CS310
Data Structures

K. Raven Russell
krusselc@gmu.edu
George Mason University

Today

- **Last Lecture**
  - Advanced Java Review
  - Algorithm Analysis and Big-O
- **Today**
  - Participation
  - Transfer students…
  - List Operations
  - Static and Dynamic Array Lists
- **Coming Soon**
  - Project 1 getting closer!

Lecture Structure Note

	Lecture X	Lecture (X+1)	Lecture (X+2)
~30 min	...	Review Lecture X new thing	Review Lecture (X+1) new thing
~30 min	Learn new thing	Learn new thing	...
~15 min	Practice new thing	Practice new thing	...

Questions About Participation/Projects?

Review + Details

Two (+1) Complexities

- **Time-complexity**
  - How much time given a lot of data?
  - Bad: if I need to sort 10 items it's fast, but if I need to sort 1000 and takes 3 hours
- **Space-complexity**
  - How much memory given a lot of data?
  - Bad: if I need to store 10 items it takes 10B but if need to sort 1000 it takes 10GB
- **Implementation-complexity**
  - How easy is it to implement (hardware/software)?
  - Generally, simpler algorithms are preferred.

Summary of Analysis Approaches

- **Empirical Analysis**
  - Run the program with different n
  - Reasonable approach if no access to the code
  - Can be used to predict performance
  - Machine specific and lacks understanding
- **Mathematical Analysis**
  - Can be used to predict performance
  - Independent of machine (instructions->cycles->time)
  - Very difficult to perform
- **Approximation Analysis**
  - Eliminates details to simplify model
  - e.g. Tilde (~), Big-Oh, Theta, Omega, etc.
  - Independent of machine
  - Can make statements concerning bounds
  - Typically cannot make predictions

Approximation Analysis

Big-O (Omicron) **formal version!**

- **T(n)** is **O(F(n))** if there are positive constants **c** and $\mathbf{n_0}$ such that
  - when **n >= n_0**
  - **0 <= T(n) <= c F(n)**

- Up next:
  - Parts of the formal definition
  - Show that: **2n+1 is O(n)**
  - Show that: **n^2+1 is NOT O(n)**

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

Your Function

T(n) is **O(F(n))** — **What You're Proving**

if there are positive constants **c** and $\mathbf{n_0}$ — **What You Need To Find**

such that when $\mathbf{n >= n_0}$ — **Conditions**

$\mathbf{0 <= T(n) <= c\ F(n)}$ — **Conditions**

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

Your Function

2n+1 is **O(F(n))** — **What You're Proving**

if there are positive constants **c** and **n_0** — **What You Need To Find**

such that when **$n >= n_0$**

$0 <= T(n) <= c\ F(n)$ — **Conditions**

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

Your Function

2n+1 is **O(n)** — **What You're Proving**

if there are positive constants **c** and $\mathbf{n_0}$ — **What You Need To Find**

such that when $\mathbf{n >= n_0}$ — **Conditions**

$\mathbf{0 <= T(n) <= c\ F(n)}$ — **Conditions**

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= T(n) <= c\ F(n)$

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ F(n)$

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ n$

Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

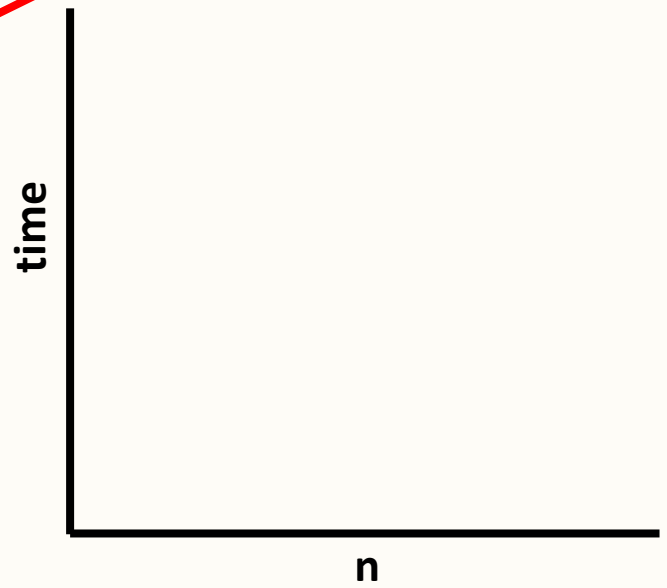
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ n$



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

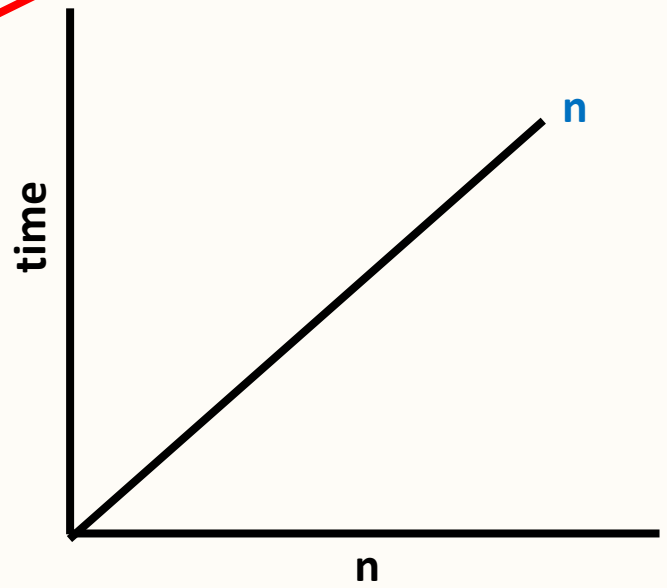
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ n$



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

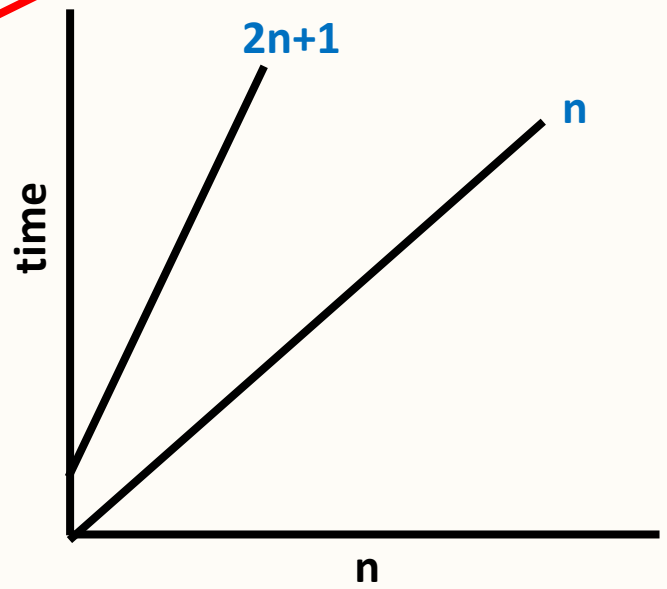
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ n$



