CSCI 261
Computer Science II
My Mom
My Mom Is a Human Yardstick

- All my mom ever does is compare things

- Small sample of things she's said over the past year:
  - "This fish cake isn't as good as the fish cake I had in Taiwan."
  - "Your salt is less salty than my salt."
  - "You do fewer chores than I did your age."
  - "My Obamacare is better than your health insurance."
  - "Your Asian grocery isn't as clean as mine."
Computer scientists? We love comparing things

- What good is a science if we can't compare solutions/methods?

The *analysis of algorithms* concerns predicting the *resources* an algorithm requires to finish running.

- Time complexity (or running time) is the *primary* concern for computer scientists
- Other concerns:
  - Memory requirements (space complexity), network bandwidth, cost, ...
Outline

- Modeling the Running Time of Algorithms
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
First, the "running time" of an algorithm is a misnomer!

- We're not interested in knowing the actual time (in days, seconds, milliseconds, etc.) it took an algorithm to terminate.

- (Why?)
Why *not* just measure actual time taken?

- **Inconsistency**: Actual times are machine-dependent and can differ from run-to-run.
  - Same algorithm could take 2 sec on my laptop and 0.002 ms on a supercomputer
  - Subsequent run can be much *faster* than the initial run (via caching)
- **No durability**: Actual times will change as hardware changes.
  - "*My algorithm is fast because it took 14.38ms on a Dell Optiplex 7300 with 8 GB of memory, an Intel i7 3.2Ghz CPU, ...*"
  - (Someone reads your research paper 30 years later... results don't translate)
Want to characterize algorithms' running times in terms of:

• Problem (or input) size:
  - For searching, sorting, etc., it is the number of elements in the collection
  - To determine if a number \( p \) is prime, problem size is \( p \)
  - Could have multiple variables, too!
    – To traverse a graph, it is the number of edges and nodes

• Growth rate: how much longer does it take as the input size grows?

Idea: Given two algorithms, we can compare their growth rates with respect to the input size.
Formalizing Time Complexity

- For this class, we'll adopt the following model:
  - Define *time complexity (or running time)* as the number of *simple statements* executed.
    - A *simple statement* is a line of code: `System.out.println(..), i++`
      - Assume each simple statement requires a constant unit of time to execute
      - Assume some statements are "free": `for, if, while, and return`

- For each algorithm, we want to come up with a function $T(n)$ that estimates its time complexity, where $n$ is the problem size.
Example: Linear Search

- The *Linear Search* algorithm
  - Search for key in \( x[. . .] \). Returns its index if found, otherwise -1.
  - Assume there are \( n \) elements in array \( x[. . .] \). That is, \( n = x.length \)

```java
public static int linear_search(int[] x, int key) {
    for (int i = 0; i < x.length; i++) {
        if (x[i] == key) {
            return i;
        }
    }
    // key not found
    return -1;
}
```

- How many times does the loop-body execute? \([1, n, \frac{n}{2}]\)
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of \( x[\ldots] \) is \( n \), size of \( y[\ldots] \) is \( m \)
  - *Could we reuse linear search?*

\[
\begin{align*}
\text{x:} & \quad 5 & 4 & 6 & 7 & 9 & 1 & 2 & 0 & -5 \\
& \quad 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots \text{ length: } n \\
\text{y:} & \quad 10 & 3 & 0 & -7 & 99 & 10 & 12 & 13 \\
& \quad 0 & 1 & 2 & 3 & 4 & 5 & 6 & \ldots \text{ length: } m
\end{align*}
\]
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of \( x[\ldots] \) is \( n \), size of \( y[\ldots] \) is \( m \)
  - *Could we reuse linear search?*

```
5 4 6 7 9 1 2 0 -5
0 1 2 3 4 5 6 7 ...
```

```
10 3 0 -7 99 10 12 13
0 1 2 3 4 5 6 ...
```

