

Midterm Review

Elyse Cornwall and Amrita Kaur

July 13, 2023

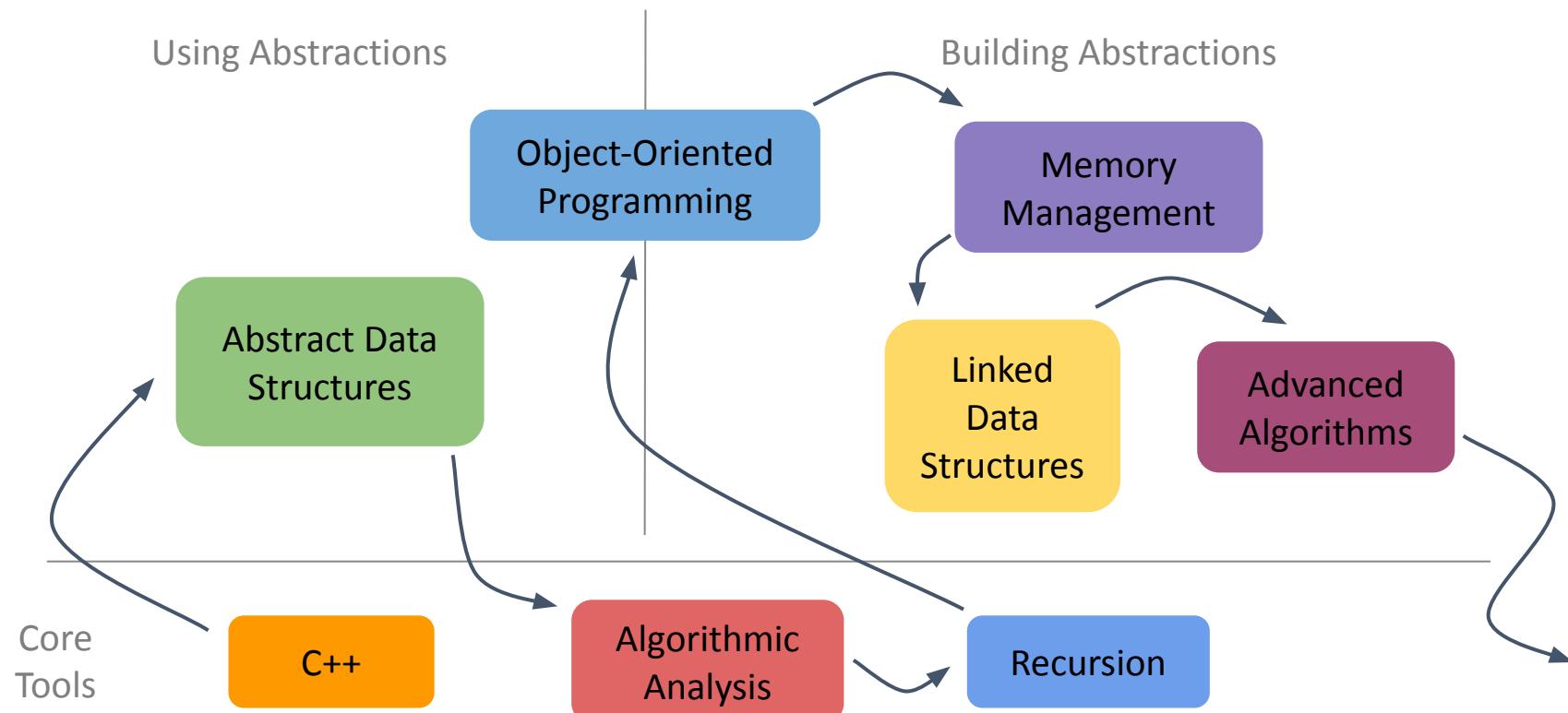
Midterm Logistics

- Monday, July 17 from 7-9pm in Hewlett Teaching Center, Room 200
 - Students with exam accommodations have already been contacted
- On paper, using pen/pencil
- Closed-book and closed-device.
 - Reference sheet on Stanford library functions
 - Notes sheet (one page, front and back, 8-1/2" x 11", have anything you want on it)
- All information is [here](#)

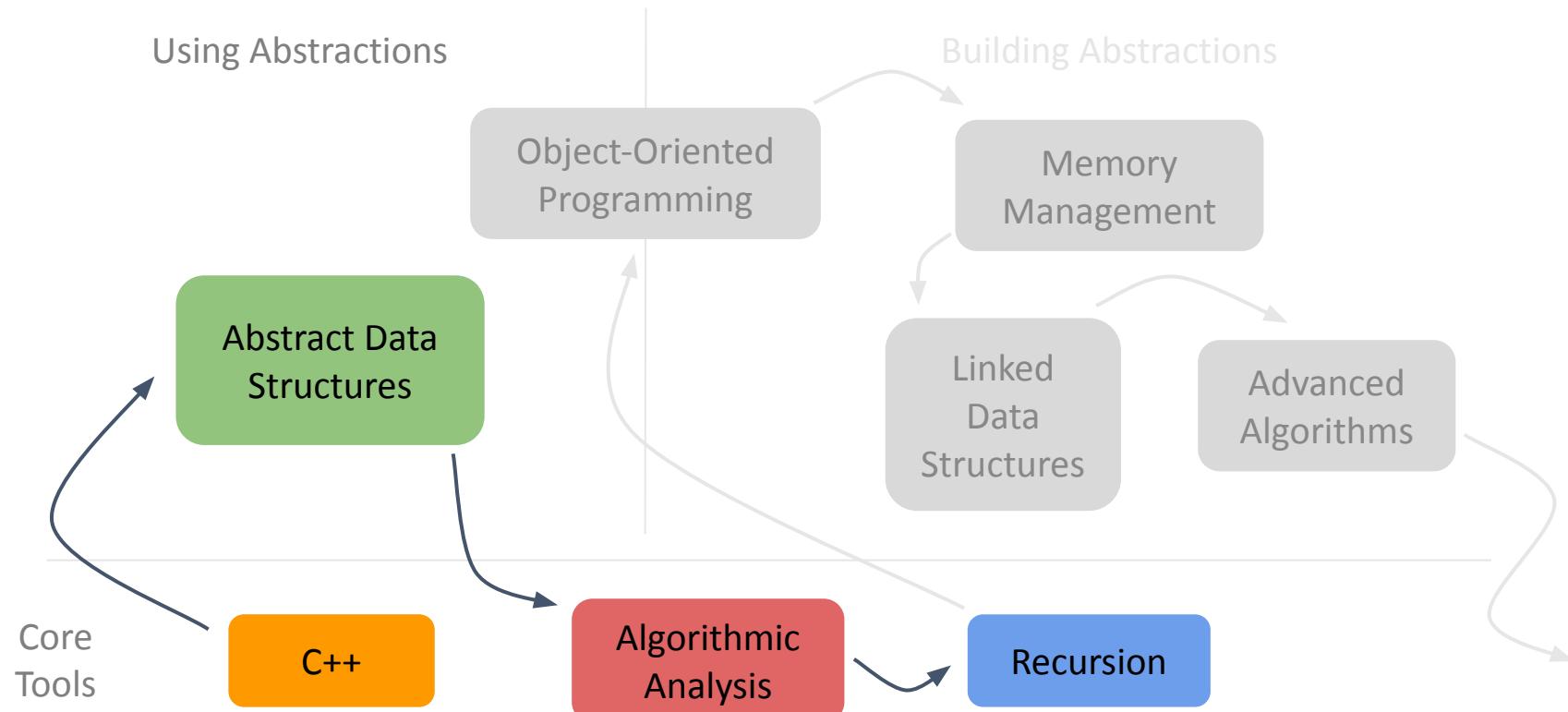
Midterm Logistics

- Evaluate your problem-solving skills and conceptual understanding of the material, not your ability to use perfect syntax
 - Most points awarded for valid approach to solving the problem, fewer points for the minute details of executing your plan
- Not taking off points for
 - Missing braces around clearly indented blocks of code
 - Missing semicolons
 - Missing `#include`

Roadmap



Roadmap - Midterm Coverage



C++ Fundamentals

Variables

- We use variables to store information in our programs
- Variables have a *type* and a *name*

```
int enrollment;
```

```
string className;
```

We name variables using “camelCase” capitalization

C++ Types

Numbers

- `int, long` // 100
- `float, double` // 3.14

Text

- `char, string` // 'a', "apple"

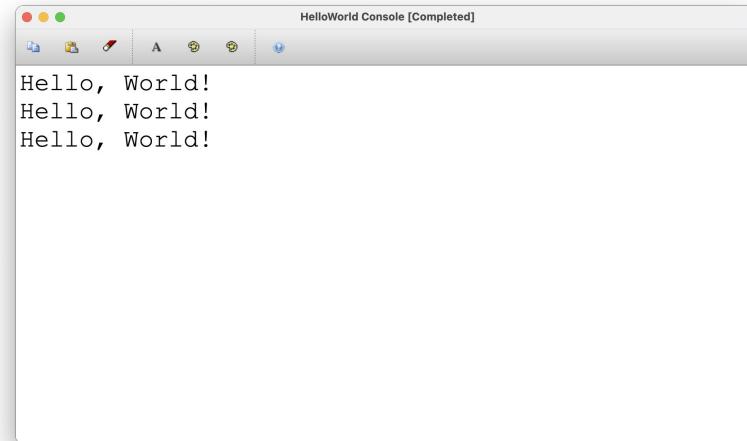
Booleans

- `bool` // true, false

Console Output

- We use `cout` and `<<` to print information to the user
- To start printing on a new line, we use `endl`

```
int main()
{
    cout << "Hello, World!" << endl;
    cout << "Hello, World!" << endl;
    cout << "Hello, World!" << endl;
    return 0;
}
```



Functions



Parameters

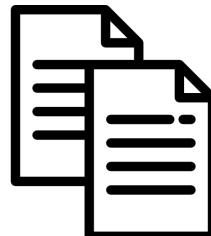
Function

Return

Passing Parameters into Functions

Pass by value

- Callee gets a **copy** of a variable from the caller function
- Changes to that variable that occur in callee do not persist in caller



Pass by reference

- Callee gets a **reference** to a variable from the caller function
- Now, the callee can directly modify the original variable



Passing Parameters into Functions

```
void valueFunc(Vector<int> vec) {  
    vec[0] = 100;  
}
```

Whoever calls valueFunc will give this function a copy of their Vector.

```
void refFunc(Vector<int>& vec) {  
    vec[0] = 100;  
}
```

Whoever calls refFunc will give this function access to their Vector.