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

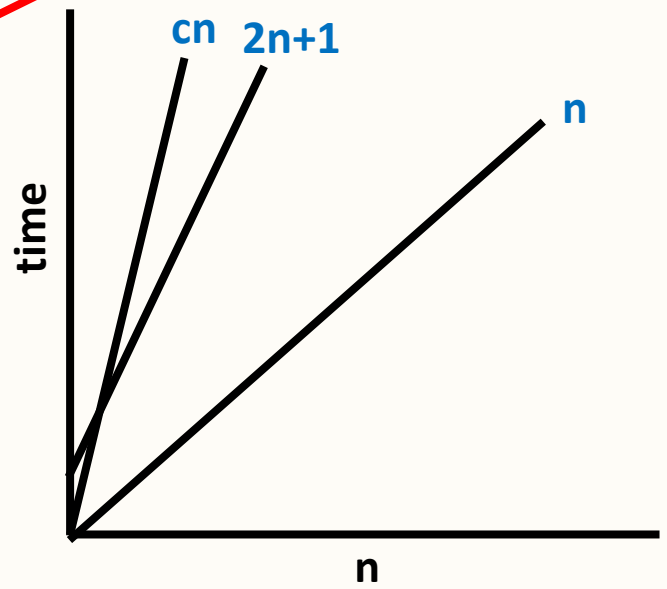
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= 2n+1 <= c\ n$



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

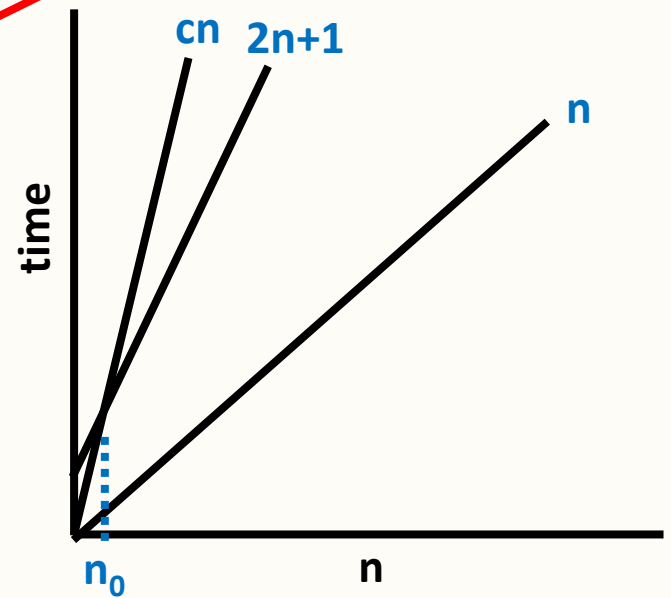
What You Need To Find

if there are positive constants **c** and n_0

such that when **n >= n_0**

Conditions

0 <= 2n+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

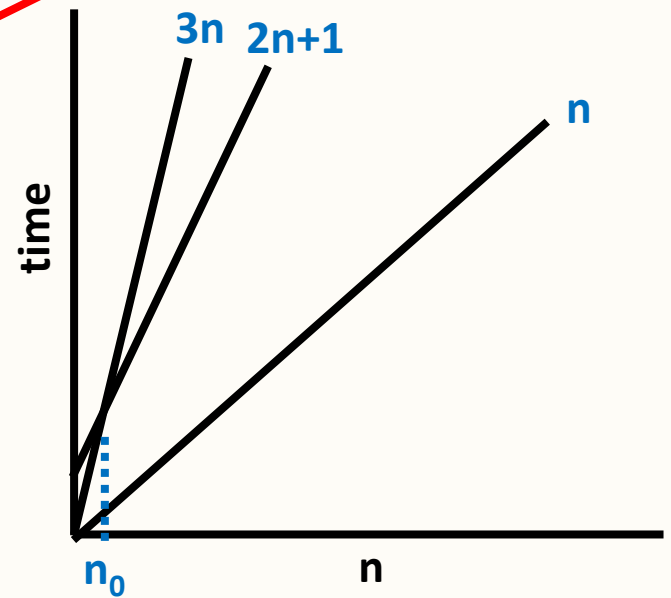
What You Need To Find

if there are positive constants **c** and n_0

such that when **n >= n_0**

Conditions

0 <= 2n+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition: 2n+1 is O(n)

