Announcements

- PA1 is out, due on Sunday
  - Autolab will be up by tonight
Recap of Runtime Complexity

**Big-$\Theta$**
- Growth functions are in the **same** complexity class.
- If $f(n) \in \Theta(g(n))$ then an algorithm taking $f(n)$ steps is "exactly" as fast as one that takes $g(n)$ steps.

**Big-$O$**
- Growth functions in the **same or smaller** complexity class.
- If $f(n) \in O(g(n))$, then an algorithm that takes $f(n)$ steps is **at least as fast** as one taking $g(n)$ (but it may be even faster).

**Big-$\Omega$**
- Growth functions in the **same or bigger** complexity class.
- If $f(n) \in \Omega(g(n))$, then an algorithm that takes $f(n)$ steps is **at least as slow** as one that takes $g(n)$ steps (but it may be even slower).
Recap of Runtime Complexity

**Big-$\Theta$ — Tight Bound**
- Growth functions are in the same complexity class
- If $f(n) \in \Theta(g(n))$ then an algorithm taking $f(n)$ steps is "exactly" as fast as one that takes $g(n)$ steps.

**Big-$O$ — Upper Bound**
- Growth functions in the same or smaller complexity class.
- If $f(n) \in O(g(n))$, then an algorithm that takes $f(n)$ steps is at least as fast as one taking $g(n)$ (but it may be even faster).

**Big-$\Omega$ — Lower Bound**
- Growth functions in the same or bigger complexity class
- If $f(n) \in \Omega(g(n))$, then an algorithm that takes $f(n)$ steps is at least as slow as one that takes $g(n)$ steps (but it may be even slower)
Common Runtimes (in order of complexity)

Constant Time: $\Theta(1)$

Logarithmic Time: $\Theta(\log(n))$

Linear Time: $\Theta(n)$

Quadratic Time: $\Theta(n^2)$

Polynomial Time: $\Theta(n^k)$ for some $k > 0$

Exponential Time: $\Theta(c^n)$ (for some $c \geq 1$)
## Common Runtimes (in order of complexity)

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Runtime Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Time</td>
<td>$\Theta(1)$</td>
<td>$T(n) = c$</td>
</tr>
<tr>
<td>Logarithmic Time</td>
<td>$\Theta(\log(n))$</td>
<td>$T(n) = c \log(n)$</td>
</tr>
<tr>
<td>Linear Time</td>
<td>$\Theta(n)$</td>
<td>$T(n) = c_1n + c_0$</td>
</tr>
<tr>
<td>Quadratic Time</td>
<td>$\Theta(n^2)$</td>
<td>$T(n) = c_2n^2 + c_1n^1 + c_0$</td>
</tr>
<tr>
<td>Polynomial Time</td>
<td>$\Theta(n^k)$ for some $k &gt; 0$</td>
<td>$T(n) = c_kn^k + ... + c_1n + c_0$</td>
</tr>
<tr>
<td>Exponential Time</td>
<td>$\Theta(c^n)$ (for some $c \geq 1$)</td>
<td>$T(n) = c^n$</td>
</tr>
</tbody>
</table>
Given the following pseudocode:

```
for (i ← 0 until n) { /* do work */ }
```

If the `/* do work */` portion of the code originally takes 10 steps...

But we optimize it to now take 7 steps...

Our total runtime goes from 10n steps to 7n steps: 30% faster!

...but still $\Theta(n)$
Compare the two runtimes:

\[ T_1(n) = 100n \]
\[ T_2(n) = n^2 \]

- \( 100n \in O(n^2) \) (\( T_2 \) is the slower runtime)
- ...but consider if \( c_{\text{high}} = 1, n_0 = 100 \)
- Until our input size reaches 100 or more, \( T_2 \) is the faster runtime
Takeaways

Asymptotically slower runtimes can be better in real-world situations.

- An algorithm with runtime $T_2$ is better on small inputs
- An algorithm with runtime $T_2$ might be easier to implement/maintain
- An algorithm with runtime $T_1$ might not exist
**Takeaways**

Asymptotically slower runtimes *can* be better in real-world situations.

- An algorithm with runtime $T_2$ is better on small inputs
- An algorithm with runtime $T_2$ might be easier to implement/maintain
- An algorithm with runtime $T_1$ might not exist

*(sometimes this is provable...see CSE 331)*
The important thing is learning the tools to *reason* about the different algorithms and *why* you might choose one over the other!
Takeaways

The important thing is learning the tools to reason about the different algorithms and why you might choose one over the other!