Passing Parameters into Functions

```
void valueFunc(Vector<int> vec) {  
    vec[0] = 100;  
}
```

Vector<int> vec = {1, 2, 3};



```
valueFunc(vec);
```

// valueFunc doesn't change our Vector
EXPECT_EQUAL(vec, {1, 2, 3});

```
void refFunc(Vector<int>& vec) {  
    vec[0] = 100;  
}
```



```
refFunc(vec);
```

// refFunc does change our Vector!
EXPECT_EQUAL(vec, {100, 2, 3});

When Do We Pass by Reference?

Yes:

- When we want the callee function to edit our data
- To avoid making copies of large data structures
- When we need to return multiple values

No:

- Just because
- Passing by reference is risky because another function can modify your data!
- When the data we're passing to the callee is small, and thus copying isn't expensive

Conditionals

```
// assuming age variable is already defined
if (age < 12) {
    cout << "Eligible for kids meal.";
} else if (age > 65) {
    cout << "Eligible for senior discount.";
} else {
    cout << "Must use regular menu.";
}
```

While Loops

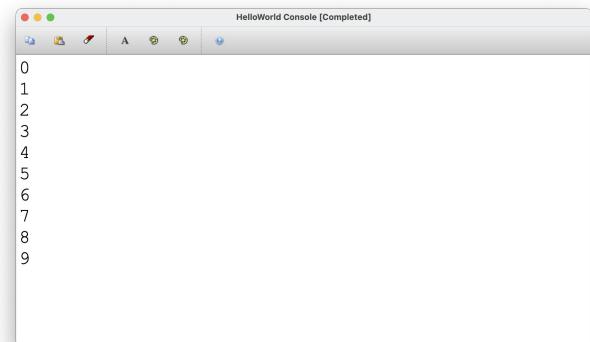
- “While this condition is true, do this”
- Use when you don’t know how many times you want to repeat

```
while (condition) {  
    // code to repeat while condition is true  
}
```

For Loops

- Use when you know how many times you want to repeat
- Typical for loop uses int counter *i* that starts at 0:

```
for (int i = 0; i < 10; i++) {  
    cout << i << endl;  
}
```



For Each Loops

- We can also loop for each element in a collection
 - Vectors, Grids, Sets, Maps

```
Vector<int> vec = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
for (int elem : vec)  
    cout << elem << endl;  
}
```



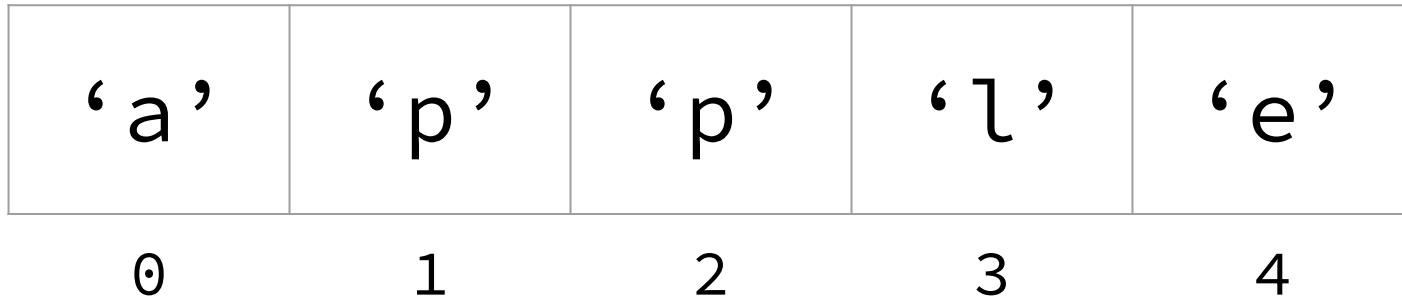
String

- Data type that represents a sequence of characters
- Marked by double quotes
- Ex: “apple”

Char

- Data type that represents a single character (letters, digits, symbols)
- Marked by single quotes
- Ex: ‘a’
- Have numerical representation (ASCII codes)

“apple”



Key Characteristics of Strings

- Mutable in C++ `word[1] = 'q';`
- Concatenated using + or +=
 - Add strings and strings, output is a string
 - Add strings with chars, output is a string
 - Adding chars will NOT give a string output
- Compared using relational operators (<, >, ==, !=)

C Strings and Strings

```
string word = "apple" + "sauce";
```

- Concatenating C strings with +
- Not possible (does not compile)

```
string word1 = "apple";
```

```
string word = word1 + "sauce";
```

- Concatenating C++ and C string with +
- Works perfectly! (autoconversion of C string)

Libraries for Strings and Chars

- <cctype> library
 - Built-in C++ char methods
- <string> library
 - Built-in C++ string methods
- “strlib.h” library
 - Stanford string functions

Don't Memorize - You'll Have a Reference

Strings

<code>str.at(i) or s[i]</code>	character at a given 0-based index in the string
<code>str.append(str)</code>	add text to the end of a string (<i>in-place</i>)
<code>str.compare(str)</code>	return -1, 0, or 1 depending on relative ordering
<code>str.erase(i, length)</code>	delete text from a string starting at given index (<i>in-place</i>)
<code>str.find(str)</code>	returns the first or last index where the start of the given string or character appears in this string (or <code>string::npos</code> if not found)
<code>str.rfind(str)</code>	
<code>str.insert(i, str)</code>	add text into a string at a given index (<i>in-place</i>)
<code>str.length() or str.size()</code>	number of characters in this string
<code>str.replace(i, len, str)</code>	replaces <code>len</code> chars at given index with new text (<i>in-place</i>)
<code>str.substr(start, length) or str.substr(start)</code>	returns the next <code>length</code> characters beginning at index <code>start</code> (inclusive); if <code>length</code> is omitted, grabs from <code>start</code> to the end of the string
<code>endsWith(str, suffix), startsWith(str, prefix)</code>	returns true if the string begins or ends with the given prefix/suffix
<code>integerToString(int), stringToInteger(str)</code>	returns a conversion between numbers and strings
<code>stringContains(str, substr)</code>	true if <code>substr</code> contained in <code>str</code>
<code>stringSplit(str, separator)</code>	breaks apart a string into a Vector of substrings divided by separator
<code>toLowerCase(str), toUpperCase(str)</code>	returns an upper/lowercase version of a string
<code>trim(str)</code>	returns string with any surrounding whitespace removed

char

<code>isalpha(c), isdigit(c), isspace(c),</code> <code>ispunct(c), islower(c), isupper(c),</code> <code>tolower(c), toupper(c)</code>	returns true if character is an alphabetic character from a-z or A-Z, a digit from 0-9, a whitespace character (space, \t, \n, etc.), a punctuation mark (#, \$, !, etc.) respectively returns lower/uppercase equivalent of a character (unchanged if not alpha)
---	--

Practice problem: Hashtags

Write a function `Vector<string> findHashtags(string s)` that returns a Vector of all of the hashtags in the string `s`. A hashtag starts with a '#' and ends with a space or the end of the string.

- `findHashtags("CS is #so #cool")` returns `{"#so", "#cool"}`
- `findHashtags("#what #is good")` returns `{"#what", "#is"}`
- `findHashtags("nothing to see here")` returns `{}`

```
Vector<string> findHashtags(string s) {
    Vector<string> result;
    bool inHashtag = false;
    string curWord = "";
    for (char ch: s) {
        if (ch == '#') {
            inHashtag = true;          // start of hashtag
        } else if (ch == ' ' && inHashtag) {
            inHashtag = false;        // end of hashtag, add to Vector
            result.add(curWord);
            curWord = "";
        }
        if (inHashtag) {
            curWord += ch;
        }
    }
    if (inHashtag) {
        result.add(curWord);        // add hashtag if it came at the end of s
    }
    return result;
}
```



Abstract Data Types (ADTs)

Abstract Data Type (ADTs)

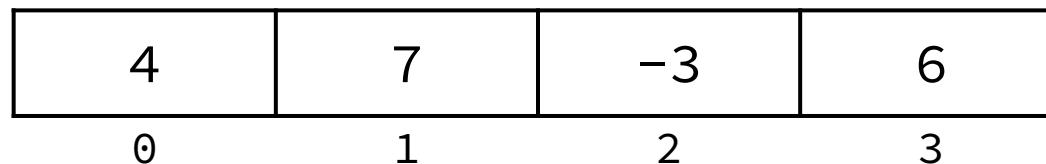
- Aka containers or data structures
- Allow programmers to store data in predictable, organized ways
- Can use without understanding the underlying implementation

Big Questions

- What type of data is stored in each ADT?
- How can you manipulate the data in each ADT?
- When would you want to use each specific ADT?
- What are the similarities and differences between the ADTs?

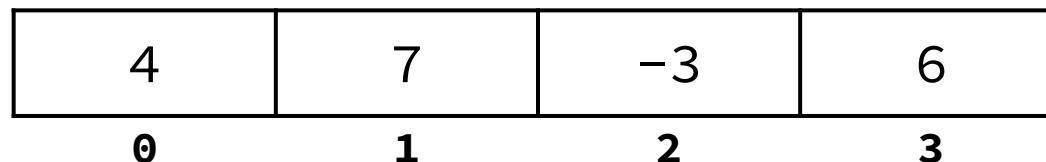
Vectors

- Ordered (indexed)
- 1-dimensional
- Can grow and shrink in size
- All elements must be of the same type



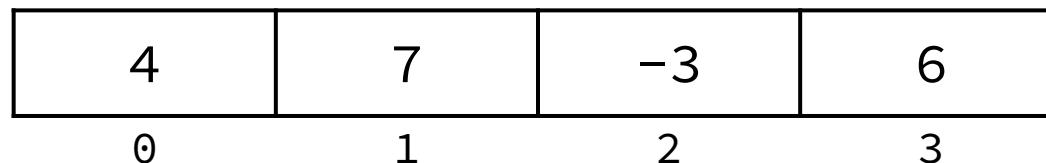
Vectors

- **Ordered (indexed)**
- 1-dimensional
- Can grow and shrink in size
- All elements must be of the same type



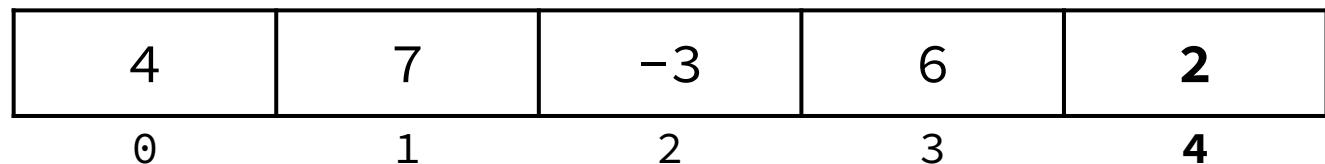
Vectors

- Ordered (indexed)
- **1-dimensional**
- Can grow and shrink in size
- All elements must be of the same type



