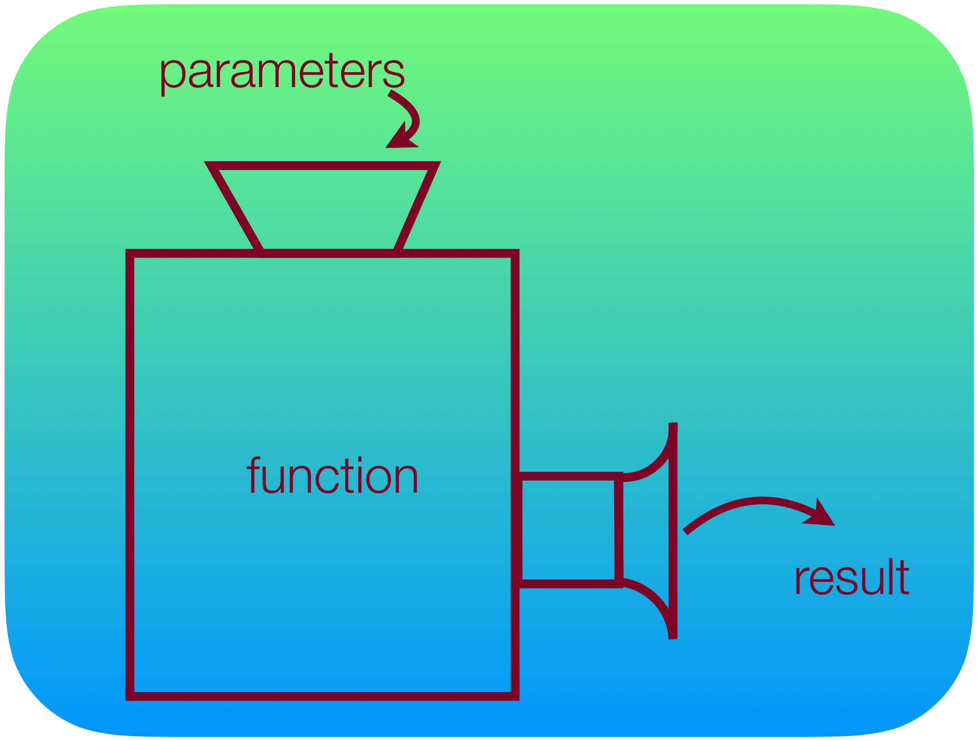
CS 106X
Lecture 2: C++ Functions / Computational Complexity

Wednesday, January 11, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapters 2-3, Section 10.2



Today's Topics

- Logistics:
  - Signing up for section
  - Qt Creator installation help on Thursday
- Homework 1: Fauxtoshop!
  - Due Friday, January 20th, at Noon
  - If you are having trouble starting Fauxtoshop, you can watch last quarter's "YEAH Hours": https://youtu.be/-tuaVHYhH3U
- Functions
  - Some review — functions are very similar to Java functions!
  - Value semantics, Reference semantics
- Introduction to Computational Complexity and "Big O"
- Reading Assignment: Chapters 2 and 3, Section 10.2

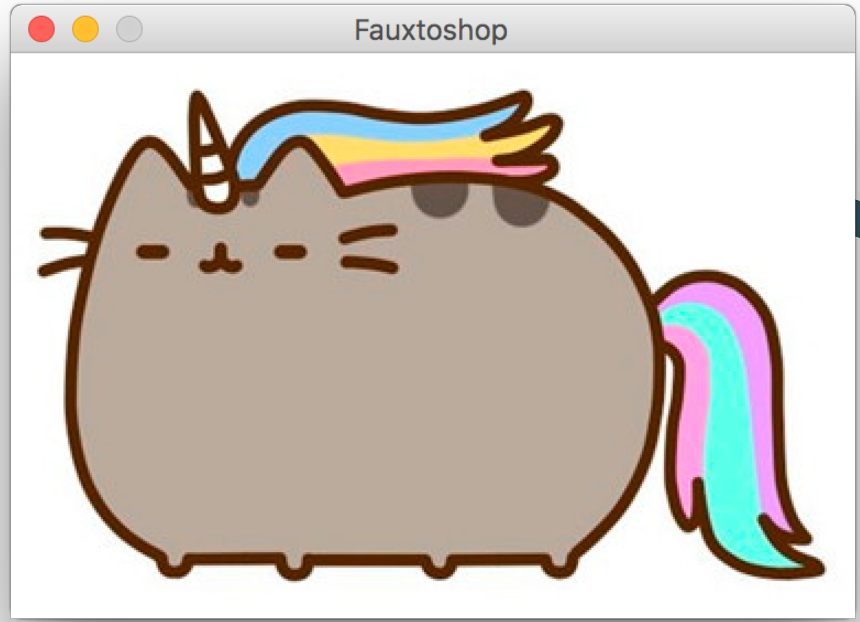
Logistics

- Signing up for section: you must put your available times by Sunday January 15th at 5pm (opens Thursday at 5pm).
  - Go to cs198.stanford.edu to sign up.

