Lecture 4: Asymptotic Analysis
Announcements

HW 0 – 143 Review Project
- Live on website
- Due Wednesday April 7th

Find a partner by Wednesday April 7th
- Groups of 3 are ok
Questions?
Queues
**Review: What is a Queue?**

**queue**: Retrieves elements in the order they were added.
- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

**Queue ADT**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>add(item) add item to back</td>
</tr>
<tr>
<td>Number of items</td>
<td>remove(item) remove and return item at front</td>
</tr>
<tr>
<td></td>
<td>peek() return item at front</td>
</tr>
<tr>
<td></td>
<td>size() count of items</td>
</tr>
<tr>
<td></td>
<td>isEmpty() count of items is 0?</td>
</tr>
</tbody>
</table>

**supported operations:**
- **add(item)**: aka “enqueue” add an element to the back.
- **remove()**: aka “dequeue” Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size()**: how many items are stored in the queue?
- **isEmpty()**: if 1 or more items in the queue returns true, false otherwise
Implementing a Queue with an Array

Queue ADT

**state**
- Set of ordered items
- Number of items

**behavior**
- `add(item)` add item to back
- `remove()` remove and return item at front
- `peek()` return item at front
- `size()` count of items
- `isEmpty()` count of items is 0?

ArrayQueue<E>

**state**
- `data[]`
- `Size`
- `front` index
- `back` index

**behavior**
- `add` - `data[Size] = value, if out of room grow data`
- `remove` - `return data[Size - 1], size-1`
- `peek` - `return data[Size - 1]`
- `size` - `return size`
- `isEmpty` - `return size == 0`

**Big O Analysis**
- `remove()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `add()` O(N) linear if you have to resize
  O(1) otherwise

---

Take 1 min to respond to activity

What do you think the worst possible runtime of the “add()” operation will be?
Implementing a Queue with an Array

> Wrapping Around

add(7)
add(4)
add(1)

front
back

4 5 9 2 7

front
back

numberOfItems = 5
Implementing a Queue with Nodes

Queue ADT

**state**
Set of ordered items
Number of items

**behavior**
- `add(item)` add item to back
- `remove()` remove and return item at front
- `peek()` return item at front
- `size()` count of items
- `isEmpty()` count of items is 0?

**LinkedQueue<E>**

**state**
- Node front
- Node back
- size

**behavior**
- `add()` add node to back
- `remove()` return and remove node at front
- `peek()` return node at front
- `size()` return size
- `isEmpty()` return size == 0

Big O Analysis

- `remove()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `add()` O(1) Constant

numberOfItems = 2

add(5)
add(8)
remove()

front 5 8 back

www.pollev.com/cse373activity
What do you think the worst case runtime of the “add()” operation will be?
Questions?
Dictionaries (aka Maps)

Every Programmer’s Best Friend

You’ll probably use one in almost every programming project.
- Because it’s hard to make a big project without needing one sooner or later.

// two types of Map implementations supposedly covered in CSE 143
Map<String, Integer> map1 = new HashMap<>();
Map<String, String> map2 = new TreeMap<>();
**Review: Maps**

**map**: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.  
- a.k.a. "dictionary"

---

**Dictionary ADT**

**state**
- Set of items & keys  
- Count of items

**behavior**
- `put(key, item)`: add item to collection indexed with key  
- `get(key)`: return item associated with key  
- `containsKey(key)`: return if key already in use  
- `remove(key)`: remove item and associated key  
- `size()`: return count of items

---

**supported operations:**
- `put(key, value)`: Adds a given item into collection with associated key,  
  - if the map previously had a mapping for the given key, old value is replaced.  
- `get(key)`: Retrieves the value mapped to the key  
- `containsKey(key)`: returns true if key is already associated with value in map, false otherwise  
- `remove(key)`: Removes the given key and its mapped value

---

**Example**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;at&quot;</td>
<td>43</td>
</tr>
<tr>
<td>&quot;in&quot;</td>
<td>37</td>
</tr>
<tr>
<td>&quot;you&quot;</td>
<td>22</td>
</tr>
<tr>
<td>&quot;the&quot;</td>
<td>56</td>
</tr>
</tbody>
</table>