\( \text{linear_search}(y, x[0]) \)
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of $x[\ldots]$ is $n$, size of $y[\ldots]$ is $m$
  - *Could we reuse linear search?*
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of $x[\ldots]$ is $n$, size of $y[\ldots]$ is $m$
  - *Could we reuse linear search?*
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of \( x[\ldots] \) is \( n \), size of \( y[\ldots] \) is \( m \)
  - *Could we reuse linear search?*
Determining whether two arrays share no common element

- Assume size of $x[\ldots]$ is $n$, size of $y[\ldots]$ is $m$

```java
/** Determine whether two arrays have no common elements. *
 * @param x One array
 * @param y The other array
 * @return true if there are no common elements *
 */
public static boolean noCommonItems(int[] x, int[] y) {
    // For each val in x[\ldots], search for it in array y[\ldots]
    for (int i = 0; i < x.length; i++) {
        if (linear_search(y, x[i]) > -1) {
            // Found an element in common
            return false;
        }
    }
    // No elements found in common
    return true;
}
```

How many times does the loop body run in the **worst** case? $mn$
Example: No Duplicates

- Determining whether the array contain a duplicated item
  - Assume size of $x[\ldots]$ is $n$
Example: No Duplicates

- Determining whether the array contain a duplicated item
  - Assume size of $x[\ldots]$ is $n$
Example: No Duplicates

- Determining whether the array contain a duplicated item
  - Assume size of $x[\ldots]$ is $n$
Example: No Duplicates

- Determining whether the array contain a duplicated item
  - Assume size of \( x \) is \( n \)

```
5 4 6 7 9 1 2 0 8
0 1 2 3 4 5 6 7 ...
```

- Assume there's no duplicates (worst case)
- How many total times do we do this?
Determining whether the array contain a duplicated item

- Assume size of $x[..]$ is $n$

```java
/**
 * Determine whether the contents of an array are all unique.
 * @param x The array
 * @return true if all elements of x are unique
 */
public static boolean noDuplicates(int[] x) {
    // For each val in x[], look for matches elsewhere in array x
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x.length; j++) {
            if (i != j && x[i] == x[j]) {
                return false; // Found a match
            }
        }
    }
    // No matches found
    return true;
}
```

- How many times does the loop body run in the worst case? $n^2$
No Duplicates (2.0)

- Determining whether the array contain a duplicated item
  - Assume size of \( x[\ldots] \) is \( n \)
No Duplicates (2.0)

- Determining whether the array contain a duplicated item
  - Assume size of $x[\ldots]$ is $n$
No Duplicates (2.0)

- Determining whether the array contain a duplicated item
  - Assume size of \( x[\ldots] \) is \( n \)
  
  Assume there's no duplicates (worst case)
  How many total times do we do this?
Solves the same problem, but faster!

- Examine inner loop: haven't the first \( i \) elements been determined unique? Skip them!!!

```java
public static boolean noDuplicates(int[] x) {
    // For each val in x[], look for matches elsewhere in array x
    for (int i = 0; i < x.length; i++) {
        for (int j = i+1; j < x.length; j++) {
            if (x[i] == x[j]) {
                return false; // Found a match
            }
        }
    }
    // No matches found
    return true;
}
```

How many times does the loop body run? \( \frac{n(n - 1)}{2} \) or \( \frac{1}{2} n^2 - \frac{1}{2} n \)
Note

- NoDuplicates 2.0 is clearly faster than NoDuplicates 1.0

- We can make the claim via number of simple steps run

- We *can't* say how long either would take in seconds, though.
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
Silly Example

How many simple statements are executed?

```java
public static void silly(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            sum += i + j; // Contributes $n^2$ statements
        }
    }
    for (int k = 0; k < n; k++) {
        System.out.println("I like the number 1");
        System.out.println("I like the number 2");
        System.out.println("I like the number 3");
        System.out.println("I like the number 4");
        System.out.println("I like the number 5");
        // Contributes 5n statements
    }
    // (22 statements omitted...)
    System.out.println("Finally, I like the number 30");
    // Contributes 25 more statements
}
```

\[ T(n) = n^2 + 5n + 25 \]
Let's take a closer look at `silly()`'s time complexity: \( T(n) = n^2 + 5n + 25 \)

Important to note

- The value of \( T(n) \) gets dominated by the \( n^2 \) term as \( n \) becomes large!

