Mathematical Background

Logarithms

Summations

Recursion

Induction Proofs

Recurrence Relations
Logarithm

Definition:

Logarithms have the following properties, for any positive values of \( m, n, \) and \( r, \) and any positive integers \( a \) and \( b. \)

1. \( \log(nm) = \log n + \log m. \)
2. \( \log(n/m) = \log n - \log m. \)
3. \( \log(n^r) = r \log n. \)
4. \( \log_a n = \log_b n / \log_b a. \)
What’s the algebraic equation for runtime?

\[ T(n) = T(n - 1) + n; \quad T(1) = 1; \]
Recursion

An algorithm is recursive if it calls itself to do part of its work.

Example:

1. Compute n!

```java
long fact(int n) { // Compute n! recursively
    // To fit n! into a long variable, we require n <= 12
    Assert((n >= 0) && (n <= 12), "Input out of range");
    if (n <= 1) return 1; // Base case: return base solution
    return n * fact(n-1); // Recursive call for n > 1
}
```

2. Hanoi puzzle
Mathematical Proof

Three templates for mathematical proof

1. Direct Proof
2. Proof by Contradiction
3. Proof by mathematical induction

E.g., prove that 2¢ and 5¢ stamps can be used to form any value (for values ≥ 4).
Estimation Techniques

Known as “back of the envelope” or “back of the napkin” calculation

Determine the major parameters that affect the problem.

Derive an equation that relates the parameters to the problem.

Select values for the parameters, and apply the equation to yield estimated solution.
Estimation Example

How many library bookcases does it take to store books totaling one million pages?

Estimate: How many library bookcases does it take to store books containing one million pages?

• Pages/inch
• Feet/shelf
• Shelves/bookcase
There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer’s resources.
Algorithm Efficiency (cont)

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm’s cost?
How to Measure Efficiency?

Empirical comparison (run programs)

Asymptotic Algorithm Analysis

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on “size” of the input.

Running time is expressed as $T(n)$ for some function $T$ on input size $n$. 
Examples of Growth Rate

Example 1.

/** @return Position of largest value in "A" */
static int largest(int[] A) {
    int currlarge = 0;  // Position of largest
    for (int i=1; i<A.length; i++)
        if (A[currlarge] < A[i])
            currlarge = i; // Remember pos
    return currlarge;   // Return largest pos
}
Examples (cont)

Example 2: Assignment statement.

Example 3:

```c
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum++;
```

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for $K$ in an array of $n$ integers:
  • Begin at first element in array and look at each element in turn until $K$ is found

Best case:

Worst case:

Average case:
Which Analysis to Use?

While average time appears to be the fairest measure, it may be difficult to determine.

When is the worst case time important?
Faster Computer or Algorithm?

Suppose we buy a computer 10 times faster.

$n$: size of input that can be processed in one second on old computer

$n'$: size of input that can be processed in one second on new computer

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>n</th>
<th>$n'$</th>
<th>Change</th>
<th>$n'/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>1000</td>
<td>10,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$20n$</td>
<td>500</td>
<td>5000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$5n \log n$</td>
<td>250</td>
<td>1842</td>
<td>$\sqrt{10n} &lt; n' &lt; 10n$</td>
<td>7.37</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>70</td>
<td>223</td>
<td>$n' = \sqrt{10n}$</td>
<td>3.16</td>
</tr>
<tr>
<td>$2^n$</td>
<td>13</td>
<td>16</td>
<td>$n' = n + 3$</td>
<td>--</td>
</tr>
</tbody>
</table>
Asymptotic Analysis: Big-oh for Upper Bound

Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

Use: The algorithm is in $O(n^2)$ in [best, average, worst] case.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps in [best, average, worst] case.
Big-oh Notation (cont)

Big-oh notation indicates an upper bound.

Example: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.

Look for the tightest upper bound:

While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$. 
Big-Oh Examples

Example 1: Finding value $X$ in an array (average cost).

Then $T(n) = c_s n/2$.

For all values of $n > 1$, $c_s n/2 \leq c_s n$.

Therefore, the definition is satisfied for $f(n) = n$, $n_0 = 1$, and $c = c_s$.

Hence, $T(n)$ is in $O(n)$. 
Example 2: Suppose $T(n) = c_1 n^2 + c_2 n$, where $c_1$ and $c_2$ are positive.

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

Then $T(n) \leq cn^2$ whenever $n > n_0$, for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $T(n)$ is in $O(n^2)$ by definition.

Example 3: $T(n) = c$. Then $T(n)$ is in $O(1)$.
A Common Misunderstanding

“The best case for my algorithm is $n=1$ because that is the fastest.” WRONG!

Big-oh refers to a growth rate as $n$ grows to $\infty$.

