Last Time

• Rather than timing an algorithm, we can evaluate its runtime through counting primitive operations.

• We can use the bounds of loops to determine which complexity class / order of growth an algorithm belongs to.

• Big-O notation allows us to express this order of growth while limiting the details – gives us a ballpark figure.
Better Hardware is not a Solution

• Let’s say that we have three algorithms, which run at different rates, in different complexity classes.

<table>
<thead>
<tr>
<th>Running Time (n = 1μs)</th>
<th>Problem Size Solved in 1s</th>
<th>Problem Size Solved in 1m</th>
<th>Problem Size Solved in 1h</th>
</tr>
</thead>
<tbody>
<tr>
<td>400n</td>
<td>2,500</td>
<td>150,000</td>
<td>9,000,000</td>
</tr>
<tr>
<td>2n²</td>
<td>707</td>
<td>5,477</td>
<td>42,426</td>
</tr>
<tr>
<td>2^n</td>
<td>20</td>
<td>26</td>
<td>32</td>
</tr>
</tbody>
</table>
Better Hardware is not a Solution

- Now, let’s say that we improve our hardware, so that our new machine is 256 times faster than our old machine.

<table>
<thead>
<tr>
<th>Running Time ( (n = \frac{1}{256}\mu s) )</th>
<th>Problem Size Solved in 1s</th>
<th>Problem Size Solved in 1m</th>
<th>Problem Size Solved in 1h</th>
</tr>
</thead>
<tbody>
<tr>
<td>400( n )</td>
<td>640,000</td>
<td>38,400,000</td>
<td>2,304,000,000</td>
</tr>
<tr>
<td>( 2n^2 )</td>
<td>11,314</td>
<td>87,636</td>
<td>678,822</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>28</td>
<td>34</td>
<td>40</td>
</tr>
</tbody>
</table>
Better Hardware is not a Solution

- If we set our old program running speed to $m$, then our speedup with 256x better hardware is:

<table>
<thead>
<tr>
<th>Algorithm Running Time</th>
<th>New Maximum Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$400n$</td>
<td>$256m$</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>$16m$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$m + 8$</td>
</tr>
</tbody>
</table>
Let’s Analyze the arrayMax Function

double arrayMax(double[] data) {
    int n = data.length;
    double currentMax = data[0];
    for (int j = 1; j < n; j++) {
        if (data[j] > currentMax) {
            currentMax = data[j];
        }
    }
    return currentMax;
}

03/03/2017 Evaluating Algorithms
Let’s Analyze the arrayMax Function

- The n initialization, currentMax initialization, and return are all constant-time operations.
- The runtime of arrayMax is dominate by the for loop, which executes \( n - 1 \) times.
  - We conclude that the running time of arrayMax is \( O(n) \), because the if is computed on every iteration.
- What about the currentMax update inside of the if?
  - If the input sequence is in random order, the probability that the \( j^{\text{th}} \) element is the largest of the first \( j \) elements is \( \frac{1}{j} \).
  - Thus, the expected number of times that we will update the biggest is \( H_n = \sum_{j=1}^{n} \left( \frac{1}{j} \right) \), which is called the \( n^{\text{th}} \) Harmonic number.
  - \( H_n \) converges to \( O(\log(n)) \), so the number of times we expect to update currentMax is \( O(\log(n)) \).
Let’s Analyze the disjoint Function

```java
boolean disjoint(int[] A, int[] B, int[] C) {
    for (int a: A) {
        for (int b: B) {
            for (int c: C) {
                if ((a==b) && (b==c)) {
                    return false;
                } //if
            } //for
        } //for
    } //for
    return true;
} //disjoint
```
Let’s Analyze the disjoint Function

• If sets A, B, and C all have size $n$, then we iterate through the a loop $n$ times, the b loop $n$ times, and the c loop $n$ times.
  – Each of these loops are nested, so our running time is $O(n^3)$.

• How can we improve this performance?
TODO: Analyze a Function

• Try to get into groups (may be difficult because everyone is at different times in the lecture).

• Take a look at the disjoint2 function on the next slide, and see if you can evaluate its Big-O order of growth (without looking at the slide 12 explanation).
Let’s Analyze the disjoint2 Function

```java
boolean disjoint2(int[] A, int[] B, int[] C) {
    for (int a: A) {
        for (int b: B) {
            if (a == b) {
                for (int c: C) {
                    if (a == c) {
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

Let’s Analyze the disjoint2 Function

• The outer two `for` loops are quadratic, $O(n^2)$.
  – Assuming that each item in A and B is unique, there are at most $n$ times when items a and b are equal.
  – Thus, the innermost `for` loop over C only runs $n$ times. Our overall running time is $O(n^2) + O(n^2)$. 
Any Questions?