CSE 146

Asymptotic Analysis
Interview Question of the Day
Homework 1 & Project 1 Work Session
Comparing Algorithms

- Rough Estimate
- Ignores Details
  - Or really: independent of details
- What are some details we should ignore?
Big-O Analysis

• What are some details we should ignore?
  – Speed of machine
  – Programming language used
  – Amount of memory
  – Order of input
  – Size of input
  – Compiler
Analysis of Algorithms

- Efficiency measure
  - How long the program runs (time complexity)
  - How much memory it uses (space complexity)

- Why analyze at all?
  - Decide which one to implement before going to the trouble
  - Given code, idea of where bottlenecks will be – without running and timing
Asymptotic Analysis

- One “detail” we won't ignore – problem size, # elements
- Complexity as a function of input size n
  - $T(n) = 4n+5$
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$
- What happens as n grows?
Why Asymptotic Analysis?

- We'd like to be able to compare two functions
- Most algorithms are fast for small $n$
  - Time difference too small to be noticeable
  - External things dominate (OS, disk I/O, ...)
- BUT $n$ is often large in practice
  - Databases, internet, graphics, ...
- Time difference really shows up as $n$ grows!
Big-O Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$
- linear: $O(n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ (k is a constant)
- exponential: $O(c^n)$ (c is a constant > 1)
Exercise

bool ArrayFind (int array[ ], int n, int key) {

}
Linear Search Analysis

bool LinearArrayFind(int array[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (array[i] == key)
            // Found it!
            return true;
    }
    return false;
}

Best case?
Worst case?
Analyzing Code

- **Basic Java operations** - Constant Time
- **Consecutive statements** – Sum of times
- **Conditionals** – Larger branch plus test
- **Loops** – Sum of iterations
- **Function calls** – Cost of function body
- **Recursive functions** – Solve recurrence relation, Number of calls * work for each call
bool BinArrayFind(int array[], int low, int high, int key) {

    // The subarray is empty
    if( low > high ) return false;

    //Search this subarray recursively
    int mid = (high + low) /2;
    if( key == array[mid]) return true;
    else if (key < array[mid]){ return BinArrayFind( array, low, mid-1, key); }

    else {
        return BinArrayFind( array, mid+1, high, key);
    }
}

Solving Recurrence Relations

- It takes $O(1)$ time to do the comparisons, then it cuts the search range in half
  - $T(N) = T(N/2) + 1$

- Repeat the recurrence (basically expanding the relation)...  
  - $T(N) = T(N/4) + 2$
  - $= T(N/8) + 3$
  - $= T(N/2^k) + k$

- Round up $N$ to the nearest power of 2: $N \leq 2^m$
  - $T(N) \leq T(2^m/2^k) + k$

- Let $k = m$
  - $T(N) \leq T(2^m/2^m) + m = T(1) + m = 1 + m = O(m)$
  - If $N=2^m$, then making $m = \log N$. So $T(N) = O(\log N)$
Linear Search vs Binary Search

- Linear: Best case 4 at [0], Worst 3n+2
- Binary: Best case 4 at [mid], Worst 4 log n +4

Which algorithm is better? What tradeoffs can you make?
Asymptotic Analysis

• Asymptotic analysis looks at the order of the running time of the algorithm
  – A valuable tool when the input gets “large”
  – Ignores the effects of different machines or different implementations of the same algorithm

• Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  – Linear search is $T(n) = 3n + 2$ becomes $O(\cdot)$
  – Binary search is $T(n) = 4 \log n + 4$ becomes $O(\cdot)$
Remember

- The fastest algorithm has the slowest growing function for its runtime
Asymptotic Analysis

• Eliminate low order terms
  - $4n + 5 \Rightarrow$
  - $0.5 \, n \log n + 2n + 7 \Rightarrow$
  - $n^3 + 2^n + 3n \Rightarrow$

• Eliminate coefficients
  - $4n \Rightarrow$
  - $0.5 \, n \log n \Rightarrow$
  - $n \log n^2 \Rightarrow$
Properties of logs

- We will assume logs to base 2 unless specified otherwise.
- \( \log AB = \log A + \log B \)
- \( \log(A/B) = \log A – \log B \)
- \( \log(A^B) = B \log A \)
- Any base \( k \) log is equivalent to base 2.
Definition of Order Notation

• Upper bound: \( T(n) = O(f(n)) \) \hspace{1cm} \text{Big-O}
  – Exist constants \( c \) and \( n' \) such that
    • \( T(n) \leq c f(n) \) for all \( n \geq n' \)

• Lower bound: \( T(n) = \Omega(g(n)) \) \hspace{1cm} \text{Omega}
  – Exist constants \( c \) and \( n' \) such that
    • \( T(n) \geq c g(n) \) for all \( n \geq n' \)

• Tight bound \( T(n) = \Theta(f(n)) \) \hspace{1cm} \text{Theta}
  – When both hold: \( T(n) = O(f(n)) \) and \( T(n) = \Omega(f(n)) \)
Order Notation: Definition

O(f(n)) : a set or class of functions

g(n) ∈ O(f(n)) iff there exist consts c and n₀ such that:
g(n) ≤ c f(n) for all n ≥ n₀

.... homework 1
Example

- $g(n) = 1000n$ vs. $f(n) = n^2$
- Is $g(n) \in O(f(n))$?
  - Pick: $n_0 = 1000$, $c = 1$
  - $1000n \leq 1 \cdot n^2$ for all $n \geq 1000$
  - So $g(n) \in O(f(n))$

