Announcements

- Homework requires you get the textbook (either E2 or E3)
- Go to Thursdays sections
- Homework #1 out on today (Wednesday)
  - Due at the beginning of class next Wednesday (Jan 17).

Algorithm Analysis

- Correctness:
  - Does the algorithm do what is intended.

- Performance:
  - Speed: time complexity
  - Memory: space complexity

- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Correctness

Correctness of an algorithm is established by proof. Common approaches:

- (Dis)proof by counterexample
- Proof by contradiction
- Proof by induction
  - Especially useful in recursive algorithms

Proof by Induction

- **Base Case:** The algorithm is correct for a base case or two by inspection.

- **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases.

- **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly.

Recursive algorithm for sum

```java
sum(int array v, int n) returns int
if n = 0 then
  sum = 0
else
  sum = nth number + sum of first n-1 numbers
return sum
```
Program Correctness by Induction

• Base Case:

• Inductive Hypothesis (n=k):

• Inductive Step (n=k+1):

How to measure performance?

We will focus on analyzing time complexity. First, we have some “rules” to help measure how long it takes to do things:

- **Basic operations**
  - Constant time

- **Consecutive statements**
  - Sum of times

- **Conditionals**
  - Test, plus larger branch cost

- **Loops**
  - Sum of iterations

- **Function calls**
  - Cost of function body

- **Recursive functions**
  - Solve recurrence relation...

Second, we will be interested in **best** and **worst** case performance.

Analyzing Performance

We will focus on analyzing time complexity. First, we have some “rules” to help measure how long it takes to do things:

- **Problem size N**
  - **Worst-case complexity**: \( \text{max} \) # steps algorithm takes on “most challenging” input of size \( N \)

- **Best-case complexity**: \( \text{min} \) # steps algorithm takes on “easiest” input of size \( N \)

Exercise - Searching

```c++
bool ArrayContains(int array[], int n, int key){
  // Insert your algorithm here
}
```

What algorithm would you choose to implement this code snippet?

Linear Search Analysis

```c++
bool LinearArrayContains(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:
Binary Search Analysis

```c
bool BinArrayContains( 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:  
Worst case:  

Solving Recurrence Relations

1. Determine the recurrence relation and base case(s).
2. “Expand” the original relation to find an equivalent expression in terms of the number of expansions (k).
3. Find a closed-form expression by setting $k$ 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</td>
<td>5 at [middle]</td>
</tr>
<tr>
<td>Worst Case</td>
<td>3n+3</td>
<td>7$\lceil \log n \rceil$ + 9</td>
</tr>
</tbody>
</table>

Linear search—empirical analysis

Each search produces a dot in above graph. 
Blue = less frequently occurring, Red = more frequent

Empirical comparison

Each search produces a dot in above graph. 
Blue = less frequently occurring, Red = more frequent
Asymptotic Analysis

- Consider only the order of the running time
  - A valuable tool when the input gets “large”
  - Ignores the effects of different machines or different implementations of same algorithm

Asymptotic Analysis

To find the asymptotic runtime, throw away the constants and low-order terms

- Linear search is \( T_{\text{linear}}(n) = 3n + 3 \in O(n) \)
- Binary search is \( T_{\text{binary}}(n) = 7 \log_2 n + 9 \in O(\log n) \)

Remember: the “fastest” algorithm has the slowest growing function for its runtime

Asymptotic Analysis

Eliminate low order terms
- \( 4n + 5 \Rightarrow 0.5 n \log n + 2n + 7 \Rightarrow n^3 + 32n + 8n \Rightarrow \)

Eliminate coefficients
- \( 4n \Rightarrow 0.5 n \log n \Rightarrow 32n \Rightarrow \)
Properties of Logs

Basic:
- \( A^{\log_b A} = B \)
- \( \log_b A = \)

Independent of base:
- \( \log(AB) = \)
- \( \log(A/B) = \)
- \( \log(A^B) = \)
- \( \log((A^B)^C) = \)

Changing base - multiply by constant
- For example: \( \log_2 x = 3.22 \log_{10} x \)
- More generally
  \[ \log_b n = \frac{1}{\log_a n} \log_a n \]
- Means we can ignore the base for asymptotic analysis (since we’re ignoring constant multipliers)

Another example
- Eliminate low-order terms
  \[ 16n^3 \log_8(10n^2) + 100n^2 \]
- Eliminate constant coefficients

Comparing functions
- \( f(n) \) is an upper bound for \( h(n) \)
  if \( h(n) \leq f(n) \) for all \( n \)

Still too strict if we want to ignore scale factors

Definition of Order Notation
- \( h(n) \in O(f(n)) \) (Big-O “Order”)
  if there exist positive constants \( c \) and \( n_0 \)
  such that \( h(n) \leq c f(n) \) for all \( n \geq n_0 \)

\( O(f(n)) \) defines a class (set) of functions

Order Notation: Intuition

\[ a(n) = n^3 + 2n^2 \]
\[ b(n) = 100n^2 + 1000 \]

Although not yet apparent, as \( n \) gets “sufficiently large”, \( a(n) \) will be “greater than or equal to” \( b(n) \)
Order Notation: Example

Example

$h(n) \in O(f(n))$ iff there exist positive constants $c$ and $n_0$ such that:

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

Example:

$100n^2 + 1000 \leq \frac{1}{2} (n^3 + 2n^2)$ for all $n \geq 100$

So $100n^2 + 1000 \in O(n^3 + 2n^2)$

Constants are not unique

$h(n) \in O(f(n))$ iff there exist positive constants $c$ and $n_0$ such that:

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

Example:

$100n^2 + 1000 \leq 1 (n^3 + 2n^2)$ for all $n \geq 100$

$100n^2 + 1000 \leq \frac{1}{2} (n^3 + 2n^2)$ for all $n \geq 198$

Another Example: Binary Search

$h(n) \in O(f(n))$ iff there exist positive constants $c$ and $n_0$ such that:

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

Is $7\log_2n + 9 \in O(\log_2n)$?