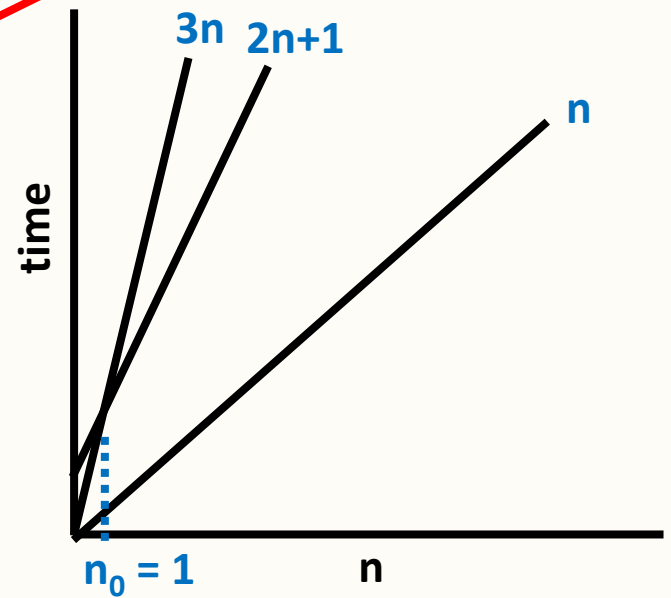
What You Need To Find

if there are positive constants **c** and n_0

such that when **n >= n_0**

Conditions

0 <= 2n+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

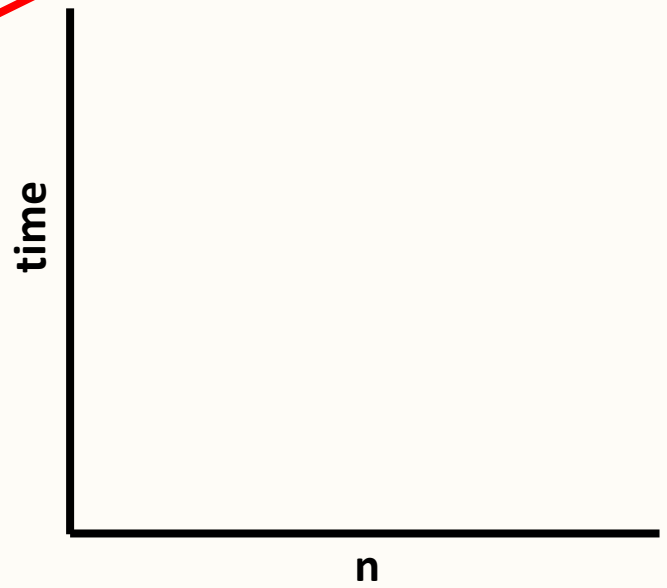
What You Need To Find

if there are positive constants **c** and **n_0**

such that when **n >= n_0**

Conditions

0 <= T(n) <= c F(n)



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

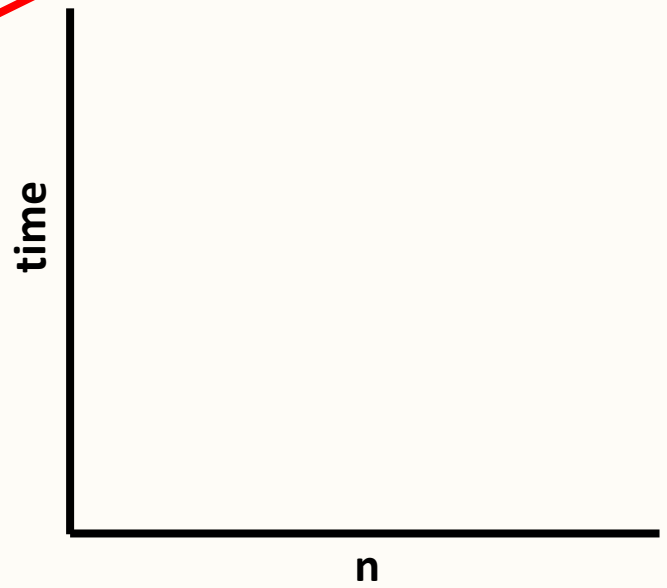
What You Need To Find

if there are positive constants **c** and **n_0**

such that when **n >= n_0**

Conditions

0 <= n^2+1 <= c F(n)



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

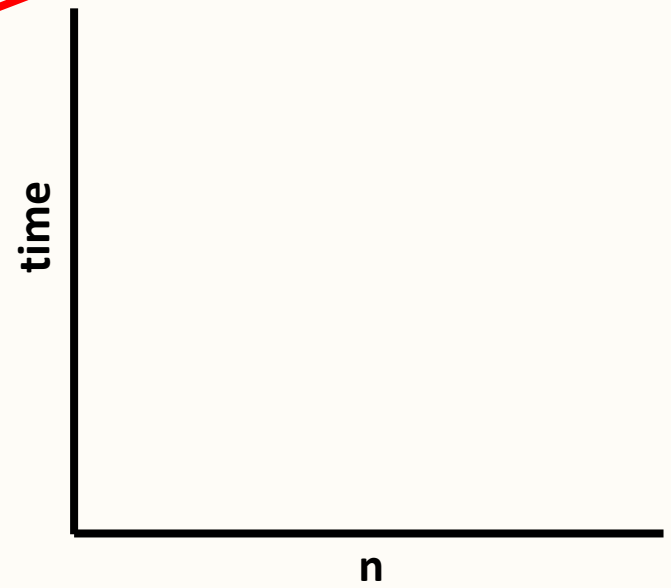
What You Need To Find

if there are positive constants **c** and **n_0**

such that when **n >= n_0**

Conditions

0 <= n^2+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

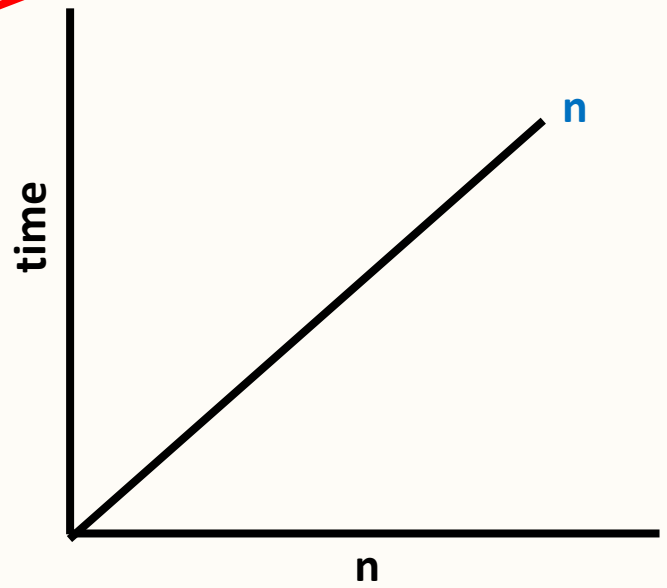
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= n^2+1 <= c\ n$