**Map Example**

- `map.get("the")`: 56

---

**Keys**

<table>
<thead>
<tr>
<th>KEYS</th>
<th>VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>327.2</td>
</tr>
<tr>
<td>Feb</td>
<td>368.2</td>
</tr>
<tr>
<td>Mar</td>
<td>197.6</td>
</tr>
<tr>
<td>Apr</td>
<td>178.4</td>
</tr>
<tr>
<td>May</td>
<td>100.0</td>
</tr>
<tr>
<td>Jun</td>
<td>69.9</td>
</tr>
<tr>
<td>Jul</td>
<td>32.3</td>
</tr>
<tr>
<td>Aug</td>
<td>37.3</td>
</tr>
<tr>
<td>Sep</td>
<td>19.0</td>
</tr>
<tr>
<td>Oct</td>
<td>37.0</td>
</tr>
<tr>
<td>Nov</td>
<td>73.2</td>
</tr>
<tr>
<td>Dec</td>
<td>110.9</td>
</tr>
<tr>
<td>Annual</td>
<td>1551.0</td>
</tr>
</tbody>
</table>

---

CSE 373 19 SU - ROBBIE WEBER
## Implementing a Dictionary with an Array

### Dictionary ADT

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of items &amp; keys</td>
<td>put(key, item) add item to collection indexed with key</td>
</tr>
<tr>
<td>Count of items</td>
<td>get(key) return item associated with key</td>
</tr>
</tbody>
</table>

### ArrayDictionary<K, V>

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair&lt;K, V&gt;[] data</td>
<td>put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary</td>
</tr>
</tbody>
</table>

#### Big O Analysis – (if key is the last one looked at / not in the dictionary)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>get()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>remove()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

#### Big O Analysis – (if the key is the first one looked at)

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>get()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contains</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>containsKey('c')</td>
</tr>
<tr>
<td>'d'</td>
<td>get('d')</td>
</tr>
<tr>
<td>'b', 97</td>
<td>put('b', 97)</td>
</tr>
<tr>
<td>'e', 20</td>
<td>put('e', 20)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>('a', 1)</td>
</tr>
<tr>
<td>1</td>
<td>('b', 97)</td>
</tr>
<tr>
<td>2</td>
<td>('c', 3)</td>
</tr>
<tr>
<td>3</td>
<td>('d', 4)</td>
</tr>
<tr>
<td>4</td>
<td>('e', 20)</td>
</tr>
</tbody>
</table>
Implementing a Dictionary with Nodes

Dictionary ADT

**state**
- Set of items & keys
- Count of items

**behavior**
- `put(key, item)` add item to collection indexed with key
- `get(key)` return item associated with key
- `containsKey(key)` return if key already in use
- `remove(key)` remove item and associated key
- `size()` return count of items

Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()` \(O(N)\) linear
`get()` \(O(N)\) linear
`containsKey()` \(O(N)\) linear
`remove()` \(O(N)\) linear
`size()` \(O(1)\) constant

Big O Analysis – (if the key is the first one looked at)

`put()` \(O(1)\) constant
`get()` \(O(1)\) constant
`containsKey()` \(O(1)\) constant
`remove()` \(O(1)\) constant
`size()` \(O(1)\) constant

**LinkedDictionary<K, V>**

**state**
- `front`
- `size`

**behavior**
- `put` if key is unused, create new with pair, add to front of list, else replace with new value
- `get` scan all pairs looking for given key, return associated item if found
- `containsKey` scan all pairs, return if key is found
- `remove` scan all pairs, skip pair to be removed
- `size` return count of items in dictionary

Example:
- `containsKey('c')`
- `get('d')`
- `put('b', 20)`

```
'a' 1
'b' 20
'c' 9
'd' 4

front
```

CSE 373 19 SU - ROBBIE WEBER
Big O
**Review: Complexity Class**

**Complexity Class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-O</th>
<th>Runtime if you double N</th>
<th>Example Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>O(1)</td>
<td>unchanged</td>
<td>Accessing an index of an array</td>
</tr>
<tr>
<td>logarithmic</td>
<td>O(log₂ N)</td>
<td>increases slightly</td>
<td>Binary search</td>
</tr>
<tr>
<td>linear</td>
<td>O(N)</td>
<td>doubles</td>
<td>Looping over an array</td>
</tr>
<tr>
<td>log-linear</td>
<td>O(N log₂ N)</td>
<td>slightly more than doubles</td>
<td>Merge sort algorithm</td>
</tr>
<tr>
<td>quadratic</td>
<td>O(N²)</td>
<td>quadruples</td>
<td>Nested loops!</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>O(2ᴺ)</td>
<td>multiplies drastically</td>
<td>Fibonacci with recursion</td>
</tr>
</tbody>
</table>

Note: You don’t have to understand all of this right now – we’ll dive into it soon.
143 general patterns: “O(1) constant is no loops, O(n) is one loop, O(n^2) is nested loops”
- This is still useful!
- But in 373 we’ll go much more in depth: we can explain more about why, and how to handle more complex cases when they arise (which they will!)
Algorithmic Analysis: The overall process of characterizing code with a complexity class, consisting of:

- **Code Modeling**: Code → Function describing code’s runtime
- **Asymptotic Analysis**: Function → Complexity class describing asymptotic behavior
Code Modeling – the process of mathematically representing how many operations a piece of code will run in relation to the input size \( n \).
- Convert from code to a function representing its runtime
What Counts?