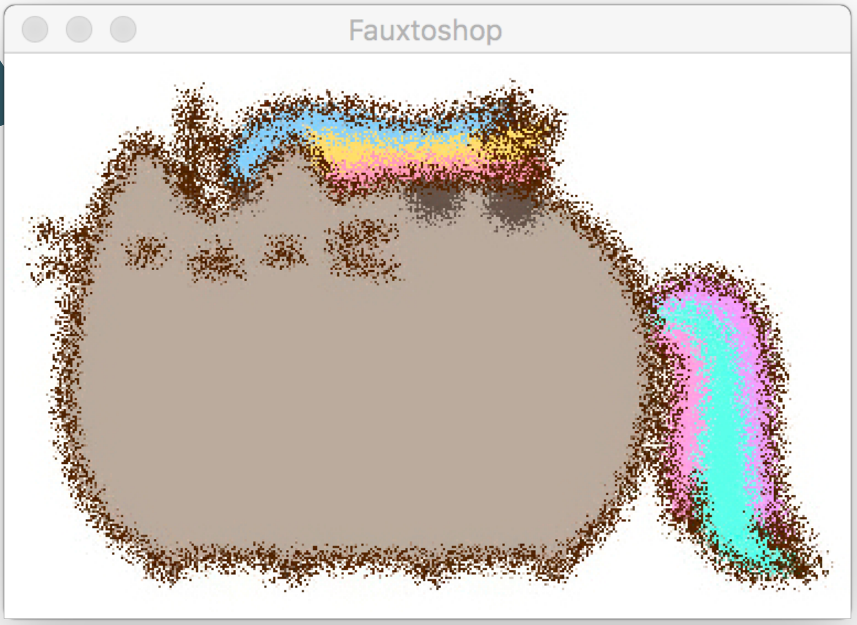
- Qt Creator installation help: Thursday at 8pm, in Tressider (eating area). Please attempt to install Qt Creator before you arrive (see the course website for details).

- Remember, Assignment 0 is due Friday at Noon

Assignment 1: Fauxtoshop!

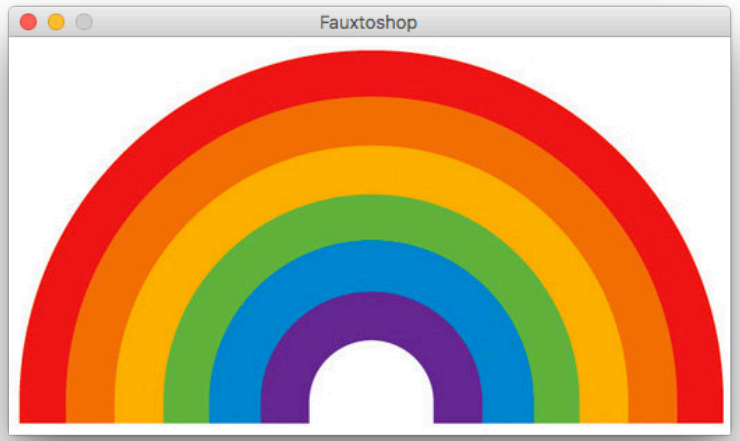


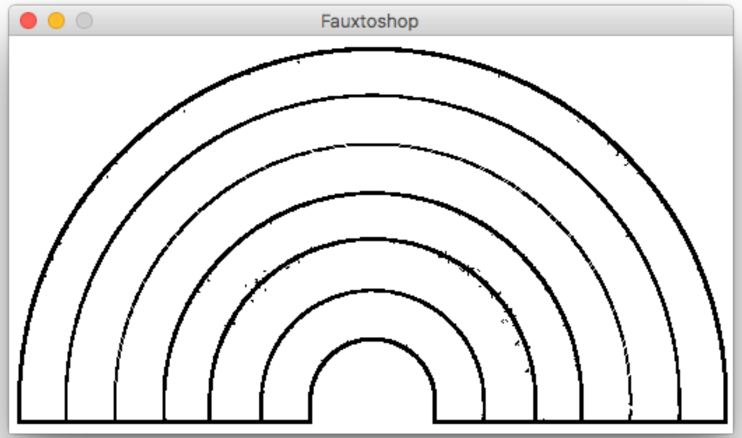
Click for Intro Video!



Scatter!