...But for this class, we can assume that if $T_2(n)$ is in a bigger complexity class, then $T_1(n)$ is better/faster/stronger.
Now some examples...
...and common pitfalls
Bubble Sort

1 bubblesort(seq: Seq[Int]):
2   n ← seq length
3   for i ← n-2 to 0, by -1:
4     for j ← i to n-1:
5       if seq(j+1) < seq(j):
6         swap seq(j) and seq(j+1)

What is the runtime complexity class for bubblesort?
Bubble Sort

```haskell
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

What is the runtime complexity class for bubblesort?
How many steps does it take?
Bubble Sort

```haskell
1  bubblesort(seq: Seq[Int]):
2      n ← seq length
3  for i ← n-2 to 0, by -1:
4      for j ← i to n-1:
5          if seq(j+1) < seq(j):
6              swap seq(j) and seq(j+1)
```

What is the runtime complexity class for bubblesort?
How times does it execute lines 5 and 6?
**Bubble Sort**

1. `bubblesort(seq: Seq[Int]):`
2. `n ← seq length`
3. `for i ← n-2 to 0, by -1:`
4. `for j ← i to n-1:`
5. `if seq(j+1) < seq(j):`
6. `swap seq(j) and seq(j+1)`

What is the runtime complexity class for bubblesort? How many times does it execute lines 5 and 6?

This loop has 1 iteration when `i = n-2`...2 iterations when `i = n-3`...3 iterations when `i = n-4`...

...n-1 iterations when `i = 0`
Bubble Sort

1. `bubblesort(seq: Seq[Int]):`
2. `n ← seq length`
3. `for i ← n-2 to 0, by -1:
   4. for j ← i to n-1:
   5. if seq(j+1) < seq(j):
   6.   swap seq(j) and seq(j+1)

This loop has 1 iteration when i = n-2
... 2 iterations when i = n-3
... 3 iterations when i = n-4
...
... n-1 iterations when i = 0

What is the runtime complexity class for bubblesort?
How times does it execute lines 5 and 6? 1 + 2 + 3 + 4 + 5 ... + n-1
Helpful Summation Rules

1. \[ \sum_{i=j}^{k} c = (k - j + 1)c \]

2. \[ \sum_{i=j}^{k} (cf(i)) = c \sum_{i=j}^{k} f(i) \]

3. \[ \sum_{i=j}^{k} (f(i) + g(i)) = \left( \sum_{i=j}^{k} f(i) \right) + \left( \sum_{i=j}^{k} g(i) \right) \]

4. \[ \sum_{i=j}^{k} f(i) = \left( \sum_{i=\ell}^{k} f(i) \right) - \left( \sum_{i=\ell}^{j-1} f(i) \right) \text{ (for any } \ell < j) \]

5. \[ \sum_{i=j}^{k} f(i) = f(j) + f(j + 1) + \ldots + f(k - 1) + f(k) \]

6. \[ \sum_{i=j}^{k} f(i) = f(j) + \ldots + f(\ell - 1) + \left( \sum_{i=\ell}^{k} f(i) \right) \text{ (for any } j < \ell \leq k) \]

7. \[ \sum_{i=j}^{k} f(i) = \left( \sum_{i=j}^{\ell} f(i) \right) + f(\ell + 1) + \ldots + f(k) \text{ (for any } j \leq \ell < k) \]

8. \[ \sum_{i=1}^{k} i = \frac{k(k+1)}{2} \]

9. \[ \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \]

10. \( n! \leq c_n n^n \) is a tight upper bound (Sterling: Some constant \( c_n \) exists)
Bubble Sort

What is the runtime complexity class for bubblesort?
How times does it execute lines 5 and 6?

\[
1 + 2 + 3 + 4 + 5 \ldots + n-1 = (n) \times (n - 1) / 2 = O(n^2)
\]
Bubble Sort

```haskell
1  bubblesort(seq: Seq[Int]):
2    n ← seq length
3    for i ← n-2 to 0, by -1:
4      for j ← i to n-1:
5        if seq(j+1) < seq(j):
6          swap seq(j) and seq(j+1)
```

Note: We can ignore the exact number of steps required by a portion of the algorithm, as long as we know its complexity...
```haskell
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

Note: We can ignore the exact number of steps required by a portion of the algorithm, as long as we know its complexity...
Bubble Sort

1 \texttt{bubblesort(seq: Seq[Int]):}
2 \hspace{1em} n \leftarrow \text{seq length}
3 \hspace{1em} \textbf{for} i \leftarrow n-2 \text{ to } 0, \text{ by } -1:
4 \hspace{2em} \textbf{for} j \leftarrow i \text{ to } n-1:
5 \hspace{4em} \textbf{if} \texttt{seq(j+1)} < \texttt{seq(j)}:
6 \hspace{6em} \text{swap} \texttt{seq(j)} \text{ and} \texttt{seq(j+1)}

\textbf{Note:} We can ignore the exact number of steps required by a portion of the algorithm, as long as we know its complexity...

