Analysis of Algorithms

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, noting the time needed:
- Plot the results

```java
long startTime = System.currentTimeMillis(); // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis(); // record the ending time
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.
- 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

© 2014 Goodrich, Tamassia, Goldwasser

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

© 2014 Goodrich, Tamassia, Goldwasser
Seven Important Functions

- 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

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

```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

- 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.

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>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</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>4c n^2</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>

runtime quadruples when problem size doubles

Comparison of Two Algorithms

insertion sort is \(n^2/4\)
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:
  - $10^5n + 10^2$ is a linear function.
  - $10^2n^2 + 10^n$ 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 $7n - 2 \leq cn$ 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>

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)”
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 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 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.
Prefix Averages (Quadratic)

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

```java
public static double[] prefixAverage1(double[] x) {
    int n = x.length;
    double[] a = new double[n]; // filled with zeros by default
    double total = 0;
    for (int i = 0; i < n; i++) {
        total += x[i];
    }
    for (int j = 0; j < n; j++) {
        a[j] = total / (j + 1); // record the average
    }
    return a;
}
```

Arithmetic Progression

- The running time of `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 `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!

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:**
  \[
  \log_b(xy) = \log_b x + \log_b y, \\
  \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y, \\
  \log_b x^a = a \log_b x, \\
  \log_b a = \log_k a / \log_k b
  \]
Relatives of Big-Oh

\textbf{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 \( n \geq n_0 \)

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

Intuition for Asymptotic Notation

\textbf{big-Oh}

- \( f(n) \) is \( O(g(n)) \) if \( f(n) \) is asymptotically less than or equal to \( g(n) \)

\textbf{big-Omega}

- \( f(n) \) is \( \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

\textbf{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 $\Omega(n^2)$
  $f(n)$ 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 $n_0 = 1$