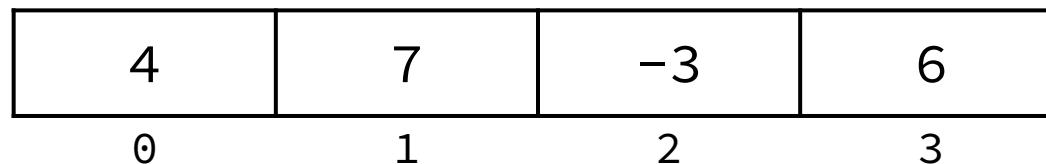
Vectors

- Ordered (indexed)
- 1-dimensional
- **Can grow and shrink in size**
- All elements must be of the same type



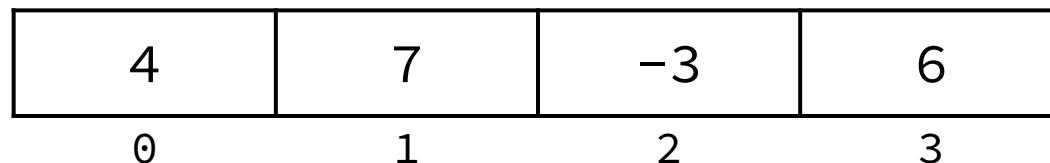
Vectors

- Ordered (indexed)
- 1-dimensional
- **Can grow and shrink in size**
- All elements must be of the same type



Vectors

- Ordered (indexed)
- 1-dimensional
- Can grow and shrink in size
- **All elements must be of the same type**



The Stanford Vector Library

- `vec.size()`: Returns the number of elements in the vector.
- `vec.isEmpty()`: Returns true if the vector is empty, false otherwise.
- `vec[i]`: Selects the i th element of the vector.
- `vec.add(value)`: Adds a new element to the end of the vector.
- `vec.insert(index, value)`: Inserts the value before the specified index, and moves the values after it up by one index.
- `vec.remove(index)`: Removes the element at the specified index, and moves the rest of the elements down by one index.
- `vec.clear()`: Removes all elements from the vector.
- `vec.sort()`: Sorts the elements in the list in increasing order.

For more information, check out the Stanford Vector class [documentation](#)!

Grids

- Ordered (rows and cols are indexed)
- 2-dimensional
- Fixed dimensions
- All elements must be of the same type

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

Grids

- **Ordered (rows and cols are indexed)**
- 2-dimensional
- Fixed dimensions
- All elements must be of the same type

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

Grids

- Ordered (rows and cols are indexed)
- **2-dimensional**
- Fixed dimensions
- All elements must be of the same type

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

Grids

- Ordered (rows and cols are indexed)
- 2-dimensional
- **Fixed dimensions**
- All elements must be of the same type

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

Grids

- Ordered (rows and cols are indexed)
- 2-dimensional
- Fixed dimensions
- **All elements must be of the same type**

	0	1	2
0	2	5	-1
1	10	11	3
2	19	-4	-2
3	4	6	2

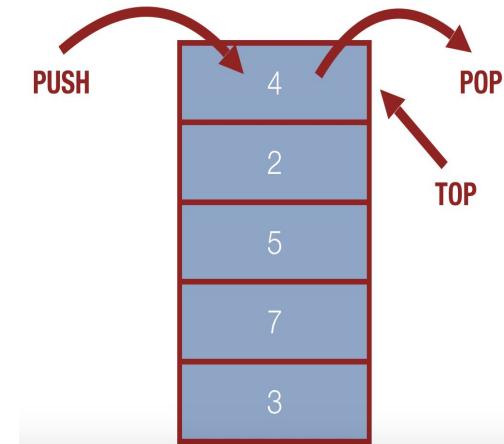
The Stanford Grid Library

- `grid numRows()`: Returns the number of rows in the grid.
- `grid numCols()`: Returns the number of columns in the grid.
- `grid[i][j]`: selects the element in the ith row and jth column.
- `grid.resize(rows, cols)`: Changes the dimensions of the grid and re-initializes all entries to their default values.
- `grid.inBounds(row, col)`: Returns true if the specified row, column position is in the grid, false otherwise.

For more information, check out the Stanford Grid [documentation!](#)

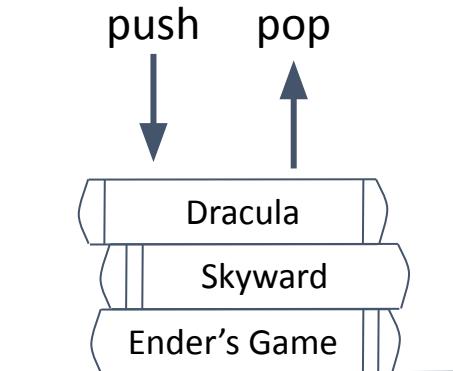
Stack

- Ordered (not indexed)
- Last In, First Out (LIFO)
- Only the top element of the stack is accessible
- Important operations:
 - **stack.push(value)**: Add an element onto the top of the stack
 - **stack.pop()**: Remove an element from the top of the stack and return it
 - **stack.peek()**: Look at the element from the top of the stack, but don't remove it



The Stanford Stack Library

- **stack.push(value)**: Add an element onto the top of the stack
- **stack.pop()**: Remove an element from the top of the stack and return it
- **stack.peek()**: Look at the element from the top of the stack, but don't remove it
- **stack.isEmpty()**: Returns a boolean value, true if the stack is empty, false if it has at least one element
 - Note: a runtime error occurs if a pop() or peek() operation is attempted on an empty stack
- **stack.clear()**: Removes all elements from the stack
- **stack.size()**: Returns the number of elements in the stack



For more information, check out the Stanford Stack class [documentation](#)!

Queue

- Ordered (not indexed)
- First In, First Out (FIFO)
- Add to back, remove from front
- Important operations:
 - **queue.enqueue(value)**: Add an element to the back of the queue
 - **queue.dequeue()**: Remove an element from the front of the queue and return it
 - **queue.peek()**: Look at the element from the front of the queue, but don't remove it



The Stanford Queue Library

- **queue.enqueue(value)**: Add an element to the back of the queue
- **queue.dequeue()**: Remove an element from the front of the queue and return it
- **queue.peek()**: Look at the element from the front of the queue, but don't remove it
- **queue.isEmpty()**: Returns a boolean value, true if the queue is empty, false if it has at least one element
 - Note: a runtime error occurs if a dequeue() or peek() operation is attempted on an empty queue
- **queue.clear()**: Removes all elements from the queue
- **queue.size()**: Returns the number of elements in the queue

For more information, check out the Stanford Queue class [documentation!](#)



Tradeoffs with Stacks and Queues

What are some downsides?

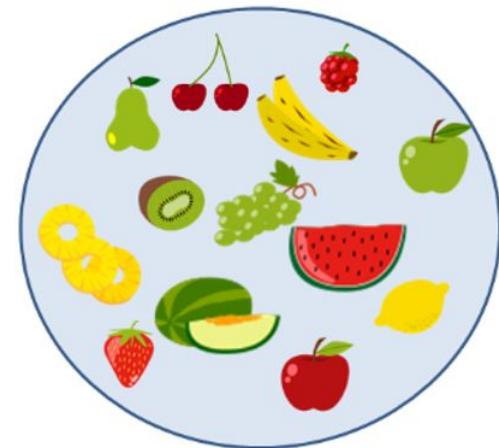
- No random access of elements
- Difficult to traverse - requires removal of elements
- No easy way to search

What are some benefits?

- Useful for many real world problems
- Easy to build such that access is guaranteed to be fast

Set

- Unordered
- No duplicates
 - All unique elements
- Faster at finding elements than ordered data structures
- Can be compared and combines using operands (`==`, `!=`, `+`, `*`, `-`)



The Stanford Set Library

- **set.add(value)**: Adds the value to the set, ignores if the set already contains the value
- **set.remove(value)**: Removes the value from the set, ignores if the value is not in the set
- **set.contains(value)**: Returns a boolean value, true if the set contains the value, false otherwise
- **set.isEmpty()**: Returns a boolean value, true if the set is empty, false otherwise
- **set.size()**: Returns the number of elements in the set

For more information, check out the Stanford Set class [documentation!](#)

Set Patterns and Pitfalls

- Use for each loops to iterate over a set

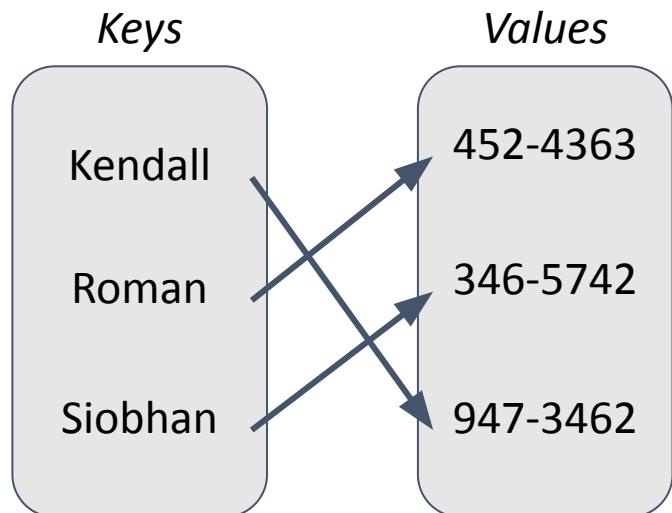
```
for(type currElem : set) {  
    // process elements one at a time  
}
```

- Cannot use anything that attempts to index into a set

```
for(int i=0; i < set.size(); i++) {  
    // does not work, no index!  
    cout << set[i];  
}
```

Map

- Unordered
- Collection of pairs
 - Called key/value pairs
- Use the key to quickly find the value
 - Keys must be unique
- Generalization of ordered data structure, where “indices” are not integers



The Stanford Map Library

- **map.clear()**: Removes all key/value pairs from the map
- **map.containsKey(key)**: Returns **true** if the map contains a value for the given key
- **map[key]**: Returns the value mapped to the given key
 - If **key** is not in the map, adds it with the default value (e.g., 0 or "")
- **map.get(key)**: Returns the value mapped to the given key
 - If **key** is not in the map, returns the default value for the value type, but does not add it to the map.
- **map.isEmpty()**: Returns **true** if the map contains no key/value pairs (size 0)
- **map.keys()**: Returns a **Vector** copy of all keys in the map
- **map[key] = value** and **map.put(key, value)**: Adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
- **map.remove(key)**: Removes any existing mapping for the given key (ignored if the key doesn't exist in the map)
- **map.size()**: Returns the number of key/value pairs in the map
- **map.toString()**: Returns a string such as "{a:90, d:60, c:70}"
- **map.values()**: Returns a **Vector** copy of all the values in the map

For more information, check out the Stanford Map class [documentation](#)!

