Last Time

• Some algorithms are fast, others are slow.
• Rather than timing an algorithm, we can evaluate its runtime through counting primitive operations.
• Seven orders of growth, ranging from constant time to exponential time.
# Orders of Growth

<table>
<thead>
<tr>
<th>Order</th>
<th>Function</th>
<th>Descriptor</th>
<th>n=10</th>
<th>n=20</th>
<th>n=30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>$f(n) = c$</td>
<td>Primitive function</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$f(n) = \log(n)$</td>
<td>Split list on all iterations</td>
<td>3.32</td>
<td>4.32</td>
<td>4.91</td>
</tr>
<tr>
<td>Linear</td>
<td>$f(n) = n$</td>
<td>For loop</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>“Linearithmic”</td>
<td>$f(n) = n \cdot \log(n)$</td>
<td>Fast sorting</td>
<td>33.2</td>
<td>86.4</td>
<td>147.3</td>
</tr>
<tr>
<td>Quadratic</td>
<td>$f(n) = n^2$</td>
<td>For loop inside for loop</td>
<td>100</td>
<td>400</td>
<td>900</td>
</tr>
<tr>
<td>Cubic</td>
<td>$f(n) = n^3$</td>
<td>For-For-For</td>
<td>1000</td>
<td>8000</td>
<td>27000</td>
</tr>
<tr>
<td>Exponential</td>
<td>$f(n) = 2^n$</td>
<td>Recursive Fibonacci</td>
<td>1024</td>
<td>1048756</td>
<td>1073926144</td>
</tr>
</tbody>
</table>
Adding an Item to a List

```java
public void add(int newInt) {
    Node newest = new Node(newInt, null);
    if (isEmpty()) {
        head = newest;
    } else {
        tail.setNext(newest);
    }
    tail = newest;
    size++;
}
```
Adding an Item to a List

```java
public Node(int newValue, Node nextNode) {
    value = newValue;
    next = nextNode;
} //Node (constructor)

public void setNext(Node newNext) {
    next = newNext;
} //setNext

public boolean isEmpty() {
    if (size == 0) {
        return true;
    } else {
        return false;
    } //if-else
} //isEmpty
```
Adding Many Items to a List

LL myList = new LL();
int N = 10000;

for (int i = 0; i < N; i++) {
  myList.add(i);
} //for

for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    myList.add(i*(j-3));
  } //for
} //for
Big-O Notation

• Let \( f(n) \) and \( g(n) \) be functions mapping positive integers to positive real numbers. We say that \( f(n) \) is \( O(g(n)) \) if there exists a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \).
Evaluating a Function

- **Theorem:** The function \( f(n) = 8n + 5 \) is \( O(n) \).

- **Proof:**
  - By the Big-O definition, we need to find a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \), such that \( 8n + 5 \leq cn \) for every integer \( n \geq n_0 \).
  - \( c = 9 \) is a constant for \( c \) such that \( cn \geq 8n \).
  - \( 9n \) starts growing faster than \( 8n + 5 \) at \( n_0 = 5 \).
  - Our definition of Big-O says that, since \( f(n) \leq c \ast g(n) \) for \( n \geq n_0 \), \( f(n) \) is \( O(g(n)) \).
  - \( g(n) = n \), so \( f(n) \) is \( O(n) \).
  - (Note that our choices for \( c \) and \( n_0 \) could change. \( c = 13 \) and \( n_0 = 1 \) also work. This doesn’t change the Big-O proof that \( f(n) \) is \( O(n) \) because \( g(n) \) is still \( n \).)
Big-O Rules

• Ignore constant factors.
  – $3n$, $6n$, $0.5n$, and $10000000000000n$ are all $O(n)$.

• Ignore lower-order terms.
  – $3n + 6$ is still $O(n)$, $4n^3 + 8n − 6$ is still $O(n^3)$, $2n + 100000000000000$ is still $O(n)$.

• Use the most specific order of growth.
  – $3n + 6$ is $O(n)$, but it’s also $O(n^2)$ and $O(2^n)$ for different $c$ and $n_0$ values.
Other Examples

```c
for (i = 0; i < N; i++) {
    for (j = 1; j < N; j=j*2) {
        sum++;
    } //for
} //for

for (i = 0; i < N; i+=2) {
    for (j = i; j < N; j++) {
        count--;  
    } //for
} //for
```
Any Questions?