Some Notes on Notation

Sometimes you’ll see (e.g., in Weiss)

$h(n) = O(f(n))$

or

$h(n)$ is $O(f(n))$

These are equivalent to

$h(n) \in O(f(n))$
Big-O: Common Names

- constant: \( O(1) \)
- logarithmic: \( O(\log n) \) (log, \(n\), \(n^2\) \(\in\) \(O(\log n)\))
- linear: \( O(n) \)
- log-linear: \( O(n \log n) \)
- 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)

Asymptotic Lower Bounds

\( \Omega(\ g(n)\ ) \) is the set of all functions asymptotically greater than or equal to \( g(n) \)

\( h(n) \in \Omega(\ g(n)\ ) \) iff
There exist \( c>0 \) and \( n_0>0 \) such that \( h(n) \geq c \ g(n) \) for all \( n \geq n_0 \)

Asymptotic Tight Bound

\( \theta(\ f(n)\ ) \) is the set of all functions asymptotically equal to \( f(n) \)

\( h(n) \in \theta(\ f(n)\ ) \) iff
\( h(n) = O(f(n)) \) and \( h(n) = \Omega(f(n)) \)
- This is equivalent to:
  \[
  \lim_{n \to \infty} h(n)/f(n) = c = 0
  \]

Full Set of Asymptotic Bounds

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

\( \Omega(\ g(n)\ ) \) is the set of all functions asymptically greater than or equal to \( g(n) \)
- \( \omega(\ g(n)\ ) \) is the set of all functions asymptically strictly greater than \( g(n) \)

\( \theta(\ f(n)\ ) \) is the set of all functions asymptotically equal to \( f(n) \)

Formal Definitions

\( h(n) \in O(\ f(n)\ ) \) iff
There exist \( c>0 \) and \( n_0>0 \) such that \( h(n) \leq c f(n) \) for all \( n \geq n_0 \)

\( h(n) \in o(\ f(n)\ ) \) iff
There exists an \( n_0>0 \) such that \( h(n) < c f(n) \) for all \( c>0 \) and \( n \geq n_0 \)
- This is equivalent to:
  \[
  \lim_{n \to \infty} h(n)/f(n) = 0
  \]

\( h(n) \in \Omega(\ g(n)\ ) \) iff
There exist \( c>0 \) and \( n_0>0 \) such that \( h(n) \geq c g(n) \) for all \( n \geq n_0 \)

\( h(n) \in \omega(\ g(n)\ ) \) iff
There exists an \( n_0>0 \) such that \( h(n) > c g(n) \) for all \( c>0 \) and \( n \geq n_0 \)
- This is equivalent to:
  \[
  \lim_{n \to \infty} h(n)/g(n) = \infty
  \]

\( h(n) \in O(\ f(n)\ ) \) iff
\( h(n) \in \Omega(\ f(n)\ ) \) and \( h(n) \in \Omega(\ f(n)\ ) \)
- This is equivalent to:
  \[
  \lim_{n \to \infty} h(n)/f(n) = c = 0
  \]

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>
Complexity cases (revisited)

Problem size \( N \)
- **Worst-case complexity**: \( \max \) # steps algorithm takes on "most challenging" input of size \( N \)
- **Best-case complexity**: \( \min \) # steps algorithm takes on "easiest" input of size \( N \)
- **Average-case complexity**: \( \text{avg} \) # steps algorithm takes on \text{random} inputs of size \( N \)
- **Amortized complexity**: \( \max \) total # steps algorithm takes on \( M \) "most challenging" consecutive inputs of size \( N \), divided by \( M \) (i.e., divide the max total by \( M \)).

---

Bounds vs. Cases

Two orthogonal axes:

- **Bound Flavor**
  - Upper bound (\( O \), \( o \))
  - Lower bound (\( \Omega \), \( \omega \))
  - Asymptotically tight (\( \Theta \))

- **Analysis Case**
  - Worst Case (Adversary), \( T_{\text{worst}}(n) \)
  - Average Case, \( T_{\text{avg}}(n) \)
  - Best Case, \( T_{\text{best}}(n) \)
  - Amortized, \( T_{\text{amort}}(n) \)

One can estimate the bounds for any given case.

---

Big-Oh Caveats

- Asymptotic complexity (Big-Oh) considers only **large** \( n \)
  - You can "abuse" it to be misled about trade-offs
  - Example: \( n^{1/10} \) vs. \( \log n \)
    - Asymptotically \( n^{1/10} \) grows more quickly
    - But the "cross-over" point is around \( 5 \times 10^{17} \)
    - So \( n^{1/10} \) better for almost any real problem

- Comparing \( O() \) for **small** \( n \) values can be misleading
  - Quicksort: \( O(n\log n) \)
  - Insertion Sort: \( O(n^2) \)
  - Yet in reality Insertion Sort is faster for small \( n \)
  - We'll learn about these sorts later