Map Patterns and Pitfalls

- Use for each loops to iterate over a map

```
for(type currKey : map) {  
    // see map values using map[currKey]  
    // don't edit the map  
}  
  
for(type currKey : map.keys()) {  
    // see map values using map[currKey]  
    // can now edit the map!  
}
```

Map Patterns and Pitfalls

- Use for each loops to iterate over a map
- Auto-insert: a feature that can also cause bugs

`map[key]`: Returns the value mapped to the given key

- If key is not in the map, adds it with the default value (e.g., 0 or "")

Recap of ADTs

Ordered ADTs

Elements with indices

- Vectors (1D)
- Grids (2D)

Elements without indices

- Stacks (LIFO)
- Queues (FIFO)

Unordered ADTs

- Sets (unique elements)
- Maps (key, value pairs)

Nested ADTs

- We can “nest” ADTs (e.g. `Map<string, Set<string>>`)
- This allows us to represent more complex data
- Nested ADTs can be tricky to work with, especially because of reference and copies



Practice Problem (Vectors)

Write a function **removeBadPairs** that accepts a reference to a Vector of integers and removes any adjacent pair of integers in the list if the left element of the pair is larger than the right element of the pair. Every pair's left element is an even-numbered index in the list, and every pair's right element is an odd index in the list.

If the vector has an odd length, the last element is not part of a pair and is also considered "bad;" it should therefore be removed by your function.

If an empty vector is passed in, the vector should still be empty at the end of the call.

Practice Problem (Vectors) - Example

Suppose a variable called `vec` stores the following element values:

{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}

We can think of this list as a sequence of pairs:

{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}

The pairs 9-2, 8-5, 6-3, and 3-1 are "bad" because the left element is larger than the right one, so these pairs should be removed.

So the call of **`removeBadPairs(vec)`** would change the vector to store:

{3, 7, 5, 5, 4, 7}

Practice Problem (Vectors) - Solution 1

```
void removeBadPairs(Vector<int>& v) {  
    if (v.size() % 2 != 0) {  
        v.remove(v.size() - 1);  
    }  
    for (int i = v.size() - 1; i > 0; i--) {  
        if (i % 2 != 0 && v[i - 1] > v[i]) {  
            v.remove(i);  
            v.remove(i - 1);  
        }  
    }  
}
```

Practice Problem (Vectors) - Solution 2

```
void removeBadPairs(Vector<int>& v) {  
    if (v.size() % 2 != 0) {  
        v.remove(v.size() - 1);  
    }  
    for (int i = 0; i < v.size(); i += 2) {  
        if (v[i] > v[i + 1]) {  
            v.remove(i);  
            v.remove(i);  
            i -= 2;  
        }  
    }  
}
```

Practice Problem (Maps)

Write a function **byAge** that accepts three parameters:

1. a reference to a Map where each key is a person's name (a string) and the associated value is that person's age (an integer) (this map should not be modified in your function)
2. an integer for a minimum age
3. an integer for a max age

Your function should return a new Map with information about people with ages between the minimum and maximum, inclusive. For this result Map:

- each key is an integer age
- the value for a key is a string with the names of all people at that age, separated by "and" if there is more than one person of that age
- the order of names for a given age should be in alphabetical order, such as "Bob and Carl" rather than "Carl and Bob" (this is the order in which they naturally occur in the parameter map)
- include only ages between the min and max inclusive, where there is at least one person of that age in the parameter map
- if the map passed is empty, or if there are no people in the map between the min/max ages, return an empty map.

Practice Problem (Maps) - Example

If a **Map** named **ages** stores the following key:value pairs:

```
{"Allison":18, "Benson":48, "David":20, "Erik":20,  
"Galen":15, "Grace":25, "Helene":40, "Janette":18,  
"Jessica":35, "Marty":35, "Paul":28, "Sara":15,  
"Stuart":98, "Tyler":6, "Zack":20}
```

The call of **byAge(ages, 16, 25)** should return the following map:

```
{18:"Allison and Janette", 20:"David and Erik and Zack",  
25:"Grace"}
```

For the same map, the call of **byAge(ages, 20, 40)** should return the following map:

```
{20:"David and Erik and Zack", 25:"Grace", 28:"Paul",  
35:"Jessica and Marty", 40:"Helene"}
```

Practice Problem (Maps) - Solution 1

```
Map<int, string> byAge(Map<string, int>& ages, int min, int max) {
    Map<int, string> result;
    for (string name : ages) {
        int age = ages[name];
        if (min <= age && age <= max) {
            if (result.containsKey(age)) {
                string value = result.get(age);
                value += " and " + name;
                result.put(age, value);
            } else {
                result.put(age, name);
            }
        }
    }
    return result;
}
```

Practice Problem (Maps) - Solution 2

```
Map<int, string> byAge(Map<string, int>& ages, int min, int max) {  
    Map<int, string> result;  
    for (string name : ages) {  
        if (min <= ages[name] && ages[name] <= max) {  
            if (result.containsKey(ages[name])) {  
                result[ages[name]] += " and ";  
            }  
            result[ages[name]] += name;  
        }  
    }  
    return result;  
}
```

Algorithmic Analysis

The Big Idea: Big-O

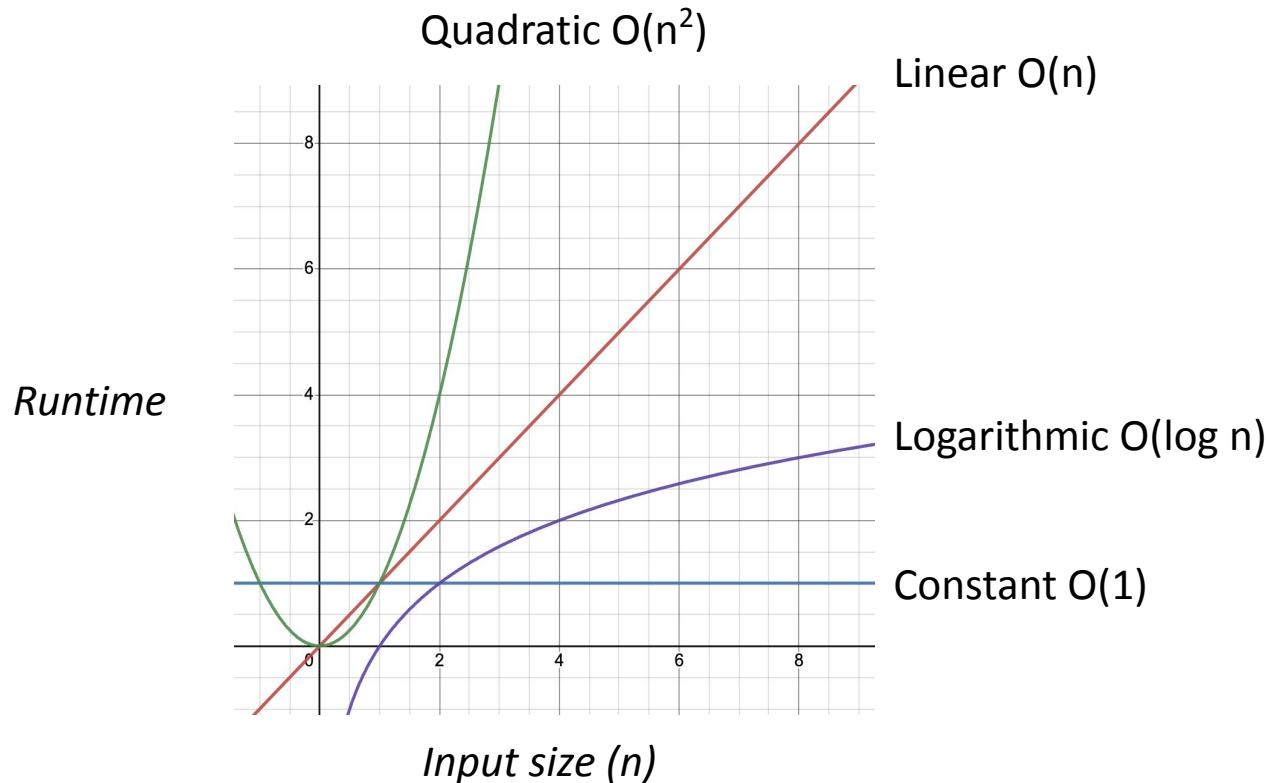
- General enough to compare across different computer systems
- Focuses on how the runtime will grow with the input size
 - It's all about growth rate
- This allows us to predict the runtime of future inputs

Different Big-O Time Complexities

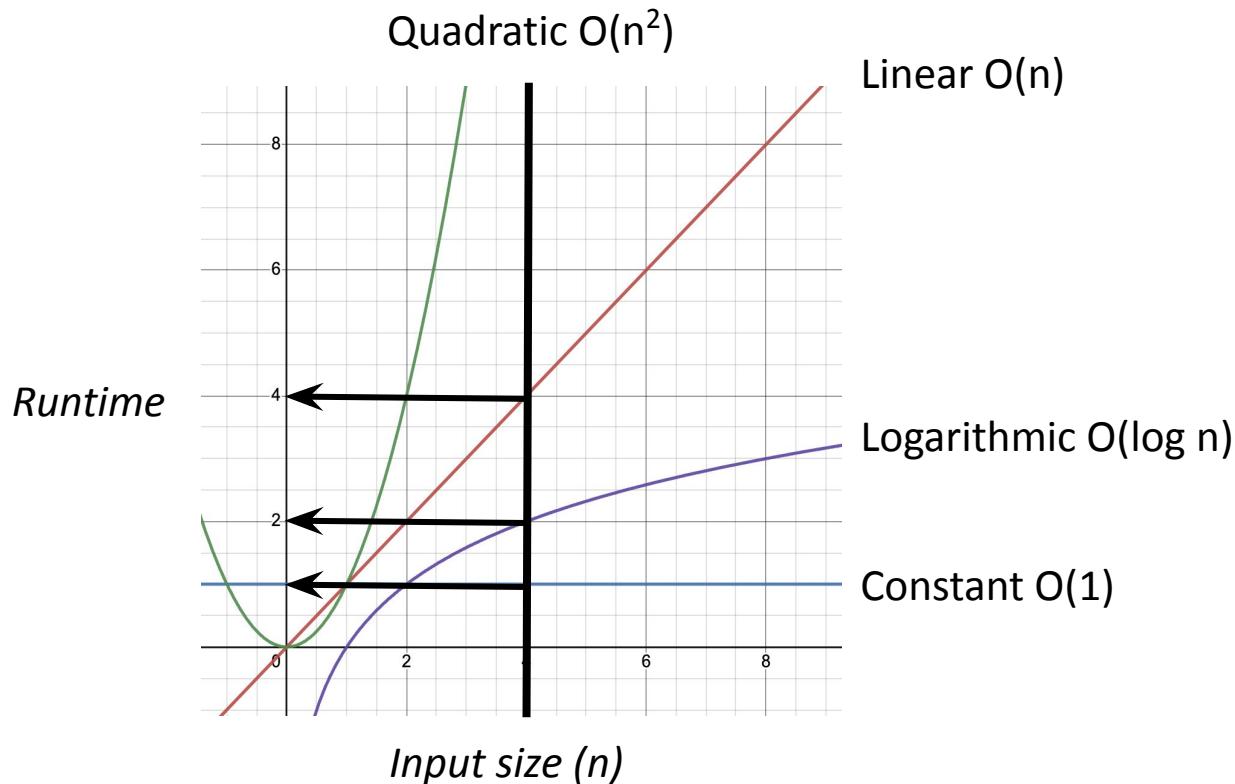
Constant	Logarithmic	Linear	$n \log n$	Quadratic	Polynomial	Exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $a > 1$

Faster  Slower

What Does this Graph Mean?