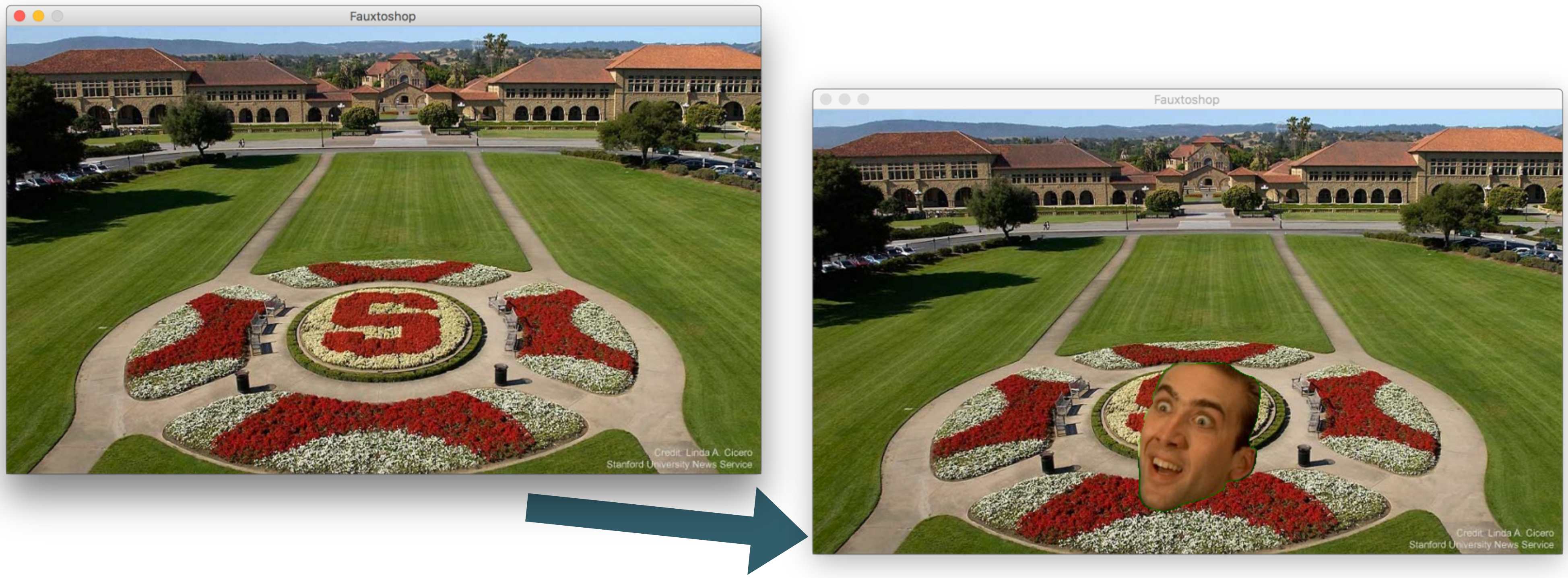
Assignment 1: Fauxtoshop!





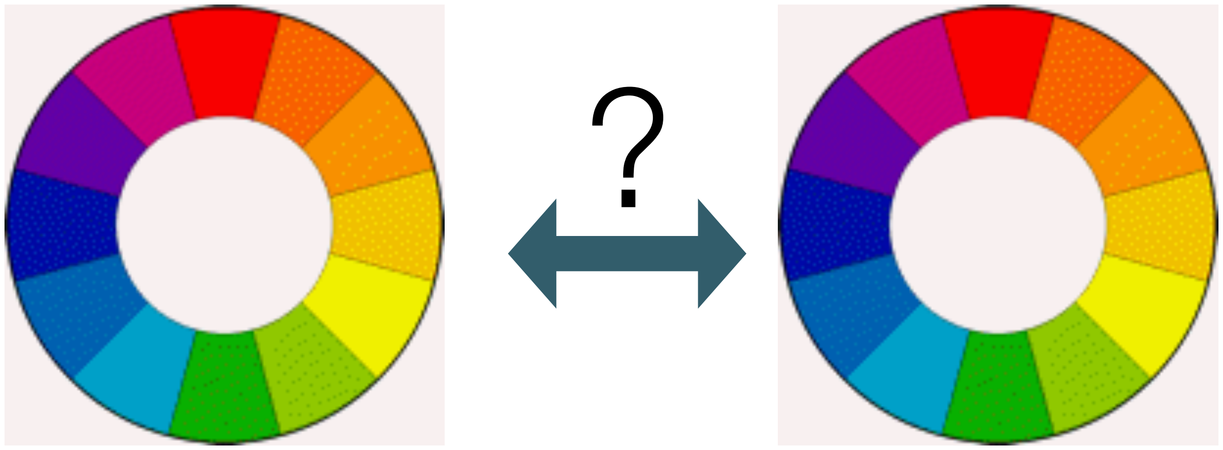
Edge Detection!

Assignment 1: Fauxtoshop!



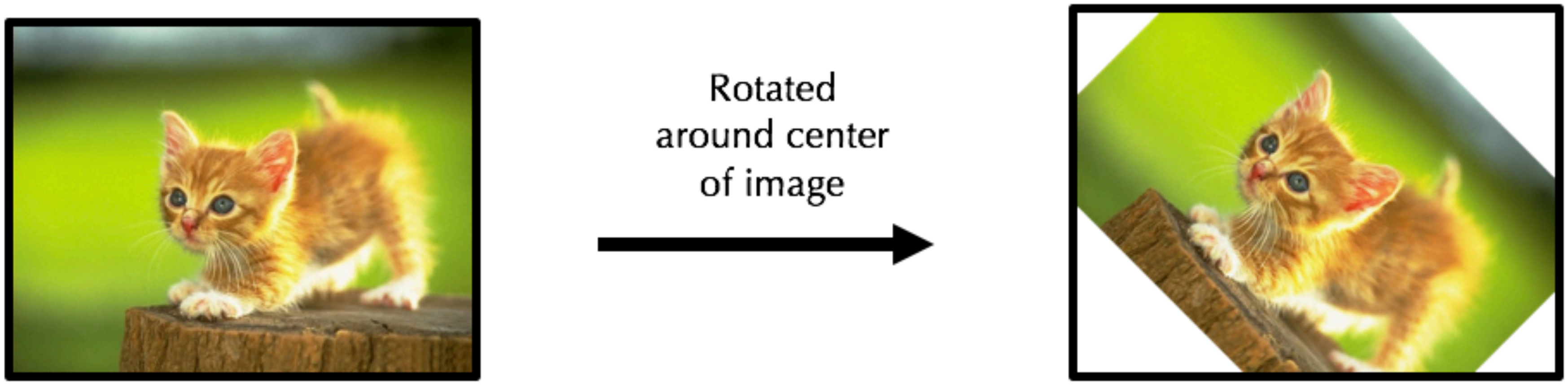
Green Screen Merging!