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

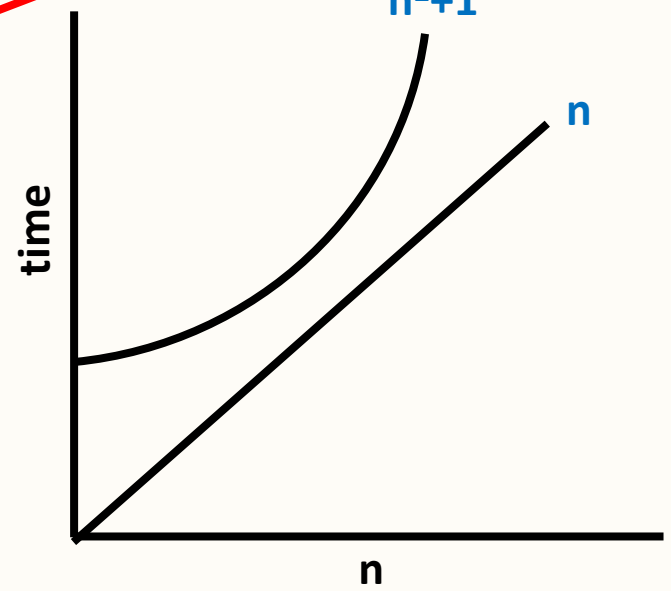
What You Need To Find

if there are positive constants **c** and **n_0**

such that when **n >= n_0**

Conditions

0 <= n^2+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

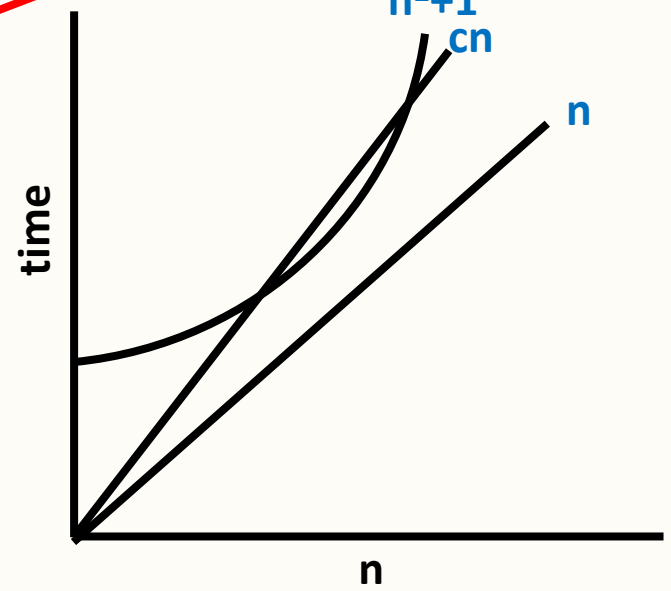
What You Need To Find

if there are positive constants **c** and **n_0**

such that when **n >= n_0**

Conditions

0 <= n^2+1 <= c n



Big-O (Omicron) formal version!

Prove by Definition FALSE: n^2+1 is O(n)

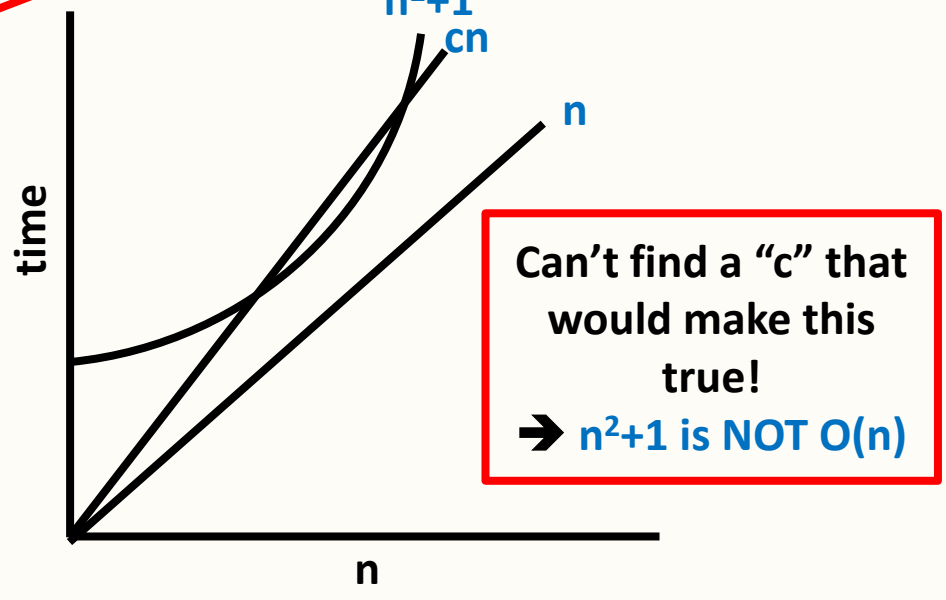
What You Need To Find

if there are positive constants **c** and n_0

such that when **$n >= n_0$**

Conditions

$0 <= n^2+1 <= c\ n$



Can't find a "c" that would make this true!

➔ **n^2+1 is NOT O(n)**

Other Notations

- Big O (Omicron): Upper bound, “big-oh”
  - T(n) is O(F(n)) if there are positive constants c and n0 such that
    - *when n >= n0*
    - *T(n) <= cF(n)*
- Big Ω (Omega): Lower bound
  - T(n) is Ω(F(n)) if there are positive constants c and n0 such that
    - *when n >= n0*
    - *T(n) >= cF(n)*
- Big Θ (Theta): Upper and lower bound
  - If something is O(F(n)) and Ω(F(n)) it is Θ(F(n))
- Little o (Omicron): T(n) grows much slower than F(n)
- Little ω (Omega): T(n) grows much faster than F(n)

Another Way to Think of It

Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \lessdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \leqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f \eqcirc g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
$f \in \Omega(g)$	$f \geqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f \gtrdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Image Source: http://stackoverflow.com/questions/1364444/difference-between-big-o-and-little-o-notation
See also: https://en.wikipedia.org/wiki/Big_O_notation#Family_of_Bachmann.E2.80.93Landau_notations

Approximation Algorithm Gotchas

(1) Algorithms vs. Programs

Algorithms vs. Programs

- Algorithm

  = how you do something

- Program

  = concrete instructions to a computer to do something

- Kinda like the relationship between classes and instances
  - Algorithm is the "blueprint"
  - Program is the "instantiation"

- Analyzing a *program* has a lot more gochas…

Sometimes it's sneaky...

- This code is **O(n) not O(n^2)... why?**

```
void method8(int n, int m) {
     for(int i = 0; i < 10; i++) {
          for(int j = 0; j < n; j++) {
               System.out.println("banana");
          }
     }
}
```

And it's not just what you do...

