Analysis of Algorithms

- We want yardsticks to compare algorithms

- The *analysis of algorithms* estimates the *resources* that an algorithm requires to run.
  - Running time is the *primary* yardstick we consider
  - Other concerns:
    - Space requirements
    - Network bandwidth
    - Hardware cost
Outline

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

- First, the "running time" is a misnomer!
  - We're not actually interested in knowing the wall-clock time (in days, seconds, milliseconds, etc.) it took an algorithm to finish.
  - (Why?)

To understand, why subsequent runs are generally faster than the initial run we need to know something about the memory hierarchy.
Why not just measure wall-clock time?

- **Inconsistency**: Wall-clock times are machine-dependent and can differ from run-to-run.
  - Same algorithm could take 2 sec on my laptop and 0.0002 ms on a supercomputer
  - Also, subsequent runs may be faster than the first run

- **No durability**: Running times change over time.
  - "My algorithm is great because it took 14.38ms on state-of-the-art 2019 machine ..."
What Is Consistent and Durable?

- What are the lasting properties of algorithms that can approximate wall-clock time?
  - Machine independent
  - Durable

Number of statements!

```java
public static void print_up_to() {
    for (int i = 0; i < 100000; i++) {
        System.out.println(i);
    }
}

"100000 statements run in 1950"

(Better than "Took 30 seconds to finish" on IBM 360)
```

```java
public static void print_up_to() {
    for (int i = 0; i < 100000; i++) {
        System.out.println(i);
    }
}

"100000 statements run in 2019"

(Better than "Took 3 milliseconds to finish" on my Dell)
```
Modeling Time Complexity

- **We want** to characterize algorithms' running times in terms of:
  - **Problem (or input) size**
    - Depends on the problem being studied.
    - For searching, sorting, it is the number of items, $n$, in a list
    - Sometimes, it's best described using two numbers (e.g., dimensions of 2D array)
  - **Rate of growth**: how many more statements does it take as the problem size grows?

- **Yardstick**: Given 2 algorithms that solve the same problem, we compare their *rates of growth* with respect to the input size.
We'll adopt this simplifying assumption:

- A *simple statement* is a primitive operation
  
  - *For example:* `n = n/2; System.out.println(..), i++`
  
  - Assume simple statements cost a constant unit of time
  
  - Some statements are "free": `for, if, while, throw, break, and return`

For each algorithm, we want to express a function $T(n)$ that estimates its time complexity, where $n$ is the problem size.

- Then we determine $T(n)$'s rate of growth
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;
}
```

- *What is* $T(n)$ *in the worst case?*
  
  $$[1, n, \frac{n}{2}]$$
Example: No Common Items

- Determining whether two arrays share no common element
  - Assume size of $x[...]$ is $n$, size of $y[...]$ 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?*

```
linear_search(y, x[1])
```

```
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 ...
```
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) {

    // TODO -- I'll write this one (board)
}
```

**How many times does loop body run** in the worst case?

- Well I have two inputs: \( T(m, n) = mn \)
Example: No Duplicates

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

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

- Determining whether an array contain a duplicated item
  - Assume size of \( x[... \] \) is \( n \)
Example: No Duplicates

- Determining whether an array contain a duplicated item
  - Assume size of $x[\ldots]$ is $n$

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

- Assume there's no duplicates (worst case)
- How many total times do we do this?
Example: No Duplicates (Cont.)

- 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) {
  // TODO: I'll write this one again (on board)
}
```

- How many times does 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?
No Duplicates (2.0)

- Solves the same problem, but faster!
  - Examine inner loop: haven't the first $i$ elements been determined unique? Skip them!!!

```java
public static boolean enhanced_noDuplicates(int[] x) {
    // TODO -- your turn!
}
```

- What is $T(n)$ in the worst case? \( \frac{n(n - 1)}{2} \) or \( \frac{1}{2}n^2 - \frac{1}{2}n \)