Assignment 1: Fauxtoshop!



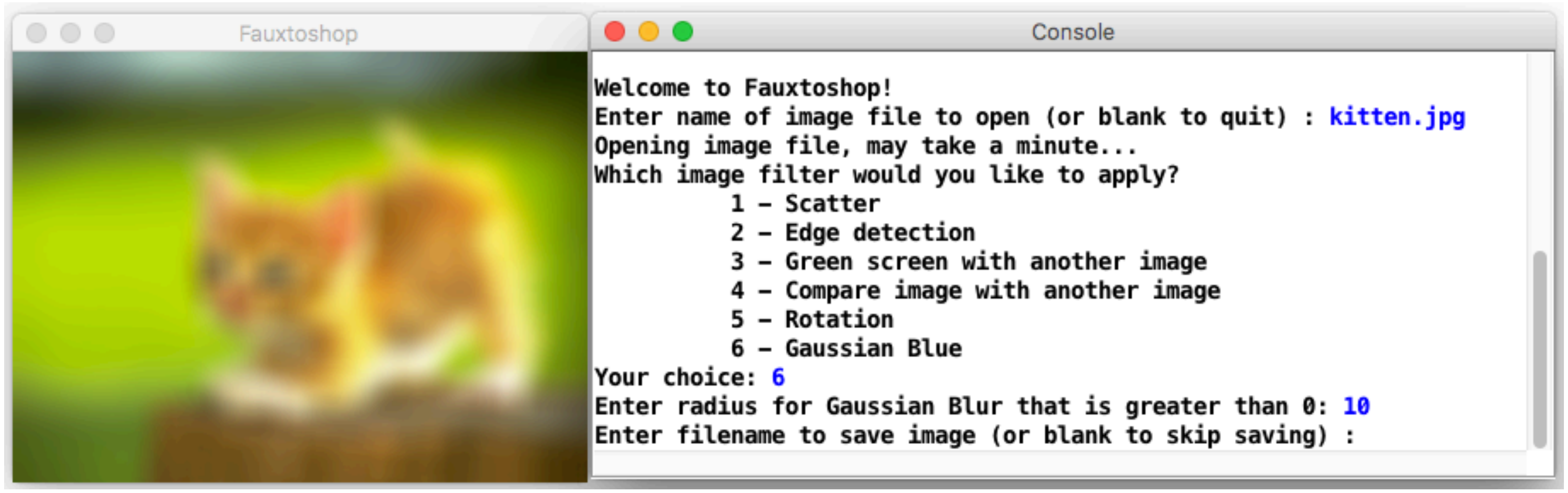
Compare Images!

Assignment 1: Fauxtoshop!



Rotate!

Assignment 1: Fauxtoshop!



```
Welcome to Fauxtoshop!
Enter name of image file to open (or blank to quit) : kitten.jpg
Opening image file, may take a minute...
Which image filter would you like to apply?
        1 - Scatter
        2 - Edge detection
        3 - Green screen with another image
        4 - Compare image with another image
        5 - Rotation
        6 - Gaussian Blue
Your choice: 6
Enter radius for Gaussian Blur that is greater than 0: 10
Enter filename to save image (or blank to skip saving) :
```

Gaussian Blur!

Fauxtoshop

- The program you write will utilize:
  - Functions
  - Constants
  - Loops
  - I/O (cout, getLine(), getInteger())
  - Reference semantics, Value semantics
  - Strings
  - Logic
  - Nicholas Cage
- We will discuss all of the above before the project is due
- **Get started early**! (Idea: finish Scatter by Friday!)
- Due: 12pm (Noon) on Friday, January 20th

Defining Functions (2.3)

A C++ **function** is like a Java **method**. Similar declaration syntax but without the public or private keyword in front.

return type *parameters*

```
type functionName(type name, type name, ..., type name) {
    statement;
    statement;
    ...
    statement;
    return expression; // if return type is not void
}
```

arguments (called in the ***same order*** *as the parameters)*

Calling a function:

```
functionName(value, value, ..., value);
```

Function Return Types

A C++ function must have a return type, which can be any time (including user-defined types, which we will cover later).

```
double square(double x); // returns a double
Vector<int> matrixMath(int x, int y); // returns a Vector
                                  // probably not a good
                                  // idea! (covered later)
string lowercase(string s); // returns a string (maybe
                            // not a good idea...
void printResult(Vector<int> &v); // returns nothing!
```

A C++ function can only return a single type, and if you want to return multiple "things," you have to do it differently (unlike in languages such as Python). We will cover this later, as well.