- This code is **$O(n^2)$ not $O(n)$... why?**
- How does concatenation work?

```
void method9(String[] arr) {
     String result = "[";
     for(String s : arr) {
          result += s + " ";
     }
     result += "]";
     System.out.println(result);
}
```

(2) Real vs. Theoretical Computers

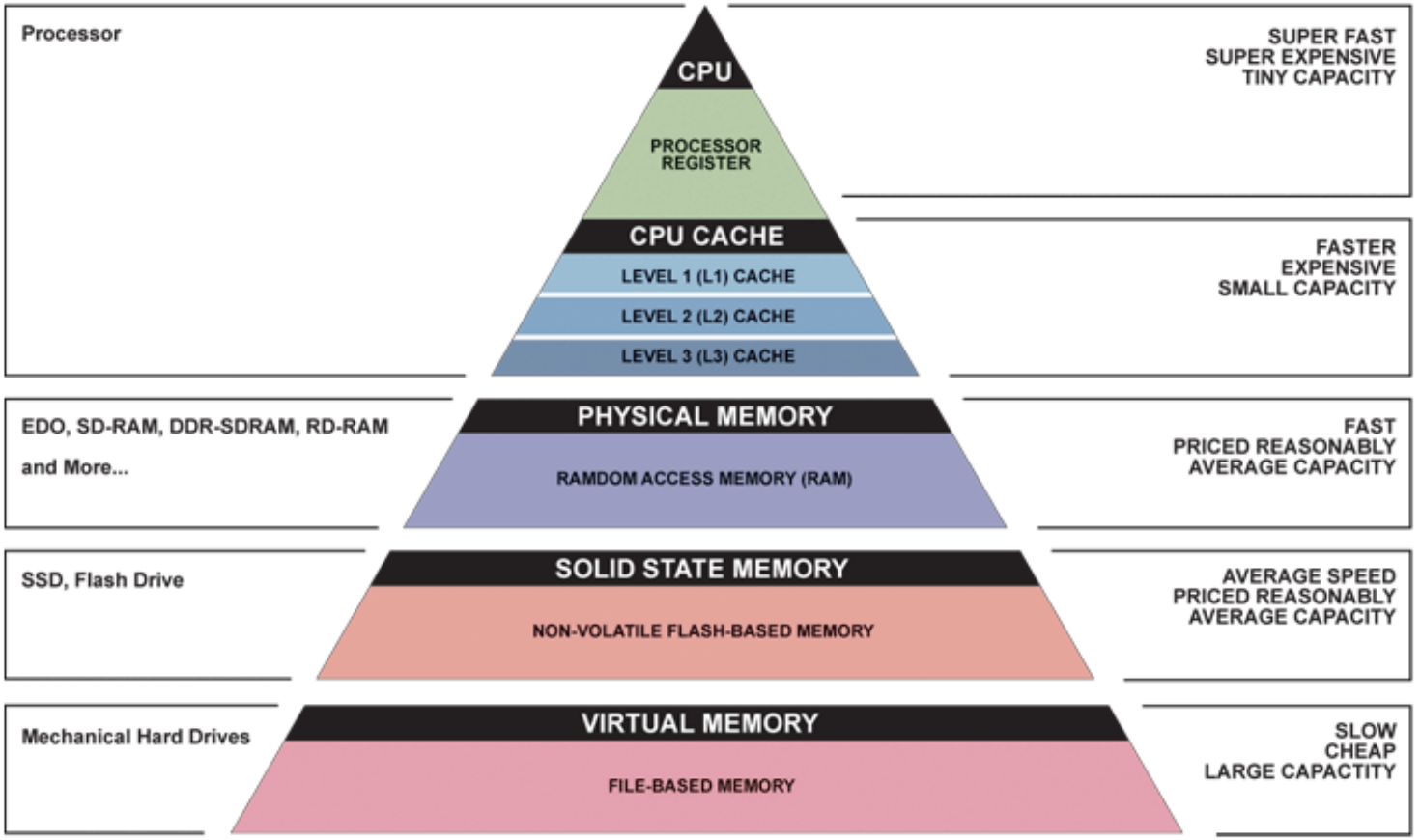
We're Lying to Ourselves…

CLRS: Introduction to Algorithms, Cormen, et al.

- These analysis techniques assume *uniform memory access*
  - commonly referred to as random-access machine (RAM) model
  - instructions executed one after the other
  - no concurrent operations
- Real life doesn't work that way
  - Some memory locations are "farther" away
  - This has a practical effect on performance

Memory Access Hierarchy

Image Source: http://www.bit-tech.net/hardware/memory/2007/11/15/ the_secrets_of_pc_memory_part_1/3



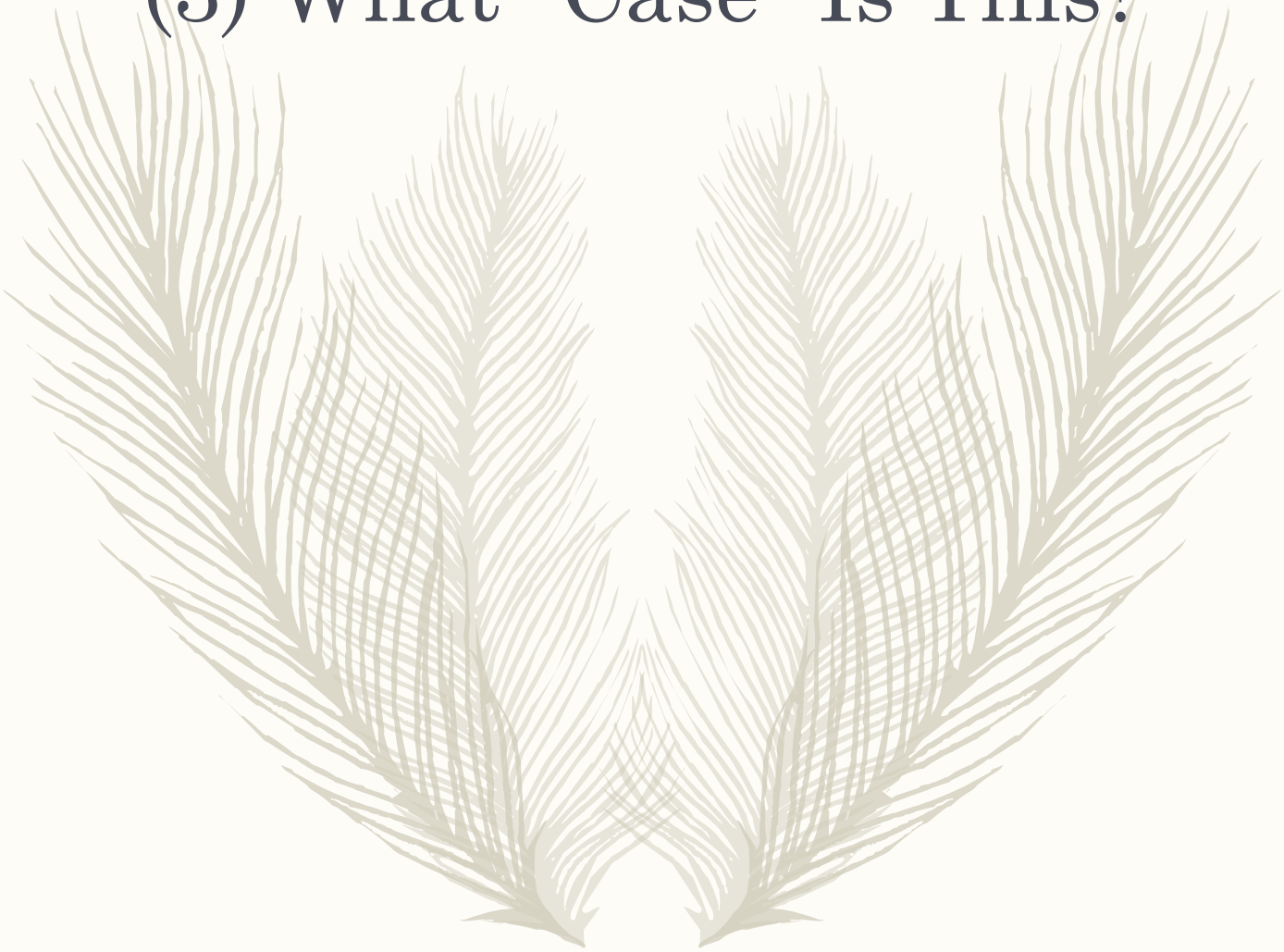
▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