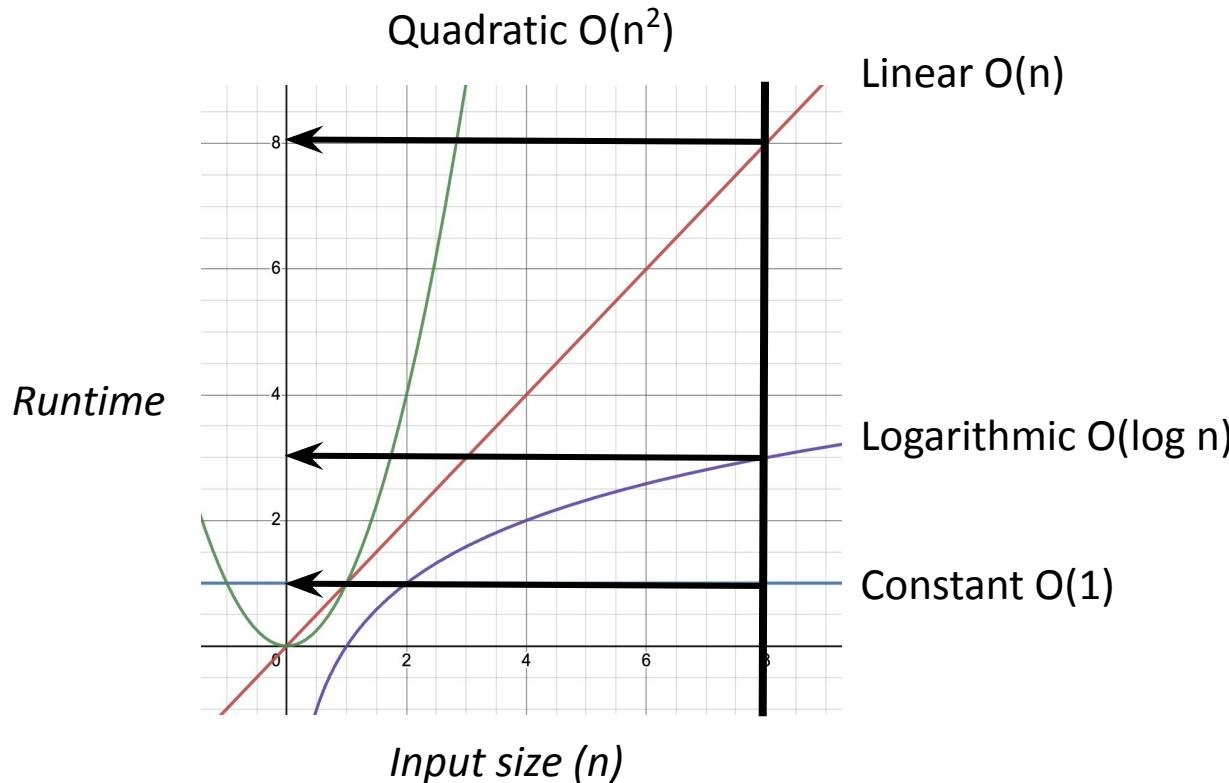
What Does this Graph Mean?



For an input of size 4

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 2 ms
- $O(n)$ takes 4 ms
- $O(n^2)$ takes 16ms

What Does this Graph Mean?



For an input of size 4

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 2 ms
- $O(n)$ takes 4 ms
- $O(n^2)$ takes 16 ms

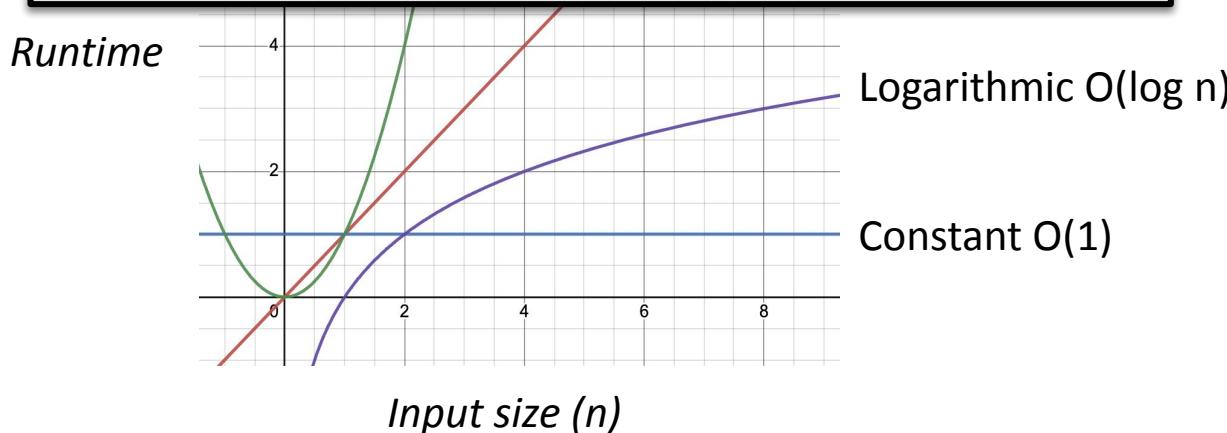
For an input of size 8

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 3 ms
- $O(n)$ takes 8 ms
- $O(n^2)$ takes 64 ms

What Does this Graph Mean?

If an algorithm is $O(1)$, it takes the same amount of time to run no matter the input size!

.add() takes the same amount of time for a Vector of size 1 and 100,000.



For an input of size 4

- **$O(1)$ takes 1 ms**
- $O(\log n)$ takes 2 ms
- $O(n)$ takes 4 ms
- $O(n^2)$ takes 16 ms

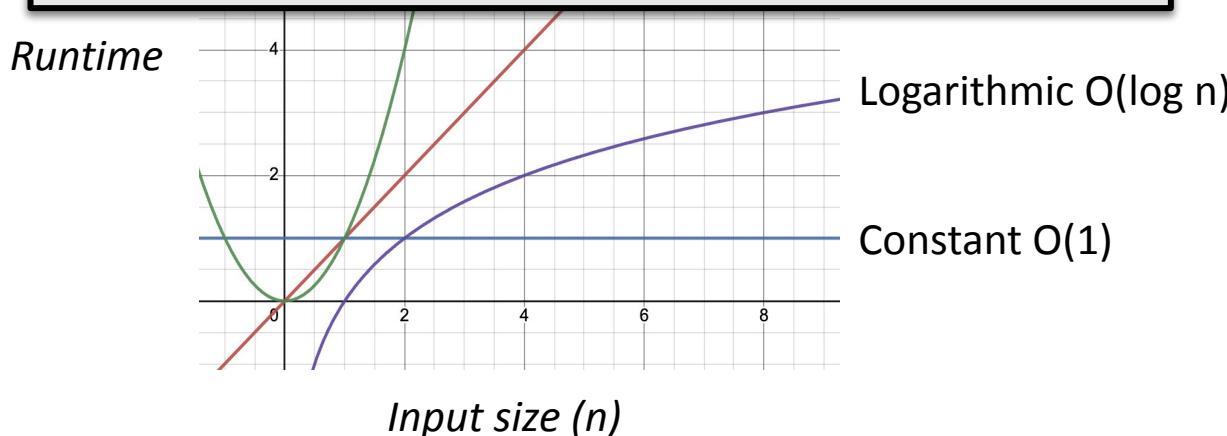
For an input of size 8

- **$O(1)$ takes 1 ms**
- $O(\log n)$ takes 3 ms
- $O(n)$ takes 8 ms
- $O(n^2)$ takes 64 ms

What Does this Graph Mean?

If an algorithm is $O(\log n)$, it often involves splitting n in half at each step.

Binary search! This is pretty darn efficient.



For an input of size 4

- $O(1)$ takes 1 ms
- **$O(\log n)$ takes 2 ms**
- $O(n)$ takes 4 ms
- $O(n^2)$ takes 16 ms

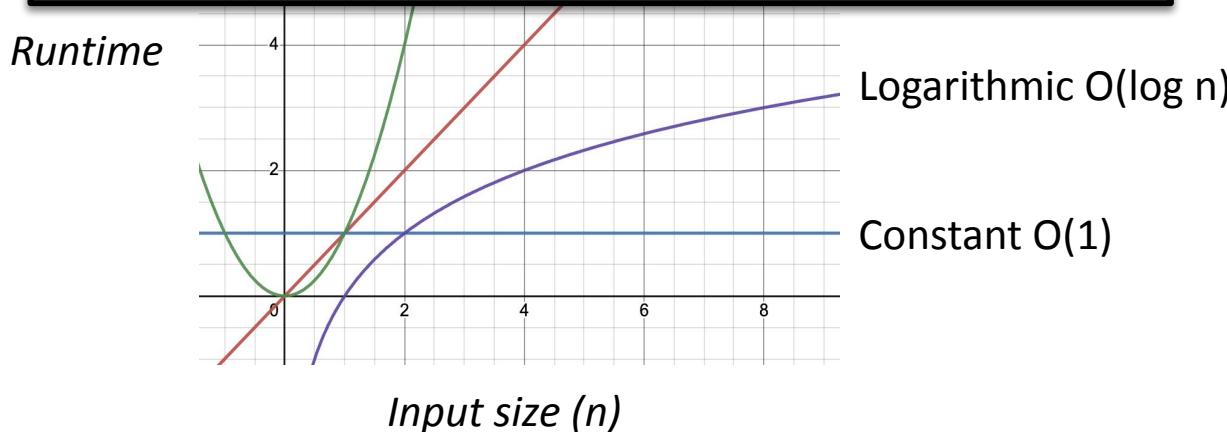
For an input of size 8

- $O(1)$ takes 1 ms
- **$O(\log n)$ takes 3 ms**
- $O(n)$ takes 8 ms
- $O(n^2)$ takes 64 ms

What Does this Graph Mean?

If an algorithm is $O(n)$, the runtime scales linearly with the input size. Runtime grows the same amount that input size grows.