Function Example, brought to you by

(bus drivers *hate* them!)

```
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

Function Example: Output

```
99 bottles of Coke on the wall.
99 bottles of Coke.
Take one down, pass it around, 98 bottles of Coke on the wall.

98 bottles of Coke on the wall.
98 bottles of Coke.
Take one down, pass it around, 97 bottles of Coke on the wall.

97 bottles of Coke on the wall.
97 bottles of Coke.
Take one down, pass it around, 96 bottles of Coke on the wall.

...

3 bottles of Coke on the wall.
3 bottles of Coke.
Take one down, pass it around, 2 bottles of Coke on the wall.

2 bottles of Coke on the wall.
2 bottles of Coke.
Take one down, pass it around, 1 bottles of Coke on the wall.

1 bottles of Coke on the wall.
1 bottles of Coke.
Take one down, pass it around, 0 bottles of Coke on the wall.
```

Function Example, brought to you by

(bus drivers *hate* them!)

```
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

How many functions does this program have?

Answer: 2. bottles() and main()

What does the `bottles()` function return?

Answer: nothing (void function)

Function Example, brought to you by

(bus drivers *hate* them!)

Why is it a good idea to make DRINK_TYPE a constant?

Answer: So we can change it to Pepsi if we are masochists. (actual answer: it allows us to make one change that affects many places in the code)

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

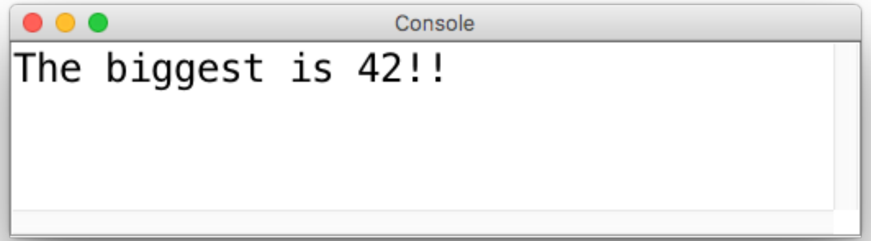
Function Example 2

```cpp
// Function example #2: returning values

#include <iostream>
#include "console.h"

using namespace std;

int larger(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Returns the larger of the two values.
int main() {
    int bigger1 = larger(17, 42); // call the function
    int bigger2 = larger(29, -3); // call the function again
    int biggest = larger(bigger1, bigger2);
    cout << "The biggest is " << biggest << "!!" << endl;
    return 0;
}
```

Function Example 2

```
// Function example #2: returning values

#include <iostream>
#include "console.h"

using namespace std;

int larger(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Returns the larger of the two values.
int main() {
    int bigger1 = larger(17, 42); // call the function
    int bigger2 = larger(29, -3); // call the function again
    int biggest = larger(bigger1, bigger2);
    cout << "The biggest is " << biggest << "!!" << endl;
    return 0;
}
```

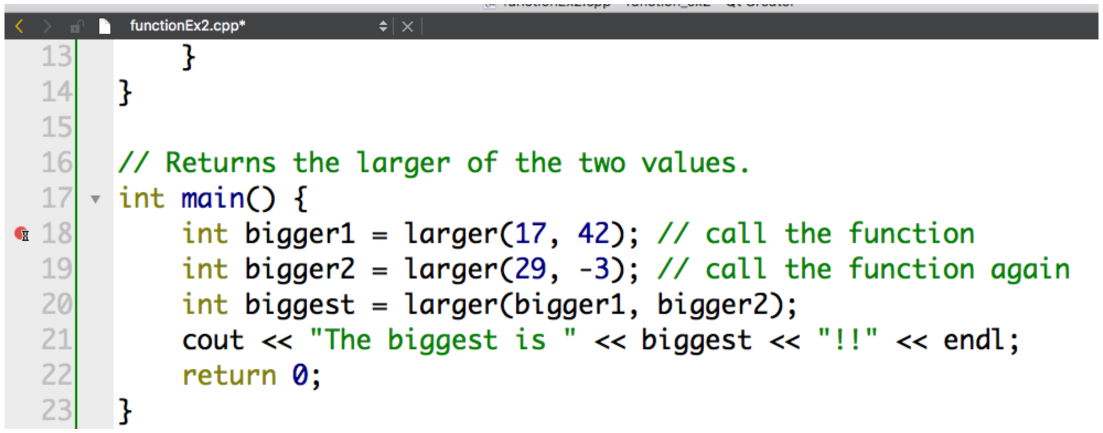
Output:



```
The biggest is 42!!
```

Function Example 2: Debugging

- One of the most powerful features of an Integrated Development Environment (IDE) like Qt Creator is the built-in debugger.
- You can stop the program's execution at any point and look at exactly what is going on under the hood!
- In your program, click to the left of a line of code (line 18 below, for example)

