the borderline will be polynomial ($n^k$) vs exponential ($a^n$)
EXAMPLE: TREE-WAY SET DISJOINTNESS

PROBLEM(three-way set disjointness): Given three sets, A, B, and C, that contains no duplicate values, determine if the intersection of the three sets is empty, namely, that there is no element x such that x ∈ A, x ∈ B, and x ∈ C.

```java
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3      for (int a : groupA)
4          for (int b : groupB)
5              for (int c : groupC)
6                  if ((a == b) && (b == c))
7                      return false; // we found a common value
8                  return true; // if we reach this, sets are disjoint
9  }
```

$O(n^3)$
HINT: There are quadratically many pairs \((a,b)\) to consider. However, if \(A\) and \(B\) are each sets of distinct elements, there can be at most \(O(n)\) such pairs with \(a\) equal to \(b\). Therefore, the inner most loop, over \(C\), executes at most \(n\) times. \(O(n^2)\)
EXAMPLE: COMPUTING PREFIX AVERAGES

PROBLEM: The $i$-th prefix average of an array $X$ is average of the first $(i + 1)$ elements of $X$:

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

Computing the array $A$ of prefix averages of another array $X$

APPLICATIONS: Given a stream of daily Web usage logs, a website manager may wish to track average usage trends over various time periods.
Prefix Averages (Quadratic time)

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

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
    int n = x.length;
    double[ ] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        total += x[j]; // record the average
        a[j] = total / (j+1);
    }
    return a;
}
```
The running time of \texttt{prefixAverage1} 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{prefixAverage1} runs in \( O(n^2) \) time
Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```java
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[] prefixAverage2(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    double total = 0; // compute prefix sum as x[0] + x[1] + ...
    for (int j=0; j < n; j++) {
        total += x[j]; // update prefix sum to include x[j]
        a[j] = total / (j+1); // compute average based on current sum
    }
    return a;
}
```

Algorithm `prefixAverage2` runs in \(O(n)\) time!
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
  
  $$f(n) \geq c \cdot g(n) \text{ for } n \geq n_0$$

**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 ASYMPOTOTIC 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 $n_0 ≥ 1$ such that $f(n) ≥ c g(n)$ for $n ≥ n_0$

  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 $n_0 ≥ 1$ such that $f(n) ≥ c g(n)$ for $n ≥ n_0$

  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 $n_0 ≥ 1$ such that $f(n) ≤ c g(n)$ for $n ≥ n_0$

  Let $c = 5$ and $n_0 = 1$
MATH YOU NEED TO REVIEW

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

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

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