“Numbers Everyone Should Know”

Reference	Time	Analogy
Register	-	Your brain
L1 cache reference	0.5 ns	Your desk
L2 cache reference	7 ns	Neighbor’s Desk
Main memory reference	100 ns	This Room
Disk seek	10,000,000 ns	Salt Lake City

Edited Excerpt From: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf

(3) What “Case” Is This?



Worst, Average, or Best Case?

- **Plan for the worst, hope for the best**

- Best case isn't usually too helpful in practice (such as O(1))
  - Any best cases for our “real life” examples?
    - *Pages? Dinner? Cats?*

- Average case can be helpful (typically requires statistics or probabilistic analysis to “prove” it)

- Worst case is the most important (usually)

No Relationship!

- I can have **any bound** (upper, lower, etc.)
- On **any case** (best, worst, average)

- Perfectly normal to ask for:
  - the "Big-O of the best case"
  - the "Little-Omega of the average case"

- Any combo is OK!

Questions?

For Home:
Big-O Challenge Problems!

Simple Practice Problems

- Give an example of an O(1) code with a loop.
- Give an example of a *single* loop that's O(n^3).
- Give an example of a *recursive* O(n) method where n is the size of some array.
- Is there a difference in the Big-O for the following problems in the worst case?

```
void foo(int n) {
    if(n < 100) {
        for(int i=0; i<n; i++) {
            //do something O(1)
        }
    }
    else {
        //do something O(1)
    }
}
```

```
void foo(int n) {
    if(n > 100) {
        for(int i=0; i<n; i++) {
            //do something O(1)
        }
    }
    else {
        //do something O(1)
    }
}
```

More Practice Problems

- Explain what the "worst case" is for the code below, and give a tight Big-O bound (using the formal definition):

```
void foo(int[] arr) {
    for(int i = 0; i < arr.length; i++) {
        if(arr[i] % 2 == 0) { break; }
        else {
            for(int i = 0; i < arr.length; i++) {
                //do something O(1)
            }
        }
    }
}
```

- There could be an average case that's different from the worst case. If so, what would we need to assume about our data to for there to be an average case?

```
//Loop 1:
for(int i = 0; i < a; i++) {
    for(int j = 0; j < a; j++) {
        for(int k = 0; k < a; k++) {
            for(int m = 0; m < a; m++) {
                //do something O(1)
            }
        }
    }
}
//Loop 2:
for(int i = 0; i < a; i++) {
    for(int j = 0; j < b; j++) {
        for(int k = 0; k < c; k++) {
            for(int m = 0; m < d; m++) {
                //do something O(1)
            }
        }
    }
}
//Loop 3:
for(int i = 0; i < a; i++) {
    for(int j = 0; j < a; j++) {
        for(int k = 0; k < b; k++) {
            for(int m = 0; m < b; m++) {
                //do something O(1)
            }
        }
    }
}
```