A for loop over a Vector of size n is $O(n)$.



For an input of size 4

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 2 ms
- **$O(n)$ takes 4 ms**
- $O(n^2)$ takes 16 ms

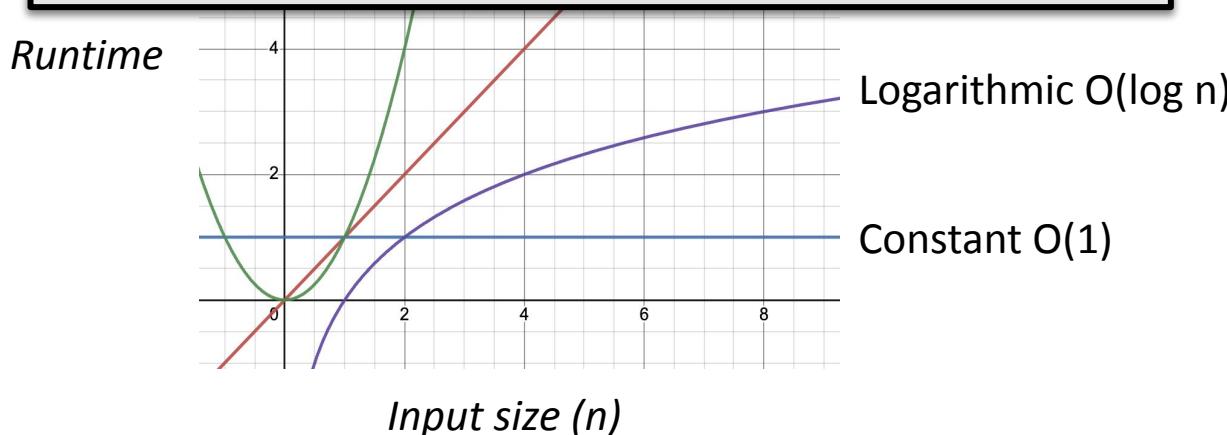
For an input of size 8

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 3 ms
- **$O(n)$ takes 8 ms**
- $O(n^2)$ takes 64 ms

What Does this Graph Mean?

If an algorithm is $O(n^2)$, it might involve an $O(n)$ operation within an $O(n)$ operation.

A for loop over a Grid of size $n \times n$ is $O(n^2)$.



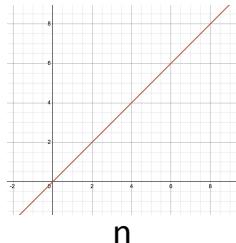
For an input of size 4

- $O(1)$ takes 1 ms
- $O(\log n)$ takes 2 ms
- $O(n)$ takes 4 ms
- **$O(n^2)$ takes 16 ms**

For an input of size 8

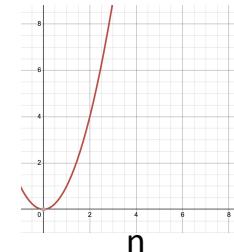
- $O(1)$ takes 1 ms
- $O(\log n)$ takes 3 ms
- $O(n)$ takes 8 ms
- **$O(n^2)$ takes 64 ms**

runtime

 $O(n)$

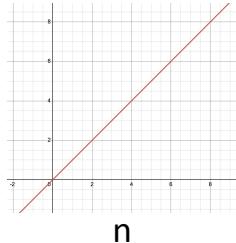
```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

runtime

 $O(n^2)$

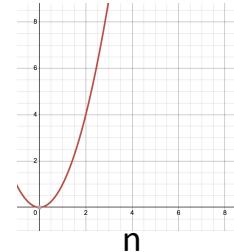
```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

runtime

 $O(n)$

```
int vectorMax(Vector<int> &v) {
    int currentMax = v[0];
    int n = v.size();
    for (int i = 1; i < n; i++) {
        if (currentMax < v[i]) {
            currentMax = v[i];
        }
    }
    return currentMax;
}
```

runtime

 $O(n^2)$

```
int vectorMax(Vector<int> &v) {
    for (int num: v) {
        bool seenLarger = false;
        for (int compareNum: v) {
            if (compareNum > num) {
                seenLarger = true;
            }
        }
        if (!seenLarger) {
            return num;
        }
    }
    return -1;
}
```

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}  
  
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}  
  
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

For a Vector of size n , we loop n times.

Within this loop, we call `printWord`, which prints our word 10 times.

This is $n * 10$ operations, which simplifies to $O(n)$.

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}  
  
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

For a Vector of size n , we loop n times.

Within this loop, we call `printWord`, which prints our word 10 times.

This is $n * 10$ operations, which simplifies to $O(n)$.

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}  
  
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

For a Vector of size n , we loop n times.

Within this loop, we call `printWord`, which prints our word 10 times.

This is $n * 10$ operations, which simplifies to $O(n)$.

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}
```

*If printWords takes **30s** to run for a Vector with **1 million** elements, how long will it take for a Vector with **4 million** elements?*

```
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

Practice problem: Runtime of printWords

```
void printWord(string word) {  
    for (int i = 0; i < 10; i++) {  
        cout << word << endl;  
    }  
}
```

```
int printWords(Vector<string> words) {  
    for (string word: words) {  
        printWord(word);  
    }  
}
```

120s.
For a function with $O(n)$ runtime, runtime scales linearly with input size.

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {  
    Vector<int> result;  
    int n = vec.size();  
    for (int i = 0; i < n; i++) {  
        int elem = vec.remove(0);  
        result.insert(0, elem);  
    }  
    return result;  
}
```

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {  
    Vector<int> result;  
    int n = vec.size();  
    for (int i = 0; i < n; i++) {  
        int elem = vec.remove(0);  
        result.insert(0, elem);  
    }  
    return result;  
}
```

For a Vector of size n , we loop n times.

Within this loop, we remove and insert, both of which are $O(n)$.

This is $n * O(n)$ operations, which simplifies to $O(n^2)$.

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {
    Vector<int> result;
    int n = vec.size();
    for (int i = 0; i < n; i++) {
        int elem = vec.remove(0);
        result.insert(0, elem);
    }
    return result;
}
```

For a Vector of size n , we loop n times.

Within this loop, we remove and insert, both of which are $O(n)$.

Vector<T>		
<code>v.add(val) or v += val</code>	appends value to end of vector	$O(1)$ *
<code>v.clear()</code>	removes all elements	$O(1)$
<code>v.get(i) or v[i]</code>	returns value at given index	$O(1)$
<code>v.insert(i, val)</code>	inserts at given index, shifting subsequent values right	$O(N)$
<code>v.isEmpty()</code>	returns true if there are no elements	$O(1)$
<code>v.remove(i)</code>	removes value at given index, shifting subsequent values left	$O(N)$
<code>v.set(i, val) or v[i] = val</code>	replaces value at given index	$O(1)$
<code>v.size()</code>	returns number of elements	$O(1)$
<code>v.subList(start, length)</code>	returns new vector containing subrange of elements	$O(N)$

operations,
 $> O(n^2)$.

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {  
    Vector<int> result;  
    int n = vec.size();  
    for (int i = 0; i < n; i++) {  
        int elem = vec.remove(0);  
        result.insert(0, elem);  
    }  
    return result;  
}
```

For a Vector of size n , we loop n times.

Within this loop, we remove and insert, both of which are $O(n)$.

This is $n * O(n)$ operations, which simplifies to $O(n^2)$.

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {  
    Vector<int> result;  
    int n = vec.size();  
    for (int i = 0; i < n; i++) {  
        int elem = vec.remove(0);  
        result.insert(0, elem);  
    }  
    return result;  
}
```

If reverseVec takes 5ms to run for a Vector with 100,000 elements, how long will it take for a Vector with 200,000 elements?

Practice problem: Runtime of reverseVec

```
Vector<int> reverseVec(Vector<int> vec) {  
    Vector<int> result;  
    int n = vec.size();  
    for (int i = 0; i < n; i++) {  
        int elem = vec.remove(0);  
        result.insert(0, elem);  
    }  
    return result;  
}
```

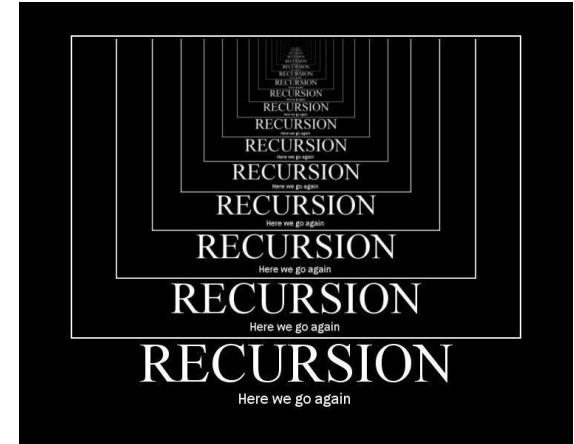
20ms.

*For a function with $O(n^2)$ runtime,
doubling input size quadruples runtime.*

Recursion

What is recursion?

- A problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**.
- The function calls itself and every time, the problem becomes a little smaller



Two main components

- Base case
 - The simplest version of your problem that all other cases reduce to
 - An occurrence that can be answered directly
- Recursive case
 - More complex version of the problem that cannot be directly answered
 - Break down the task into smaller occurrences
 - Take the “recursive leap of faith” and trust the smaller tasks will solve the problem for you!

Three “Musts” of Recursion

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

Three “Musts” of Recursion

```
int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

Recursive vs Iterative Methods

```
int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```

$n = 5$, time = 5.823 ms

$n = 100,000$, time = 8.703 ms

$n = 1,000,000$, “segmentation fault”

```
int factorialIterative (int n) {  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result = result * i;  
    }  
    return result;  
}
```

$n = 5$, time = 5.485 ms

$n = 100,000$, time = 5.589 ms

$n = 1,000,000$, time = 7.501 ms

Iteration + Recursion

- It's completely reasonable to mix iteration and recursion in the same function.
- Recursion doesn't mean "the absence of iteration." It just means "solving a problem by solving smaller copies of that same problem."
- Iteration and recursion can be very powerful in combination!

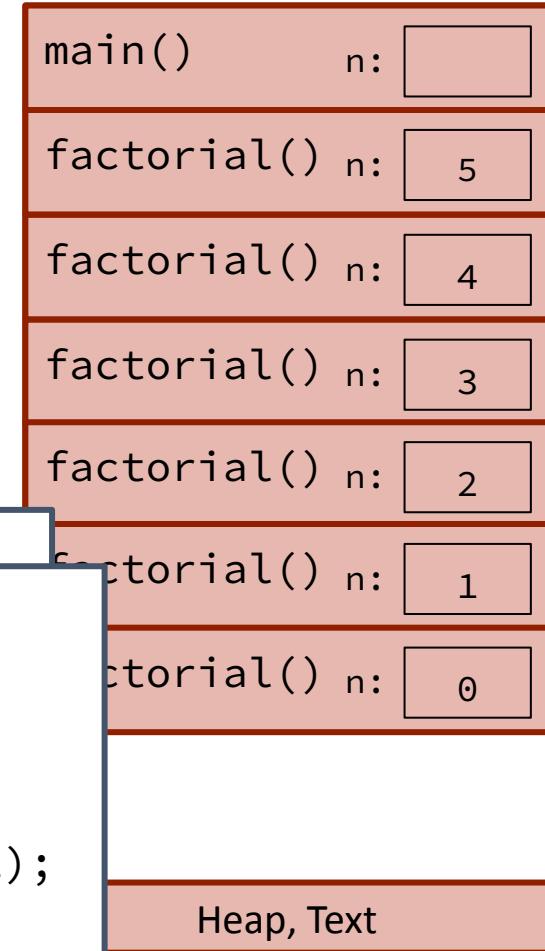
Big-O of Recursive Functions

Depends on:

- Big-O of each execution of the function
- Number of recursive calls

Factorial, Revisited