### Complexity Breakdown of each term in \( T(n) \)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( T(n) = n^2 + 5n + 25 )</th>
<th>( n^2 )</th>
<th>( 5n )</th>
<th>( 25 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31</td>
<td>1 (3%)</td>
<td>5 (16%)</td>
<td>25 (81%)</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>4 (10%)</td>
<td>10 (26%)</td>
<td>25 (64%)</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>9 (18%)</td>
<td>15 (30%)</td>
<td>25 (51%)</td>
</tr>
<tr>
<td>4</td>
<td>61</td>
<td>16 (26%)</td>
<td>20 (32%)</td>
<td>25 (41%)</td>
</tr>
<tr>
<td>5</td>
<td>75</td>
<td>25 (33%)</td>
<td>25 (33%)</td>
<td>25 (33%)</td>
</tr>
<tr>
<td>10</td>
<td>175</td>
<td>100 (57%)</td>
<td>50 (29%)</td>
<td>25 (14%)</td>
</tr>
<tr>
<td>100</td>
<td>10525</td>
<td>10000 (95%)</td>
<td>500 (5%)</td>
<td>25 (0.2%)</td>
</tr>
<tr>
<td>1000</td>
<td>1005025</td>
<td>1000000 (99.5%)</td>
<td>5000 (0.5%)</td>
<td>25 (0.002%)</td>
</tr>
</tbody>
</table>
For ease of comparing algorithms, we want to simplify and say things like, "The running time of this algorithm is order $n^2$"

**Big-O Notation:** $T(n) = O(f(n))$ if, and only if, there exists positive constants $c$ and $n_0$ such that $cf(n) \geq T(n)$ for all $n \geq n_0$
Back to silly()

- Can we show that $T(n) = n^2 + 5n + 25$ is $O(n^2)$?

- Need to find $c$ and $n_0$ such that, for all $n > n_0$: $cn^2 \geq n^2 + 5n + 25$
  
  - Not necessarily a unique pair of $c$ and $n_0$
  - Use this: https://www.desmos.com/calculator
    
    - Let $c = 1$, solve for $n_0$... doesn't exist
    - Let $c = 2$, solve for $n_0$... 8.09
    - Let $c = 3$, solve for $n_0$... 5.0
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
In Class Exercises

- Give the time complexities in Big-O notation for the following algorithms
  - Step 1: Write a function $T(n)$ that expresses the number of simple statements
    - Usually, we just care about worst and average cases
  - Step 2: Identify the dominant term of $T(n)$ and use that for Big-O
  - Step 3: Verify correctness of step 2 by finding $c$ and $n_0$
    - Graph it, if necessary

- Example 1 (do together)!

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.println(i + "", " + j);
    }
}
```
In Class Exercises (Cont.)

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 2; j++) {
        System.out.println(i+"", "+j);
    }
}
```

$O(n)$

```java
for (int i = 0; i < n; i++) {
    for (int j = n-1; j >= i; j--) {
        System.out.println(i+"", "+j);
    }
}
```

$O(n^2)$

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        if (j % i == 0) {
            System.out.println(i+"", "+j);
        }
    }
}
```

$O(n^2)$
When Problem Size Doesn't Matter

- What about these?

```java
public static void sillier(int n) {
    for (int i = 0; i < 1000000; i++) {
        System.out.println(n);
    }
}
```

```java
public static boolean is_odd(int x) {
    int remainder = x % 2;
    return (remainder == 1);
}
```

- Note: the number of simple statements does not depend on input size
  - When this is the case, the algorithm is has "constant time"
  - Denoted with $O(1)$
Last one: Binary Search Algorithm

- Assume list is sorted in ascending order