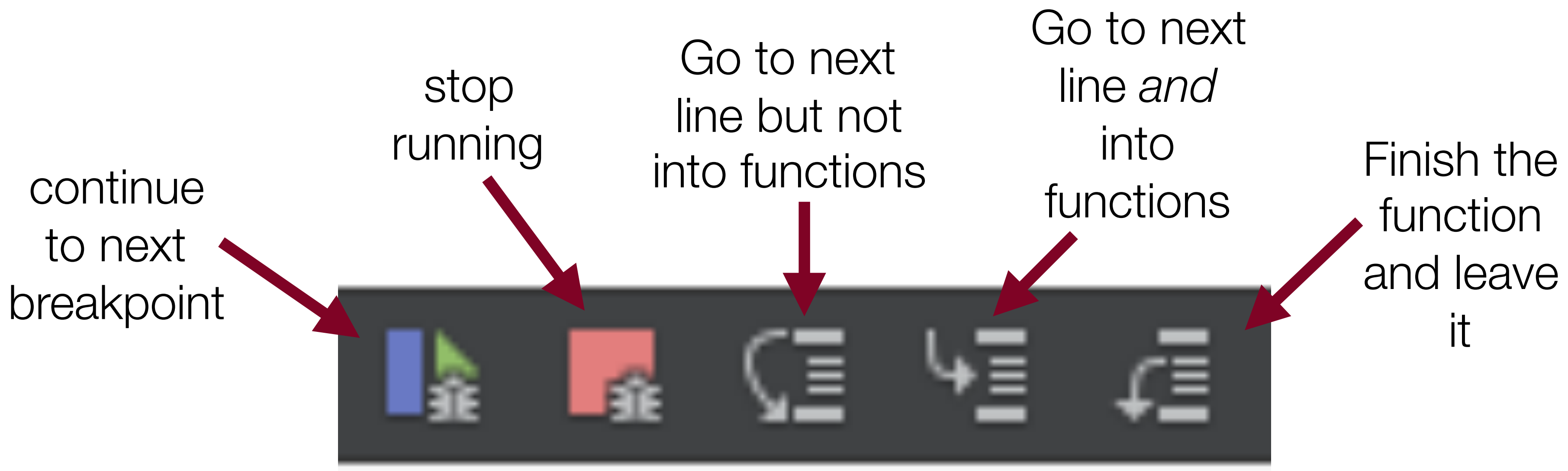


```
13         }
14     }
15 
16     // Returns the larger of the two values.
17     int main() {
18         int bigger1 = larger(17, 42); // call the function
19         int bigger2 = larger(29, -3); // call the function again
20         int biggest = larger(bigger1, bigger2);
21         cout << "The biggest is " << biggest << "!!" << endl;
22         return 0;
23     }
```

- When you run the program in Debug mode (the green triangle with the bug on it), the program will stop at that point. Let's see this in action!

Function Example 2: Debugging

- Notes from live debugging:
  - You can see variable values as the program executes
  - You use the following buttons to continue the program:


continue to next breakpoint
stop running
Go to next line but not into functions
Go to next line *and* into functions
Finish the function and leave it

- Debugging effectively takes a little time to learn, but is super effective if you have hard to find bugs.

Declaration Order

```
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}
```

- Believe it or not, this program does not compile!
- In C++, functions *must* be declared somewhere before they are used.
- But, we like to put our main() function first, because it is better style.

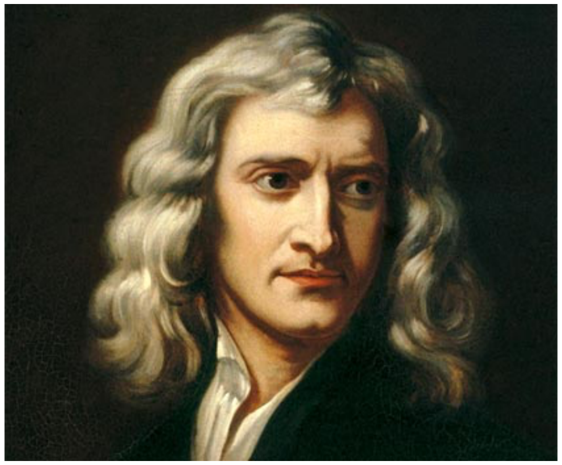
Declaration Order

```
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition
void bottles(int count);

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
// Function Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}
```

- Believe it or not, this program does not compile!
- In C++, functions *must* be declared somewhere before they are used.
- But, we like to put our main() function first, because it is better style.
- What we can do is define the function (called a "*function prototype*") without its body, and that tells the compiler about the function "signature" and the compiler is happy.

C++ Pre-written Functions

- You have written a lot of functions before. What if we wanted to find the square root of a number?
- We could manually write out a function (remember Newton's Method??) -- see the textbook!
- But, this would be counterproductive, and many math functions have already been coded (and coded well!)
- The <cmath> library already has lots and lots of math functions that you can use (you can go look up the code! Actually...it's complicated -- see https://goo.gl/Y9Y55w if you are brave. It is most likely true that the square root function is *built into your computer's processor, so there isn't any readable code*)

