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 \( \text{time complexity} \)
  – Memory \( \text{space complexity} \)

• 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

$Fib(3) \neq 2$
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*

- Write a *recursive* function to find the sum of the first $n$ integers stored in array $v$.

```c
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: \( n=0 \): \( \sum (v_i, 0) = 0 \)

• Inductive Hypothesis (n=k):
  \[
  \sum (v_i, k) = \sum_{i=0}^{k} v_i
  \]

• Inductive Step (n=k+1):
  \[
  \sum (v_i, k+1) = v_{k+1} + \sum (v_i, k)
  \]
How to measure 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:

- **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.
Complexity cases

We’ll start by focusing on two cases.

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

```
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

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;
}
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 );
    }
}
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><strong>Best Case</strong></td>
<td>4</td>
<td>5 at [middle]</td>
</tr>
<tr>
<td><strong>Worst Case</strong></td>
<td>$3n+3$</td>
<td>$7 \lfloor \log n \rfloor + 9$</td>
</tr>
</tbody>
</table>
Linear search—empirical analysis

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

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

<table>
<thead>
<tr>
<th>N (= array size)</th>
<th>time (# ops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td></td>
</tr>
</tbody>
</table>

Gives additional information
Fast Computer vs. Slow Computer

The diagram illustrates the comparison between the time taken for linear search on a Pentium-IV and a 486 processor as a function of the number of elements to be searched. The graph shows a linear relationship with time in milliseconds on the y-axis and the number of elements to be searched on the x-axis. The Pentium-IV performs significantly faster than the 486 processor.
Fast Computer vs. Smart Programmer
(small data)

The graph compares the time in milliseconds for linear search on a Pentium-IV processor and binary search on a 486 processor, as a function of the number of elements to be searched.
Fast Computer vs. Smart Programmer (big data)
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 \( T^{LS}_{worst}(n) = 3n + 3 \in O(n) \)

  – Binary search is \( T^{BS}_{worst}(n) = 7 \lceil \log_2 n \rceil + 9 \in O(\log n) \)

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

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

Eliminate coefficients
- $4n \Rightarrow$
- $0.5n \log n \Rightarrow$
- $3 \ 2^n \Rightarrow$
Properties of Logs

Basic:
• $A^{\log_A B} = B$
• $\log_A A =$

Independent of base:
• $\log(AB) =$
• $\log(A/B) =$
• $\log(A^B) =$
• $\log((A^B)^C) =$
Properties of Logs

Changing base → multiply by constant
  - For example: $\log_2 x = 3.22 \log_{10} x$

  - More generally
    
    $$\log_A n = \left( \frac{1}{\log_B A} \right) \log_B n$$

  - Means we can ignore the base for asymptotic analysis
    (since we’re ignoring constant multipliers)
Another example

- Eliminate low-order terms

- Eliminate constant coefficients

\[ 16n^3\log_8(10n^2) + 100n^2 \]
Comparing functions

• $f(n)$ is an **upper bound** for $h(n)$
  if $h(n) \leq f(n)$ for all $n$

This is too strict – we mostly care about *large* $n$

Still too strict if we want to ignore *scale factors*
Definition of Order Notation

- $h(n) \in O(f(n))$ \textbf{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

Although not yet apparent, as \( n \) gets "sufficiently large", \( a(n) \) will be "greater than or equal to" \( b(n) \).

\[
a(n) = n^3 + 2n^2 \\
b(n) = 100n^2 + 1000
\]
Order Notation: Example

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

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

\[ h(n) \in O(f(n)) \quad \text{iff there exist positive constants } c \text{ and } n_0 \text{ such that:} \]
\[ h(n) \leq c f(n) \text{ for all } n \geq n_0 \]

Example:

\[ 100n^2 + 1000 \leq 1 (n^3 + 2n^2) \text{ for all } n \geq 100 \]

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:
\[ h(n) \leq c f(n) \text{ for all } n \geq n_0 \]

Example:

\[ 100n^2 + 1000 \leq 1 (n^3 + 2n^2) \text{ for all } n \geq 100 \]

\[ 100n^2 + 1000 \leq 1/2 (n^3 + 2n^2) \text{ for all } n \geq 198 \]
Another Example: Binary Search

\[ h(n) \in O\left( f(n) \right) \quad \text{iff there exist positive constants } c \text{ and } n_0 \text{ such that:} \]
\[ h(n) \leq c f(n) \text{ for all } n \geq n_0 \]

Is \( 7\log_2 n + 9 \in O\left( \log_2 n \right) \)?
Order Notation:
Worst Case Binary Search
Some Notes on Notation

Sometimes you’ll see (e.g., in Weiss)

\[ h(n) = O(f(n)) \]

or

\[ h(n) \text{ is } O(f(n)) \]

These are equivalent to

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

- constant: \( O(1) \)
- logarithmic: \( O(\log n) \) \((\log_k n, \log n^2 \in O(\log n))\)
- linear: \( O(n) \)
- log-linear: \( O(n \log n) \)
- quadratic: \( O(n^2) \)
- cubic: \( O(n^3) \)
- polynomial: \( O(n^k) \) \((k \text{ is a constant})\)
- exponential: \( O(c^n) \) \((c \text{ is a constant } > 1)\)
Asymptotic Lower Bounds

• $\Omega( g(n) )$ is the set of all functions asymptotically \textbf{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
  \( h(n) \in O(f(n)) \) and \( h(n) \in \Omega(f(n)) \)
  - This is equivalent to:
    \[
    \lim_{n \to \infty} \frac{h(n)}{f(n)} = c \neq 0
    \]
Full Set of Asymptotic Bounds

- $\mathcal{O}(f(n))$ 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 $f(n)$

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

- $\Theta(f(n))$ is the set of all functions asymptotically equal to $f(n)$
Formal Definitions

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

- \( h(n) \in \omega(f(n)) \) 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: \( \lim_{n \to \infty} h(n)/f(n) = \infty \)

- \( h(n) \in \Theta( g(n) ) \) 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: \( \lim_{n \to \infty} h(n)/g(n) = \infty \)

- \( h(n) \in \Omega( f(n) ) \) iff
  
  \( h(n) \in \mathcal{O}( f(n) ) \) and \( h(n) \in \Omega(f(n) ) \)
  
  – This is equivalent to: \( \lim_{n \to \infty} h(n)/f(n) = c \neq 0 \)
### 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**: $\text{max}$ # steps algorithm takes on “most challenging” input of size $N$

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

- **Average-case complexity**: $\text{avg}$ # steps algorithm takes on random inputs of size $N$

- **Amortized complexity**: $\text{max}$ 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 (Ω, ω)
  - Asymptotically tight (θ)

- **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.
Bounds vs. Cases
Pros and Cons of Asymptotic Analysis
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