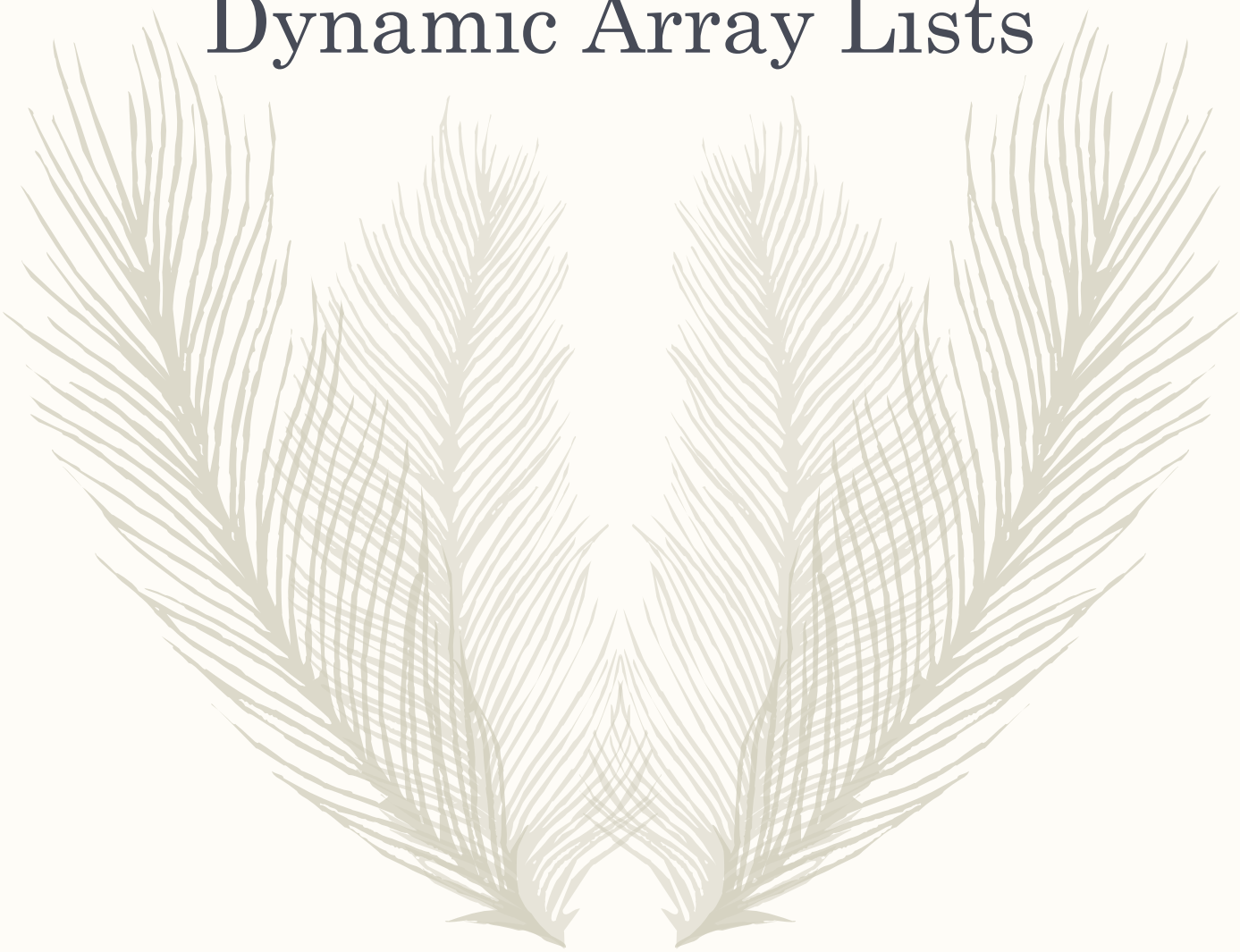
More Practice Problems

Assume a, b, c, and d are positive variables in the same way as n or m are in the class examples (and always >= 1).

What is the Big-O?

Can you *prove* these with the formal definition?

Dynamic Array Lists



What is a List?

- On a normal paper-and-pencil list I can do the following:
  - **set** a value (at some index)
  - **get** a value (at some index)
  - **append** a value (to the end)
  - **add/insert** a value (at some index)
  - **remove** a value (at some index)
  - **search** for a value (and return the index)

- How do we make an array work like a pencil-and-paper list?

Whiteboard

Basic Outline…

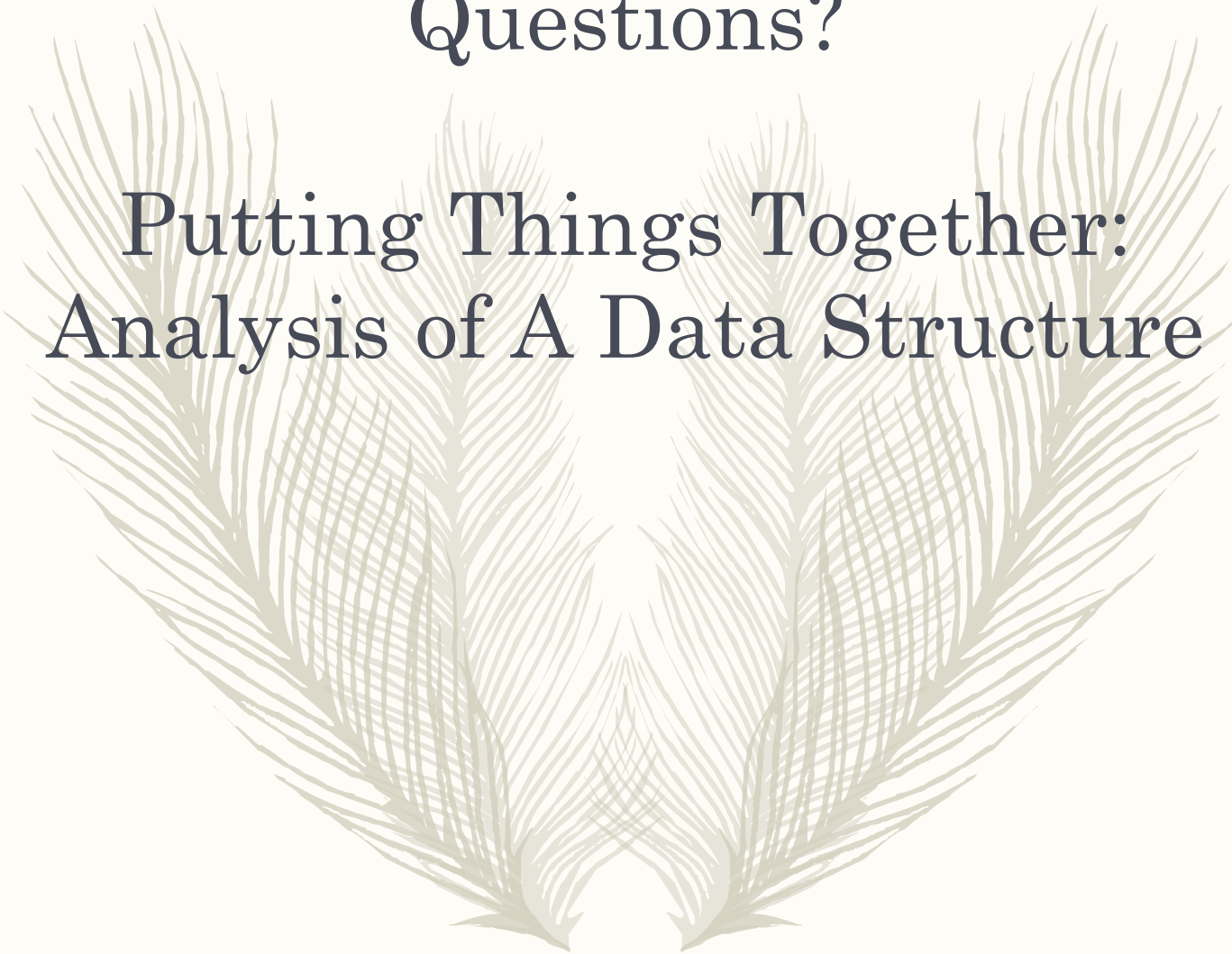
- **set()**
  - **parameters**: int index, Object to put there
  - **return**: ??
- **get()**
  - **parameters**: int index
  - **return**: Object at index
- **add/append()**
  - **parameters**: Object to add
  - **return**: ??

