Comparing Two Algorithms

I want you to create an algorithm that sorts all the
students in this class.

Time your algorithm; whoever’s
runs fastest will get stock options

Angelina: adds more memory
Jennifer: discovers optimizing flag on compiler
Angelina: uses C++ instead of Java
Jennifer: faster processor
Angelina: makes data set pre-sorted (that cheating wench!)

What we want

• Rough Estimate
• Ignores Details
  – Or really: independent of details
  – We could have been sneaky & used those
    methods; but both algorithms would benefit
  – Even without an adversary, those factors will
    improve over time
  – So running time, pure and simple, is not a good
    measure

Big-O Analysis

• Ignores “details”
• What are some details we should ignore?
  – Speed of machine
  – Programming language used
  – Amount of memory
  – Order of input
  – Size of input (we’ll talk about this in a second)
  – 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) = 0.5 \, n \log n - 2n + 7
  T(n) = 2^n + n^2 + 3n

  • What happens as $n$ grows?
Why Asymptotic Analysis?

- 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

```java
bool ArrayFind(int array[], int n, int key)
{
    // Insert your algorithm here
}
```

What algorithm would you choose to implement this code snippet?

Analyzing Code

<table>
<thead>
<tr>
<th>Basic Java operations</th>
<th>Constant time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive statements</td>
<td>Sum of times</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Larger branch plus test</td>
</tr>
<tr>
<td>Loops</td>
<td>Sum of iterations</td>
</tr>
<tr>
<td>Function calls</td>
<td>Cost of function body</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>Solve recurrence relation</td>
</tr>
<tr>
<td></td>
<td>Number of calls * work for each call</td>
</tr>
</tbody>
</table>

Analyze your code!

Linear Search Analysis

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

Best Case: 4

Worst Case: $3n + 2$

Binary Search Analysis

```java
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);
    }
}
```

Best case: 4

Worst case: $\log n$? We’ll analyze this later
Solving Recurrence Relations
1. Determine the recurrence relation. What is the base case(s)?
2. "Expand" the original relation to find an equivalent general expression in terms of the number of expansions.
3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case.

Linear Search vs Binary Search

<table>
<thead>
<tr>
<th></th>
<th>Linear Search</th>
<th>Binary Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>4 at [0]</td>
<td>4 at [quad]</td>
</tr>
<tr>
<td>Worst Case</td>
<td>3n+2</td>
<td>4 log n + 4</td>
</tr>
</tbody>
</table>

So ... which algorithm is better? What tradeoffs can you make?

Fast Computer vs. Slow Computer

Fast Computer vs. Smart Programmer (round 1)

Fast Computer vs. Smart Programmer (round 2)

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 ∈ O(n)
  - Binary search is T(n) = 4 log₂n + 4 ∈ O(log n)

Remember: the fastest algorithm has the slowest growing function for its runtime
Asymptotic Analysis

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

Properties of logs

- We will assume logs to base 2 unless specified otherwise
- $\log AB = \log A + \log B$
- Proof:
  - $A = 2^{n_A}, B = 2^{n_B}$
  - $AB = 2^{n_A+n_B}$
  - $\log AB = \log(n_A+B)$
  - $\log AB = \log A + \log B$
- $\log A/B = \log A - \log B$
- $\log(A^B) = B \log A$
- Any base $k \log$ is equivalent to base 2

Order Notation: Intuition

Although not yet apparent, as $n$ gets “sufficiently large”, $f(n)$ will be “greater than or equal to” $g(n)$

Order Notation: Definition

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

$g(n) \in O(f(n))$ if there exist constants $c$ and $n_0$ such that:

$g(n) \leq c f(n)$ for all $n \geq n_0$

Example: $g(n) = 1000n$ vs. $f(n) = n^2$

Is $g(n) \in O(f(n))$ ?

Pick: $n_0 = 1000$, $c = 1$

$1000n \leq 1 \times n^2$ for all $n \geq 1000$

So $g(n) \in O(f(n))$

Notation Notes

**Note:** Sometimes, you’ll see the notation:

$g(n) = O(f(n))$.

This is equivalent to:

$g(n) \in O(f(n))$.

**However:** The notation $O(f(n)) = g(n)$ is meaningless!

(in other words big-O is not symmetric)
Order Notation: Example

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 $g(n) \to 0$ as $n \to \infty$
- $g(n) \in \Omega(f(n))$ 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 $g(n) \to \infty$ as $n \to \infty$
- $g(n) \in \Theta(f(n))$ iff $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$

Big-Omega et al. Intuitively

- $O \subseteq \Theta \subseteq \Omega$

Pros and Cons of Asymptotic Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - worst case
  - best case
  - average case
  - amortized

Kinds of Analysis
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"

Algorithm Analysis Examples

• Consider the following program segment:
  ```plaintext
  x := 0;
  for i = 1 to N do
    for j = 1 to N do
      x := x + 1;
  ```
  • What is the value of x at the end?

Analyzing the Loop

• Total number of times x is incremented is executed =
  \[1 + 2 + 3 + \cdots + \sum_{i=1}^{N} \frac{N(N+1)}{2}\]
  \[\frac{N(N+1)}{2}\]

  • 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 vs. 100n^2 + 1000

Which Function Grows Faster?

n^{0.1} vs. \log n

Which Function Grows Faster?

5n^5 vs. n!

6/21/06  Algorithm Analysis  31

6/21/06  Algorithm Analysis  32

6/21/06  Algorithm Analysis  33

6/21/06  Algorithm Analysis  34

6/21/06  Algorithm Analysis  35

6/21/06  Algorithm Analysis  36
Nested Loops

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( \text{sum} = \text{sum} + 1 \)

Nested Loops

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    if (cond) {
      \( \text{do\_stuff(sum)} \)
    } else {
      for \( k = 1 \) to \( n^2 \)
        \( \text{sum} += 1 \)

\[ 16n^3\log_8(10n^2) + 100n^2 = \mathcal{O}(n^3\log n) \]

- Eliminate low order terms
- Eliminate constant coefficients

\[
16n^3\log_8(10n^2) + 100n^2
= 16n^3\log_8(10n^2)
= n^3\log_8(10n^2)
= n^3\log_8(10) + n^3\log_8(n^2)
= n^3\log_8(10) + n^3\log_8(n) + n^3\log_8(n)
= 2n^3\log_8(n)
= n^3\log_8(2)\log(n)
= n^3\log(n)
\]