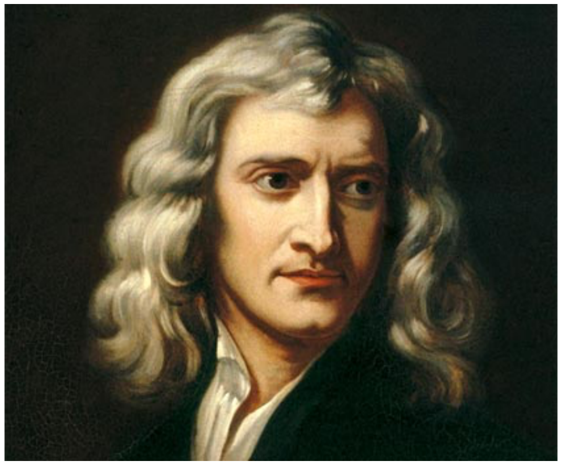


<cmath> functions (2.1)

```
#include <cmath>
```

Function	Description (returns)
abs(value)	absolute value
ceil(value)	rounds up
floor(value)	rounds down
log10(value)	logarithm, base 10
max(value1, value2)	larger of two values
min(value1, value2)	smaller of two values
pow(base, exp)	*base* to the *exp* power
round(value)	nearest whole number
sqrt(value)	square root

sin(value)
cos(value)
tan(value)
	sine/cosine/tangent of an angle in radians



- unlike in Java, you don't write Math. in front of the function name
- see Stanford "gmath.h" library for additional math functionality

Value semantics

- **value semantics:** In Java and C++, when variables (`int, double`) are passed as parameters, their values are copied.
  - Modifying the parameter will not affect the variable passed in.

```
void grow(int age) {
    age = age + 1;
    cout << "grow age is " << age << endl;
}

int main() {
    int age = 20;
    cout << "main age is " << age << endl;
    grow(age);
    cout << "main age is " << age << endl;
    return 0;
}
```

Output:

```
main age is 20
grow age is 21
main age is 20
```

Reference semantics (2.5)

- **reference semantics:** In C++, if you declare a parameter with an `&` after its type, instead of passing a copy of its value, it will link the caller and callee functions to the same variable in memory.
  - Modifying the parameter *will* affect the variable passed in.

```
void grow(int &age) {
    age = age + 1;
    cout << "grow age is " << age << endl;
}

int main() {
    int age = 20;
    cout << "main age is " << age << endl;
    grow(age);
    cout << "main age is " << age << endl;
    return 0;
}
```

Output:

```
main age is 20
grow age is 21
main age is 21
```

Reference semantics (2.5)

- **Notes about references:**
  - References are super important when dealing with objects that have a lot of elements (Vectors, for instance). Because the reference does not copy the structure, it is fast. You don't want to transfer millions of elements between two functions if you can help it!
  - The reference syntax can be confusing, as the "`&`" (ampersand) character is also used to specify the address of a variable or object. The `&` is only used as a reference parameter in the function declaration, not when you call the function:

yes!

```
void grow(int &age) {
    age = age + 1;
    cout << "grow age is "
         << age << endl;
}
```

```
int main() {
    grow(age);
```

yes!

```
    grow(&age);
```

no!

```
    return 0;
}
```

Reference pros/cons

- benefits of reference parameters:
  - a useful way to be able to 'return' more than one value
  - often used with objects, to avoid making bulky copies when passing

- downsides of reference parameters:
  - hard to tell from call whether it is ref; can't tell if it will be changed

```
        foo(a, b, c); // will foo change a, b, or c? :-/
```

  - (very) slightly slower than value parameters
  - can't pass a literal value to a ref parameter

```
        grow(39); // error
```

Reference Example

- Without references, you can't write a swap function to swap two integers. This is true about Java. What happens with the following function?

```
/*
 * Attempts to place a's value into b and vice versa.
 */
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Answer: the original variables are unchanged, because they are passed as copies (values)!

Reference Example

- *With* references, you *can* write a swap function to swap two integers, because you can access the original variables:

```
/*
 * Places a's value into b and vice versa.
 */
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Answer: the original variables **are changed**, because they are passed as references !

Tricky Reference Mystery Example

What is the output of this code? Talk to your neighbor!

```
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

```
// A. 5 2 8
// B. 5 3 7
// C. 6 1 8
// D. 61 13
// E. other
```

Note: please don't obfuscate your code like this! :(

See the International Obfuscated C Contest for much, much worse examples

Tricky Reference Mystery Example

What is the output of this code?

```
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

```
// A. 5 2 8
// B. 5 3 7
// C. 6 1 8
// D. 61 13
// E. other
```

Note: please don't obfuscate your code like this! :(
See the International Obfuscated C Contest for much, much worse examples

Quadratic Exercise -- how do you return multiple things?

- A quadratic equation for variable x is one of the form:
  `ax`^{2} **`+ bx +c = 0`**, for some numbers **`a`**, **`b`**, and **`c`**.

- The two roots of a quadratic equation can be found using the quadratic formula at right.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Example: The roots of **`x`**^{2}**`-3x-4=0`** are **`x=4`** and **`x=-1`**

- How would we write a function named quadratic to solve quadratic equations?
  - What parameters should it accept?
  - Which parameters should be passed by value, and which by reference?
  - What, if anything, should it return?

- We have choices!