Remember to check for edge cases!

- **add/insert()**
  - **parameters**: int index, Object to add
  - **return**: ??
  - **Other types of add**... parameters: Object?
- **remove()**
  - **parameters**: int index
  - **return**: ??
  - **Other types of remove**... parameters: Object?
- **search()**
  - **parameters**: Object to find
  - **return**: int index found (or -1 for not found)

Questions?

Putting Things Together: Analysis of A Data Structure



Your Turn! Big-O

- What is the Big-O of the following operations:
  - set a value (at some index)
  - get a value (at some index)
  - append a value (to the end)
  - add/insert a value (at some index)
  - remove a value (at some index)
  - search for a value (and return the index)

- Hint: is adding/removing different from the front/middle/end?
  - what does this say about the best/worst case?

Data Structure Operations Big-O

– Limitations of these structures?

Operation Implementation	get set	add remove end	insert remove begin	insert remove middle	search	grow?
Array	1	-	-	-	-	no
List (Static Array)	1	1	n	n	n	no

Data Structure Operations Big-O

- Later This Semester: We'll talk about the "n?" using a new analysis technique!

Operation Implementation	get set	add remove end	insert remove begin	insert remove middle	search	grow?
Array	1	-	-	-	-	no
List (Static Array)	1	1	n	n	n	no
List (Dynamic Array)	1	**n?**	n	n	n	Yes

Time?

How Does Java Do It?

Java ArrayLists

- Java: ArrayLists
- Data structure like a paper and pencil list
- Not enough space?
  - more paper
  - copy things over to the new paper

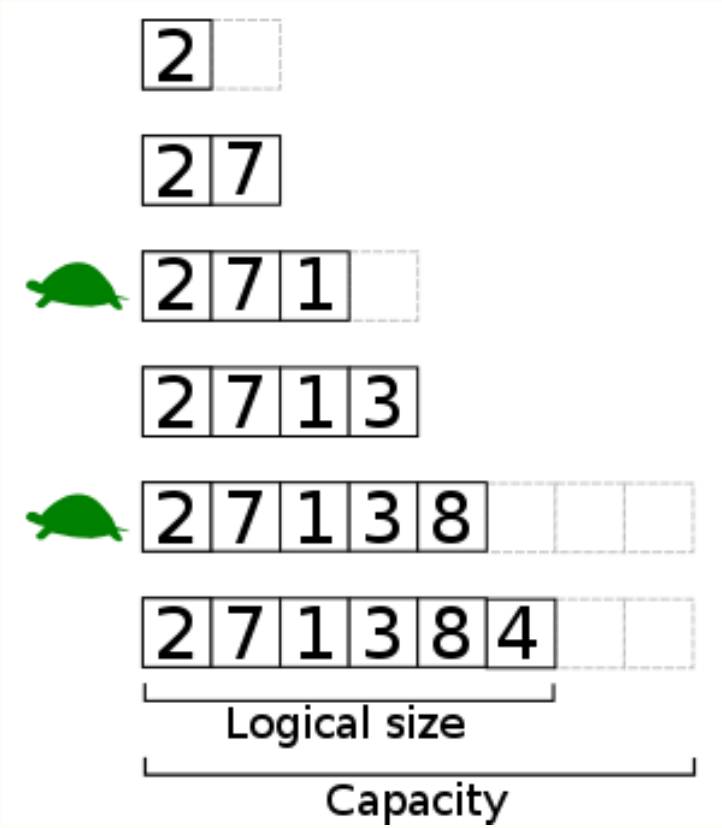


Image Source: http://en.wikipedia.org/wiki/File:Dynamic_array.svg

Java's ArrayList<E>

```
public E set(int index, E element) {
    rangeCheck(index);

        E oldValue = elementData(index);
        elementData[index] = element;
    return oldValue;
}

E elementData(int index) {
    return (E) elementData[index];
}
```

Java's ArrayList<E>

```
public E get(int index) {
    rangeCheck(index);
    return elementData(index);
}

E elementData(int index) {
    return (E) elementData[index];
}
```

Java's ArrayList<E>

```
public boolean add(E element) {
     //[...]

     elementData[size++] = e;
     return true;
}
```

Java's ArrayList<E>

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    //[...]

    System.arraycopy(elementData, index,
                     elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

Java's ArrayList<E>

```
public E remove(int index) {
     rangeCheck(index);

     modCount++;
     E oldValue = elementData(index);

     int numMoved = size - index - 1;
     if (numMoved > 0)
          System.arraycopy(elementData,
                           index+1, elementData,
                           index, numMoved);
     // Let gc do its work
     elementData[--size] = null;

     return oldValue;
}
```

Java's ArrayList<E>

```
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

Java's ArrayList<E>

```
public boolean add(E element) {
    ensureCapacityInternal(size + 1);

    elementData[size++] = e;
    return true;
}
```

Java's ArrayList<E>

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);

    System.arraycopy(elementData, index,
                     elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

Java's ArrayList<E>

```
private void ensureCapacityInternal(
                              int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

Java's ArrayList<E>

```
private void grow(int minCapacity) {
     // overflow-conscious code
     int oldCapacity = elementData.length;
     int newCapacity = oldCapacity +
                       (oldCapacity >> 1);
     if (newCapacity - minCapacity < 0)
          newCapacity = minCapacity;
     if (newCapacity - MAX_ARRAY_SIZE > 0)
          newCapacity = hugeCapacity(minCapacity);

     // minCapacity is usually close to size, so
     //         this is a win:
     elementData = Arrays.copyOf(elementData,
                                 newCapacity);
}
```

Time? Honor Court

- What is the HC?
  - Why *are* students referred to the HC?
  - What's the purpose of an HC compared to letting profs deal with it?
- Does the CS department "try" to send students to the HC?
  - It would be much easier for us NOT refer cases.
  - There is nothing in it for us. Trust me.
  - Reddit said…
- So why are so many students referred to the HC from CS?
  - Simple math: there are ~720 students in CS112 this semester, if 1/20 people decided to cheat, that would be 31 people…
  - Easy != Right
- So why do you do it? Why does it matter if people cheat?

Final Questions?