- Small cases, really don't matter. As long as it's eventually an upper bound, it fits the definition
- If $f(n)$ is in $O(n)$... what about is $f(n)$ in $O(n^2)$?
Notation Notes

- Note: Sometimes, you’ll see the notation:
  - \( g(n) = O(f(n)) \).
- This is equivalent to:
  - \( g(n) \in O(f(n)) \).
Order Notation: Example

\[100n^2 + 1000 \leq 5(n^3 + 2n^2) \text{ for all } n \geq 19\]

So \(f(n) \in O(g(n))\)
Meet the Family

- $O( f(n) )$ is the set of all functions asymptotically less than or equal to $f(n)$
  - $o( f(n) )$ is the set of all functions asymptotically strictly less than $f(n)$

- $\Omega( f(n) )$ is the set of all functions asymptotically greater than or equal to $f(n)$
  - $\omega( f(n) )$ is the set of all functions asymptotically strictly greater than $f(n)$

- $\Theta( f(n) )$ is the set of all functions asymptotically equal to $f(n)$
Meet the Family, Formally

- $g(n) \in O(f(n))$ iff There exist $c$ and $n_0$ such that $g(n) \leq c f(n)$ for all $n \geq n_0$
  - $g(n) \in o(f(n))$ iff There exists a $n_0$ such that $g(n) < c f(n)$ for all $c$ and $n \geq n_0$

- $g(n) \in \Omega(f(n))$ iff There exist $c$ and $n_0$ such that $g(n) \geq c f(n)$ for all $n \geq n_0$
  - $g(n) \in \omega(f(n))$ iff There exists a $n_0$ such that $g(n) > c f(n)$ for all $c$ and $n \geq n_0$

- $g(n) \in \Theta(f(n))$ iff $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$
Big-Omega et al. Intuitively

<table>
<thead>
<tr>
<th>Asymptotic Notation</th>
<th>Mathematics Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O$</td>
<td>$\leq$</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>$\geq$</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>$=$</td>
</tr>
<tr>
<td>$o$</td>
<td>$&lt;$</td>
</tr>
<tr>
<td>$\omega$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>
Kinds of Asymptotic Analysis

- Running time may depend on **actual data** input, not just **length of input**
- Distinguish
  - worst case
    - your worst enemy is choosing input
  - best case
  - average case
    - assumes some probabilistic distribution of inputs
  - amortized
    - average time over many operations
Types of Analysis

• Two orthogonal axes:
  - **bound flavor**
    • upper bound (O, o)
    • lower bound (Ω, ω)
    • asymptotically tight (θ)
  - **analysis case**
    • worst case (adversary)
    • average case
    • best case
    • “amortized”
Consider the following program segment:

\[
\begin{align*}
x &:= 0; \\
\text{for } i = 1 \text{ to } N \text{ do} \\
& \quad \text{for } j = 1 \text{ to } N \text{ do} \\
& \quad \quad x := x + 1; \\
\end{align*}
\]

What is the value of \( x \) at the end?
Analyzing the Loop

- Total number of times x is incremented is executed =

\[
1 + 2 + 3 + \ldots = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}
\]

- Congratulations - You’ve just analyzed your first program!
  - Running time of the program is proportional to \(N(N+1)/2\) for all \(N\)
  - Big-O ??
Which Function Grows Faster

\[ n^3 + 2n^2 \quad \text{vs.} \quad 100n^2 + 1000 \]
Which Function Grows Faster

\[ n^3 + 2n^2 \quad \text{vs.} \quad 100n^2 + 1000 \]
Which Function Grows Faster

\[ n^{0.1} \quad \text{vs.} \quad \log n \]
Nested Loops

for i = 1 to n do
    for j = 1 to n do
        sum = sum + 1
Nested Loops

for i = 1 to n do
    for j = 1 to n do
        if (cond) {
            do_stuff(sum)
        } else {
            for k = 1 to n*n
                sum += 1
$16n^3\log_8(10n^2) + 100n^2 = O(n^3\log n)$
$16n^3\log_8(10n^2) + 100n^2 = O(n^3\log n)$

- Eliminate low order terms
  \[
  16n^3\log_8(10n^2) + 100n^2 \\
  \Rightarrow 16n^3\log_8(10n^2)
  \]

- Eliminate constant coefficients
  \[
  \Rightarrow n^3\log_8(10n^2) \\
  \Rightarrow n^3[\log_8(10) + \log_8(n^2)] \\
  \Rightarrow n^3\log_8(10) + n^3\log_8(n^2) \\
  \Rightarrow n^3\log_8(n^2) \\
  \Rightarrow 2n^3\log_8(n) \\
  \Rightarrow n^3\log_8(n) \\
  \Rightarrow n^3\log_8(2)\log(n) \\
  \Rightarrow n^3\log(n) 
  \]
Interview Question of the Day

• We're storing data for 100,000 restaurants. What is the restaurant with the best recommendation?
  – What questions can you ask? How can we do this?

• Now find the 30 best restaurants

• Now find the top N best restaurants
Interview Question of the Day

• We're storing data for 100,000 restaurants. What is the restaurant with the best recommendation?
  – Look at the number of positive reviews left for each restaurant
  – Iterate over the array O(n)

• Now find the 30 best restaurants
  – Iterate over it 30 times... O(n) 30 is constant

• Now find the top N best restaurants
  – Heap (next week :-)

Homework 1 & Project 1 Study Time

- Work session
  - *in groups if you'd like*
- Please don't share answers
- I'll walk around to answer questions – will respond to whole class on common questions