\textbf{Can we safely say this algorithm is} $\Theta(n^2)$?
Bubble Sort

1 bubblesort(seq: Seq[Int]):
2   n ← seq length
3   for i ← n-2 to 0, by -1:
4     for j ← i to n-1:
5       if seq(j+1) < seq(j):
6         swap seq(j) and seq(j+1)

Note: We can ignore the exact number of steps required by a portion of the algorithm, as long as we know its complexity...

Can we safely say this algorithm is \( \Theta(n^2) \)?
Bubble Sort

1 bubblesort(seq: Seq[Int]):
2   n ← seq length
3   for i ← n-2 to 0, by -1:
4       for j ← i to n-1:
5           if seq(j+1) < seq(j):
6               swap seq(j) and seq(j+1)

Note: We can ignore the exact number of steps required by a portion of the algorithm, as long as we know its complexity…

Can we safely say this algorithm is \( \Theta(n^2) \)?
Searching Sequences

def `indexOf[T](seq: Seq[T], value: T, from: Int): Int` = {
  for(i <- from until seq.length) {
    if(seq(i).equals(value)) { return i }
  }
  return -1
}

What is the complexity?
Searching Sequences

def indexOf[T](seq: Seq[T], value: T, from: Int): Int = {
  for(i <- from until seq.length) {
    if(seq(i).equals(value)) { return i }
  }
  return -1
}

What is the complexity? O(n)
Searching Sequences

def count[T](seq: Seq[T], value: T): Int =
    var count = 0;
    var i = indexOf(seq, value, 0)
    while(i != -1) {
        count += 1;
        i = indexOf(seq, value, i+1)
    }
    return count

What is the complexity?
def count[T](seq: Seq[T], value: T): Int ={
    var count = 0;
    var i = indexOf(seq, value, 0)
    while(i != -1) {
        count += 1;
        i = indexOf(seq, value, i+1)
    }
    return count
}
Searching Sequences

```python
def count[T](seq: Seq[T], value: T): Int ={
    var count = 0;
    var i = indexOf(seq, value, 0)
    while(i != -1) {
        count += 1;
        i = indexOf(seq, value, i+1)
    }
    return count
}
```

What is the complexity? O(n)? What about this line?
def count[T](seq: Seq[T], value: T): Int ={
    var count = 0;
    var i = indexOf(seq, value, 0)
    while(i != -1) {
        count += 1;
        i = indexOf(seq, value, i+1)
    }
    return count
}
Searching Sequences

```scala
def count[T](seq: Seq[T], value: T): Int ={
    var count = 0;
    var i = indexOf(seq, value, 0)
    while(i != -1) {
        count += 1;
        i = indexOf(seq, value, i+1)
    }
    return count
}
```

What is the complexity? Each element is only checked once, so O(n).
Searching Sorted Sequences

- Assuming $O(1)$ access to elements (‘random access’)
  - Divide the set of elements in half by taking the "middle" element, $m$
    - If $m$ is greater than what we are looking for, search the lower half
    - If $m$ is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half
Searching Sorted Sequences

- Assuming O(1) access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, $m$
    - If $m$ is greater than what we are looking for, search the lower half
    - If $m$ is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

*If you have $n$ elements, how many times can you divide $n$ in half?*
Searching Sorted Sequences

- Assuming $O(1)$ access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, $m$
    - If $m$ is greater than what we are looking for, search the lower half
    - If $m$ is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

If you have $n$ elements, how many times can you divide $n$ in half?

$log(n)$
Searching Sorted Sequences

- Assuming O(1) access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, $m$
    - If $m$ is greater than what we are looking for, search the lower half
    - If $m$ is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

If you have $n$ elements, how many times can you divide $n$ in half?

$log(n)$

Therefore the runtime of this search algorithm is $O(log(n))$