Best case is defined for the input of size $n$ that is cheapest among all inputs of size $n$. 
Big-Omega for Lower Bound

Definition: For \( T(n) \) a non-negatively valued function, \( T(n) \) is in the set \( \Omega(g(n)) \) if there exist two positive constants \( c \) and \( n_0 \) such that \( T(n) \geq cg(n) \) for all \( n > n_0 \).

Meaning: For all data sets big enough (i.e., \( n > n_0 \)), the algorithm always requires more than \( cg(n) \) steps.

Lower bound.
Big-Omega Example

\[ T(n) = c_1 n^2 + c_2 n. \]

\[ c_1 n^2 + c_2 n \geq c_1 n^2 \text{ for all } n > 1. \]

\[ T(n) \geq c n^2 \text{ for } c = c_1 \text{ and } n_0 = 1. \]

Therefore, \( T(n) \) is in \( \Omega(n^2) \) by the definition.

We want the greatest lower bound.
Theta Notation

When big-Oh and $\Omega$ coincide, we indicate this by using $\Theta$ (big-Theta) notation.

Definition: An algorithm is said to be in $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$. 
A Common Misunderstanding

Confusing worst case with upper bound.

Upper bound refers to a growth rate.

Worst case refers to the worst input from among the choices for possible inputs of a given size.
Simplifying Rules

If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.

If $f(n)$ is in $O(kg(n))$ for some constant $k > 0$, then $f(n)$ is in $O(g(n))$.

If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.

If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$. 
**Time Complexity Examples (1)**

Example 3.9: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 3.10:

```plaintext
sum = 0;
for (i=1; i<=n; i++)
    sum += n;
```
Example 3.11:

```c
sum = 0;
for (j=1; j<=n; j++)
    for (i=1; i<=j; i++)
        sum++;
for (k=0; k<n; k++)
    A[k] = k;
```
Time Complexity Examples (3)

Example 3.12:

```c
sum1 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
        sum2++;
```
Example 3.13:

\[
\text{sum1} = 0; \\
\text{for (k=1; k<=n; k*=2)} \\
\hspace{1em} \text{for (j=1; j<=n; j++)} \\
\hspace{2em} \text{sum1++;} \\
\text{sum2} = 0; \\
\text{for (k=1; k<=n; k*=2)} \\
\hspace{1em} \text{for (j=1; j<=k; j++)} \\
\hspace{2em} \text{sum2++;}
\]

Example: \( T(n) = T(n - 1) + 1 \) for \( n > 1 \); \( T(1) = 0 \):
Binary Search

How many elements are examined in worst case?
/** @return The position of an element in sorted array A with value k.  If k is not in A, return A.length. */
static int binary(int[] A, int k) {
int l = -1;        // Set l and r
int r = A.length;  // beyond array bounds
while (l+1 != r) { // Stop when l, r meet
    int i = (l+r)/2; // Check middle
    if (k == A[i]) return i; // Found it
    if (k < A[i]) r = i;     // In left half
    if (k > A[i]) l = i;     // In right half
}
return A.length; // Search value not in A}
Other Control Statements

**while** loop: Analyze like a **for** loop.

**if** statement: Take greater complexity of **then**/**else** clauses.

**switch** statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.
Problems

Problem: a task to be performed.

- Best thought of as inputs and matching outputs.
- Problem definition should include constraints on the resources that may be consumed by any acceptable solution.
Problems (cont)

Problems ⇔ mathematical functions

- A function is a matching between inputs (the domain) and outputs (the range).
- An input to a function may be single number, or a collection of information.
- The values making up an input are called the parameters of the function.
- A particular input must always result in the same output every time the function is computed.
Problems and Algorithms

**Algorithm**: a method or a process followed to solve a problem.
- A recipe.

An algorithm takes the input to a problem (function) and transforms it to the output.
- A mapping of input to output.

A problem can have many algorithms.
Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for every possible algorithm.
(The least possible cost for any algorithms in the worst case)
Analyzing Problems: Example

Example of imperfect knowledge: Sorting

1. Cost of examining every element: $\Omega(n)$. (lower bound)
2. Bubble or insertion sort: $O(n^2)$.
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$.
4. We prove later that sorting is in $\Omega(n \log n)$. (upper bound)
Space Complexity

Space complexity can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Data Structure
Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  - Boolean flags
- Table lookup
  - Factorials

Example: Binsort (trade time for space)

```c
for (i=0; i<n; i++)
  B[A[i]] = A[i];
```

```c
for (i=0; i<n; i++)
  while (A[i] != i)
    swap(A, i, A[i]);
```
Compute the rank ordering for all $C$-pixel values in a picture of $P$ pixels.

for (i=0; i<C; i++)  // Initialize count
    count[i] = 0;
for (i=0; i<P; i++)  // Look at all pixels
    count[value(i)]++; // Increment count
sort(count);       // Sort pixel counts

Time complexity is $\Theta(P + C \log C)$. 