We don’t know exact runtime of every operation, but for now let’s try simplifying assumption: all basic operations take the same time

• Basics:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access

• Function Calls
  - Total runtime in body
  - Remember: new calls a function (constructor)

• Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch later

• Loops
  - Number of iterations * total runtime in condition and body
public void method1(int n) {
    int sum = 0;  
    int i = 0;  
    while (i < n) {
        sum = sum + (i * 3);  
        i = i + 1;  
    }
    return sum;  
}  

f(n) = 6n + 3
Code Modeling Example 2

public void method2(int n) {
    int sum = 0;
    int i = 0;
    while (i < n) {
        int j = 0;
        while (j < n) {
            if (j % 2 == 0) {
                // do nothing
            }
            sum = sum + (i * 3) + j;
            j = j + 1;
        }
        i = i + 1;
    }
    } return sum;
}
We just turned a piece of code into a function!
- We’ll look at better alternatives for code modeling later

Now to focus on step 2, asymptotic analysis

```java
for (i = 0; i < n; i++) {
    a[i] = 1;
    b[i] = 2;
}

f(n) = 2n
O(n)
```
We have an expression for $f(n)$. How do we get the $O(\cdot)$ that we’ve been talking about?

1. Find the “dominating term” and delete all others.
   - The “dominating” term is the one that is largest as $n$ gets bigger. In this class, often the largest power of $n$.

2. Remove any constant factors.

$$f(n) = (9n+3)n + 3$$

$$= 9n^2 + 3n + 3$$

$$\approx 9n^2$$

$$\approx n^2$$

$f(n)$ is $O(n^2)$
Can we really throw away all that info?

Big-Oh is like the “significant digits” of computer science

**Asymptotic Analysis:** Analysis of function behavior as its input approaches infinity
- We only care about what happens when \( n \) approaches infinity
- For small inputs, doesn’t really matter: all code is “fast enough”
- Since we’re dealing with infinity, constants and lower-order terms don’t meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

**Simple**
We don’t care about tiny differences in implementation, want the big picture result

**Decisive**
Produce a clear comparison indicating which code takes “longer”
Imagine you have three possible algorithms to choose between. Each has already been reduced to its mathematical model:

\[ f(n) = n \quad g(n) = 4n \quad h(n) = n^2 \]

The growth rate for \( f(n) \) and \( g(n) \) looks very different for small numbers of input. \( h(n) \) actually has a slower growth rate than either \( f(n) \) or \( g(n) \).

...but since both are linear eventually look similar at large input sizes. Whereas \( h(n) \) has a distinctly different growth rate.
**Definition: Big-O**

We wanted to find an upper bound on our algorithm’s running time, but
- We don’t want to care about constant factors.
- We only care about what happens as $n$ gets large.

**Big-O**

$f(n)$ is $O(g(n))$ if there exist positive constants $c, n_0$ such that for all $n \geq n_0$,

\[ f(n) \leq c \cdot g(n) \]

We also say that $g(n)$ “dominates” $f(n)$
Applying Big O Definition

Show that \( f(n) = 10n + 15 \) is \( O(n) \)

Apply definition term by term

\[
10n \leq c \cdot n \text{ when } c = 10 \text{ for all values of } n
\]

\[
15 \leq c \cdot n \text{ when } c = 15 \text{ for } n \geq 1
\]

Add up all your truths

\[
10n + 15 \leq 10n + 15n = 25n \text{ for } n \geq 1
\]

Select values for \( c \) and \( n_0 \) and prove they fit the definition

**Take \( c = 25 \) and \( n_0 = 1 \)**

\[
10n \leq 10n \text{ for all values of } n
\]

\[
15 \leq 15n \text{ for } n \geq 1
\]

So \( 10n + 15 \leq 25n \) for all \( n \geq 1 \), as required.

because a \( c \) and \( n_0 \) exist, \( f(n) \) is \( O(n) \)