```
int main () {  
    int factorial (int n) {  
        int factorial (int n) {  
            int factorial (int n) {  
                int factorial (int n) {  
                    int factorial (int n) {  
                        int factorial (int n) {  
                            int factorial (int n) {  
                                int factorial (int n) {  
                                    int factorial (int n) {  
                                        if (n == 0) {  
                                            return 1;  
                                        } else {  
                                            return n * factorial(n-1);  
                                        }  
                                    }  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```



Factorial, Revisited

```
int main () {
```

```
    int f
```

```
    int
```

*What's the runtime of one
call to factorial?*

```
    int factorial (int n) {
```

```
        int factorial (int n) {
```

```
            if (n == 0) {
```

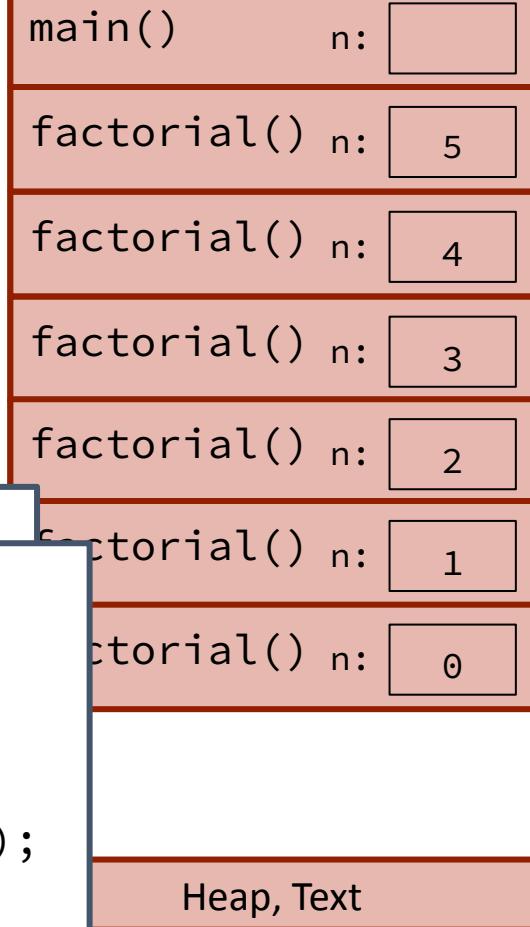
```
                return 1;
```

```
            } else {
```

```
                return n * factorial(n-1);
```

```
        }
```

```
}
```



Factorial, Revisited

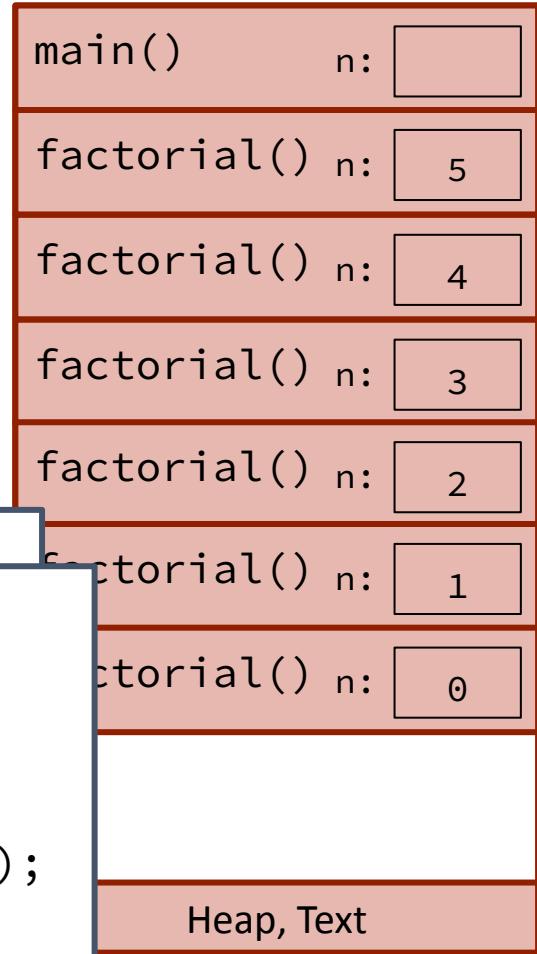
```
int m = 5;
```

```
int n = 5;
```

```
int
```

$O(1)$
*All of these operations
(comparison, multiplication) are
constant time.*

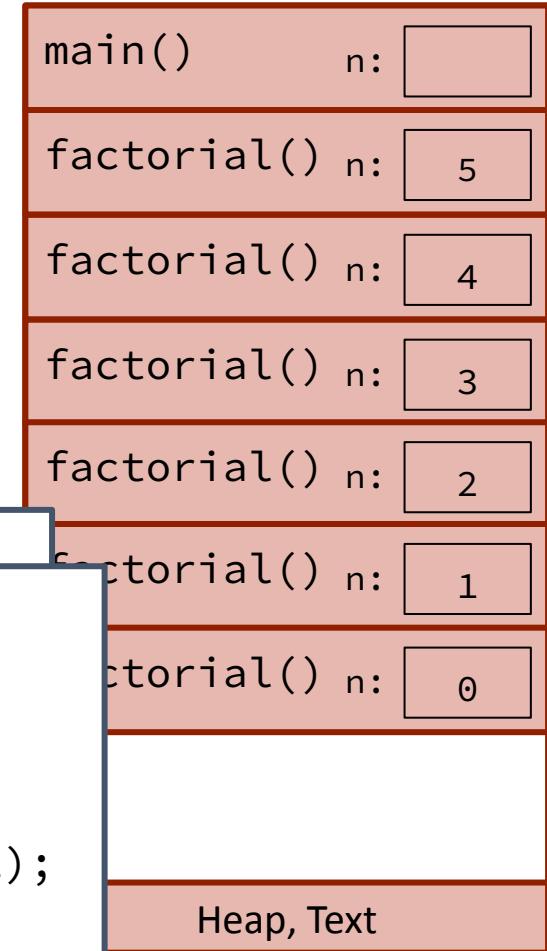
```
int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```



Factorial, Revisited

```
int main () {  
    int f;  
    int n;  
    ...  
    int factorial (int n) {  
        ...  
        if (n == 0) {  
            return 1;  
        } else {  
            return n * factorial(n-1);  
        }  
    }  
}
```

How many times does factorial get called?



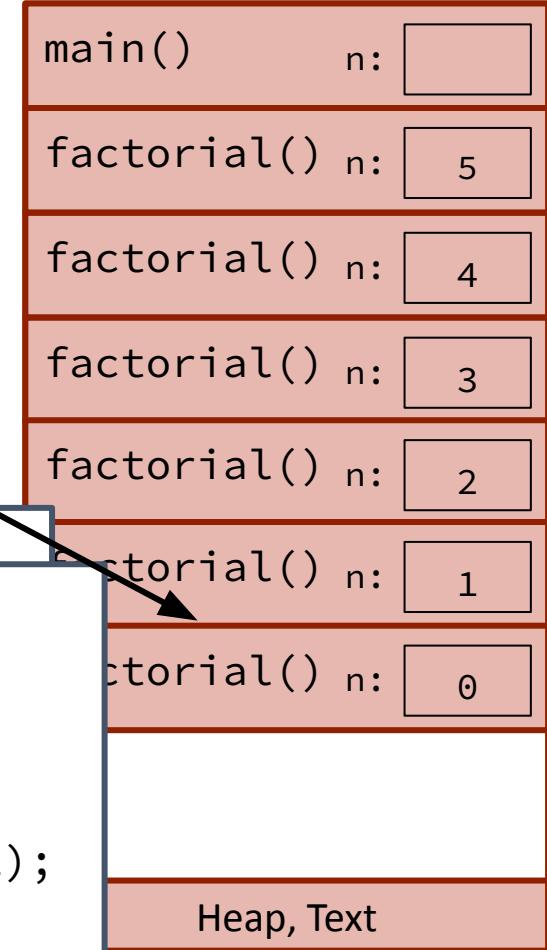
Factorial, Revisited

```
int main() {  
    int n;  
    int result;  
    cin >> n >> result;  
    cout << result << endl;  
    return 0;  
}
```

$O(n)$ times

We decrease n by 1 each recursive call, so it takes n recursive calls to get down to the base case.