```
public int binarySearch(int[] list, int key) {
    int left = 0, right = list.length - 1;
    
    while (left <= right) {
        int mid = (left + right) / 2; // find midpoint
        if (key == list[mid]) {
            return mid; // found the key! return the index and stop!
        } else if (key > list[mid]) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    // didn't find the key! return -1
    return -1;
}
```
Let's try to analyze the best and worst case scenarios for binary search:

- **Worst case is observed when...?**
  - Estimated number of simple statements in the worst case?
- **Intuition: You have $n$ elements in the list**
  - The list size shrinks by half after each comparison
  - *Visualize the worst case on the board*

**Binary search is:** $O(\log_2 n)$
Let's compare efficiency to Linear Search

<table>
<thead>
<tr>
<th>List size: N</th>
<th>Linear Search: N</th>
<th>Binary Search: Log₂(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
Potential Pitfall

Consider the following code. We can reasonably claim \( T(n) = 1 + 2n \)

```java
public static void multi_print(int n) {
    int i = 0;
    while (i < n) {
        System.out.println(n);
        i++;
    }
}
```

Okay, it's clearly \( O(n) \), but....

Would it be wrong to say `multi_print(..)` is \( O(n^3) \)? \( O(n^2) \)?

- By definition of Big-O, no! They both provide upper bound to \( T(n) = 1 + 2n \)
- But \( O(n) \) is more precise!
Tightness of Bounds

- There is an expectation that we're only interested in a tight bound.
  - After all, we don't want to undersell an algorithm's efficiency
  - Example:
    - It is actually okay for us to say:
      - Binary search is $O(n^2)$
        » It is well-known to be tightly bound by $O(\log_2 n)$
      - Linear search is $O(n)$
        » This is really the tightest bound on linear search
    - But, it is unsound to claim that linear search has a lower complexity than binary search! Need to compare tight bounds.
Common Complexity Classes

- We can also say that an algorithm runs in "linear" or "quadratic"-time
Outline

- Modeling Running Time of Algorithm
- Formalizing Complexity: The Big-O Notation
- In-Class Exercises
- A Potential Pitfall
- Conclusion
Conclusion

- The Big-O notation is used to approximate and classify algorithms' use of resources.
  - These resources can be time, space, network usage, etc.
    - Specifically, "time complexity" is measured as number of simple statements
  - For instance, the "space complexity" of silly() is \( O(1) \)
    - It only introduces a constant number of variables

- Why Big-O?
  - Want a simple and intuitive way to compare algorithms
  - Assume alg1 and alg2 solve the same problem:
    - We can claim alg2 is better than alg1 because alg1 runs in \( O(n) \) and alg2 runs in \( O(\log n) \)
Conclusion (cont.)

- We generally only care about worst-case and average-case analyses

- In practice:
  - Polynomial class (and below) algorithms are desirable.
    - Constant-time algorithms are ideal, but few exist that are very useful
  - Exponential (and higher) class algorithms are usually unacceptable!
    - Okay, when might they be acceptable?

- Be able to: Give the tight upper-bound (in Big-O) of any given algorithm
  - Except recursive ones, for now
Collections and Complexity

- Take a look at the following code, which adds all elements in the array

```java
int sum = 0;
for(int i = 0; i < n; i++) {
    sum = sum + a[i];
}
```

- Is the *simple statement in the loop* constant-time?
  - Is + an $O(1)$ operation?
  - Are array references an $O(1)$ operation?
    - What if we assume elements at larger indices take longer to reference?
    - What if we assume all elements have constant-time reference?

- But arrays aren't the only kind of lists... *(next lecture)*
Administrivia 2/12

- Hwk 3 up soon
- Hwk 1 graded (look for grades on moodle today)
- WACM meeting tonight
  - 5p, TH 409
  - Topic: Grace Hopper Travel Scholarship
Administrivia 2/14

- Hwk 1 graded. Avg = 73/90

- New stuff on course page
  - Solution to Lab 4 (Controller)
  - Lecture 3: Complexity
  - Hwk 3 posted (due Monday 2/26)

- Lab 5 Postmortem (due tonight)
  - Exceptional execution
  - Catching and dead code
  - Checked vs. Unchecked exceptions