```
int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```



Factorial, Revisited

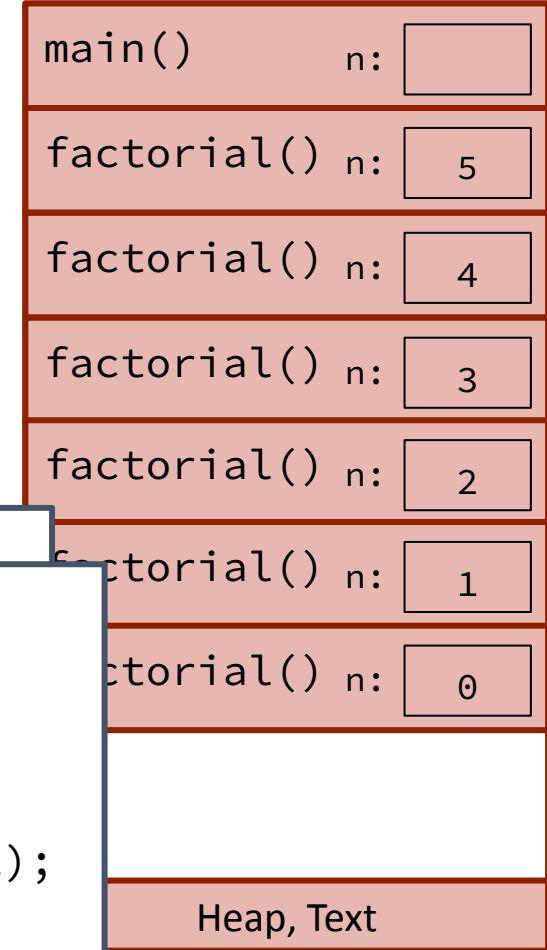
```
int m...
```

```
int n...
```

```
int
```

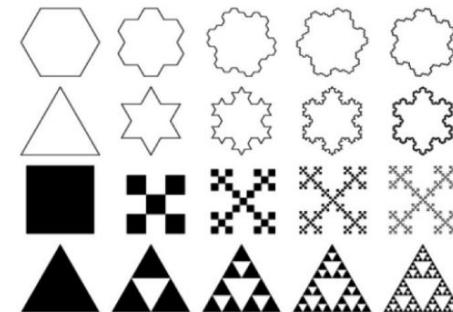
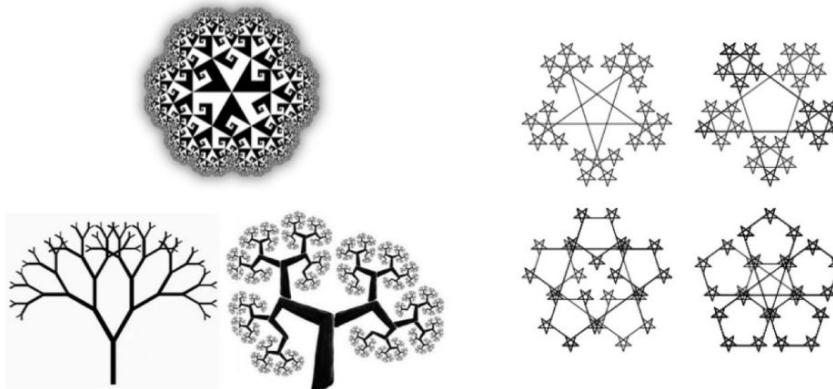
What's the runtime of factorial?
 $O(1) * O(n) = O(n)$

```
int factorial (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```



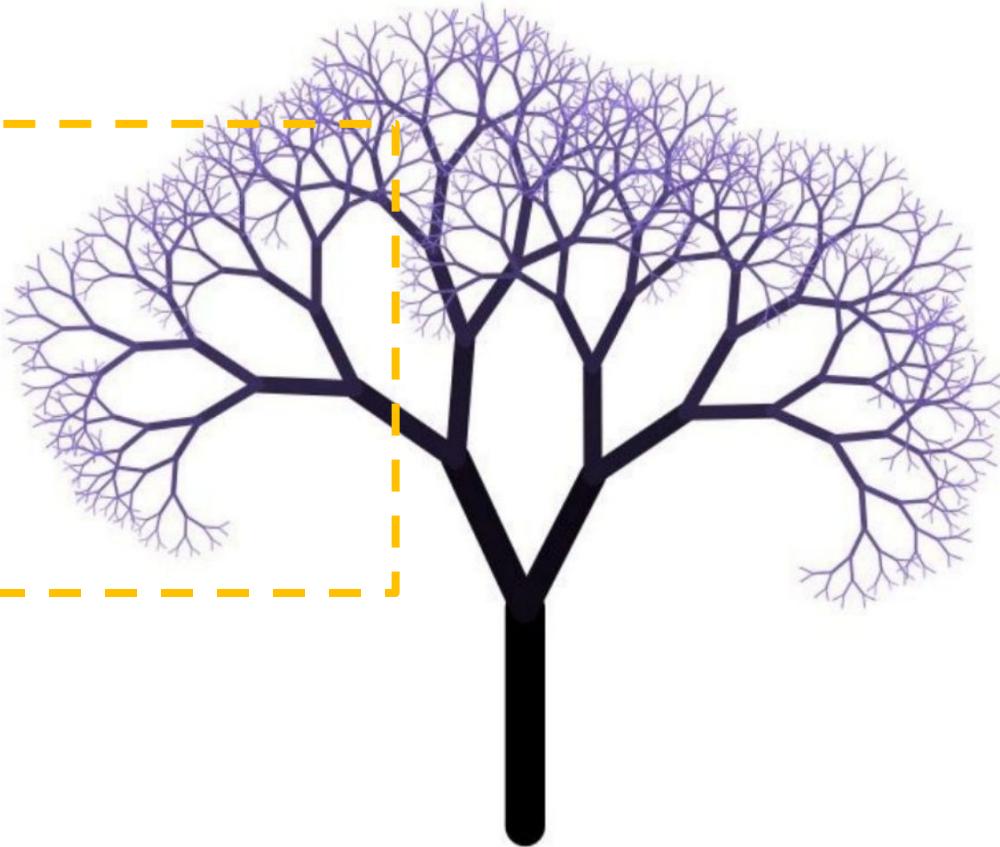
Fractal

- Any repeated, graphical pattern
- Composed of repeated instances of the same shape or pattern, arranged in a structured way



What differentiates the smaller tree from the bigger one?

1. It's at a different **position**
2. It has a different **size**
3. It has a different **orientation**
4. It has a different **order**



Fractals and self-similar structures are often defined in terms of some parameter called the **order**, which indicates the complexity of the overall structure.

Why do we use recursion?

- Elegant
 - Some problems have beautiful, concise recursive solutions
 - Ex: Towers of Hanoi
- Efficient
 - Recursive solutions can have faster runtimes
 - Ex: Binary Search

Binary Search

- We have a sorted Vector of integers and want to find some target
- **Binary search** over some range of sorted elements:
 - Choose element in the middle of the range
 - If this element is our target, success!
 - If element is less than our target, do **binary search** to the right
 - If element is greater than our target, do **binary search** to the left

-1	2	6	12	37	41	72	88	90
0	1	2	3	4	5	6	7	8

Runtime of Binary Search

- We're searching through N elements and eliminating half until only one element remains:
 - 1000 pages \rightarrow 500 pages \rightarrow 250 pages $\rightarrow \dots \rightarrow$ 1 page
 - $N \rightarrow N/2 \rightarrow N/4 \rightarrow \dots \rightarrow 2 \rightarrow 1$
- How many steps does it take to get from N down to 1?
 - Let's think of it this way: $1 \rightarrow 2 \rightarrow \dots \rightarrow N/4 \rightarrow N/2 \rightarrow N$
 - Number of times we multiply by 2 to get to N

Runtime of Binary Search

- We're searching through N elements and eliminating half until only one element remains:

If x is the number of times we multiply by 2 to get N ...

$$2^x = N$$

- $x = \log_2 N$

- Let's think of it this way: $1 \rightarrow 2 \rightarrow \dots \rightarrow N/4 \rightarrow N/2 \rightarrow N$
- Number of times we multiply by 2 to get to N

Runtime of Binary Search

- Faster than $O(n)$, since it's faster than a linear search
- Slower than $O(1)$, since it takes longer for larger input sizes
- It's $O(\log n)$

Constant	Logarithmic	Linear	$n \log n$	Quadratic	Polynomial	Exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $k \geq 1$	$O(a^n)$ $a > 1$

Approaching recursive problems

- Look for self-similarity.
- Try out an example.
 - Work through a simple example and then increase the complexity.
 - Think about what information needs to be “stored” at each step in the recursive case
- Ask yourself:
 - What is the base case? (What is the simplest case?)
 - What is the recursive case? (What pattern of self-similarity do you see?)

Practice Problem #1

Write a recursive function named **replaceAll** that accepts three parameters:

1. a string **s**
2. a char **from**
3. a char **to**

This function returns a new string that is the same as **s** but with any occurrences of **from** changed to **to**.

Your function is case-sensitive; if the character from is, for example, a lowercase 'f', your function should not replace uppercase 'F' characters. In other words, you should not need to write code to handle case issues in this problem.

Additional stipulations:

- Do not use any loops; you must use recursion.
- Do not declare any global variables.
- Do not call any of the following string functions: `find`, `rfind`, `index0f`, `contains`, `replace`, `split`. (The point of this problem is to solve it recursively; do not use a library function to get around recursion.)
- Do not use any auxiliary data structures like `Vector`, `Map`, `Set`, `array`, etc.

Practice Problem #1 - Example

The call of **replaceAll("crazy raccoons", 'c', 'k')** should return "krazy rakkoons".

The call of **replaceAll("BANANA", 'A', 'O')** should return "BONONO".

Practice Problem #1 - Solution 1

```
string replaceAll(string s, char from, char to) {  
    if (s.empty()) {  
        return s;  
    } else {  
        char first = s[0];  
        string rest = s.substr(1);  
        if (first == from) {  
            first = to;  
        }  
        return first + replaceAll(rest, from, to);  
    }  
}
```

Practice Problem #2 - Solution 2

```
string replaceAll(string s, char from, char to) {  
    if (s == "") {  
        return s;  
    } else if (s[0] == from) {  
        return to + replaceAll(s.substr(1), from, to);  
    } else {  
        return s[0] + replaceAll(s.substr(1), from,  
        to);  
    }  
}
```

Practice Problem #2

Write a recursive function named **removeEvens** that accepts a Stack of integers and an integer K as parameters and removes the first K even numbers starting from the top of the stack, leaving all other elements present in the same relative order. You should also return the number of elements that were removed.

- If the stack does not contain K even values, remove as many evens as you can and return the number that were removed.
- If the stack does not contain any elements with even values, the stack should not be modified and you should return 0

Additional stipulations:

- Do not use any loops; you must use recursion.
- Do not use any auxiliary data structures like Stack, Queue, Vector, Map, Set, array, string, etc. This includes making a "backup" of the stack passed as a parameter, or passing it by value to copy it.
- Do not declare any global variables.

Practice Problem #2 - Example

Consider a stack named `stack` that contains the following elements:

(bottom) {8, 1, 4, 9, 5, 2, 6, 7, 12} (top)

The call of **removeEvens(stack, 3)** should remove the 3 even element values closest to the top of the stack, which are 12, 6, and 2. The stack's contents after the call should be {8, 1, 4, 9, 5, 7} and the function should return 3.

If the call were **removeEvens(stack, 7)**, there are not 7 even values in the stack, but you should remove all five even values, 12, 6, 2, 4, and 8, leaving the stack storing {1, 9, 5, 7}. You would return 5.

Practice Problem #2 - Solution

```
int removeEvens(Stack<int>& stack, int k) {  
    if (stack.isEmpty() || k <= 0) {  
        return 0;  
    } else {  
        int top = stack.pop();  
        if (top % 2 == 0) {  
            // even value; remove from stack, count toward 'k'  
            return 1 + removeEvens(stack, k - 1);  
        } else {  
            // odd value; keep in stack, recur on rest of stack for evens  
            int result = removeEvens(stack, k);  
            stack.push(top);  
            return result;  
        }  
    }  
}
```

Good luck on the midterm!
You can do this!!