Quadratic Exercise -- how do you return multiple things?

```
/*
 * Solves a quadratic equation ax^2 + bx + c = 0,
 * storing the results in output parameters root1 and root2.
 * Assumes that the given equation has two real roots.
 */
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

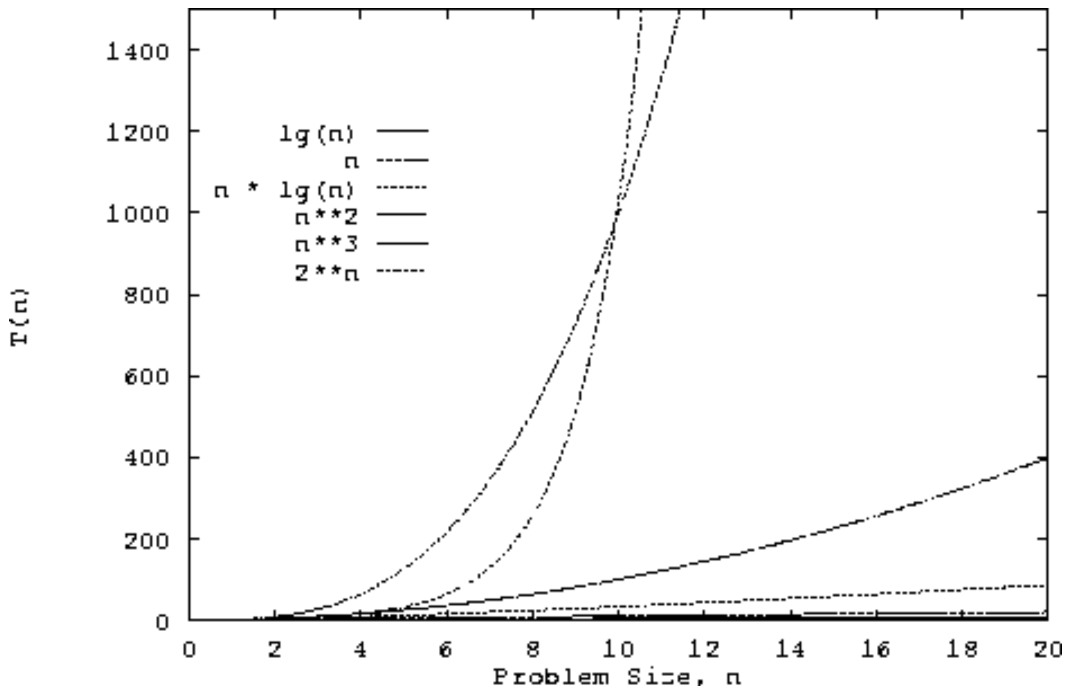
- How are we "returning" the results? Answer: by reference
- What other choices could we have made? Talk to your neighbor!

Quadratic Exercise -- how do you return multiple things?

- Possible choices:
  - We could have returned a boolean if the roots were imaginary
  - We could have added extra parameters to support some form of imaginary numbers
  - We could have called an error function inside this function (but that is not always a good idea -- functions like this should generally have an interface through the parameters and/or return value, and should gracefully fail)
  - We could have re-written the function as two functions that return either the positive or negative root, without using references.
  - We could have returned a Vector<double> object (tricky syntax!)

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Computational Complexity



Computational Complexity

How does one go about analyzing programs to compare how the program behaves as it scales? E.g., let's look at a **`vectorMax()`** function:

```
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;
}
```

What is *n*? Why is it important to this function?

Computational Complexity

```
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;
}
```

If we want to see how this algorithm behaves as *n* changes, we could do the following:

(1) Code the algorithm in C++

(2) Determine, for each instruction of the compiled program the time needed to execute that instruction (need assembly language)

(3) Determine the number of times each instruction is executed when the program is run.

(4) Sum up all the times we calculated to get a running time.

Computational Complexity

```
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;
}
```

Steps 1-4 on the previous slide…might work, but it is complicated, especially for today's machines that optimize everything "under the hood." (and reading assembly code takes a certain patience).

Assembly Code for vectorMax() function...

```
0x000000010014adf0 <+0>:  push   %rbp
0x000000010014adf1 <+1>:  mov    %rsp,%rbp
0x000000010014adf4 <+4>:  sub    $0x20,%rsp
0x000000010014adf8 <+8>:  xor    %esi,%esi
0x000000010014adfa <+10>: mov    %rdi,-0x8(%rbp)
0x000000010014adfe <+14>: mov    -0x8(%rbp),%rdi
0x000000010014ae02 <+18>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae07 <+23>: mov    (%rax),%esi
0x000000010014ae09 <+25>: mov    %esi,-0xc(%rbp)
0x000000010014ae0c <+28>: mov    -0x8(%rbp),%rdi
0x000000010014ae10 <+32>: callq  0x10014afb0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+304>
0x000000010014ae15 <+37>: mov    %eax,-0x10(%rbp)
0x000000010014ae18 <+40>: movl   $0x1,-0x14(%rbp)
0x000000010014ae1f <+47>: mov    -0x14(%rbp),%eax
0x000000010014ae22 <+50>: cmp    -0x10(%rbp),%eax
0x000000010014ae25 <+53>: jge    0x10014ae6c <vectorMax(Vector<int>&)+124>
0x000000010014ae2b <+59>: mov    -0xc(%rbp),%eax
0x000000010014ae2e <+62>: mov    -0x8(%rbp),%rdi
0x000000010014ae32 <+66>: mov    -0x14(%rbp),%esi
0x000000010014ae35 <+69>: mov    %eax,-0x18(%rbp)
0x000000010014ae38 <+72>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae3d <+77>: mov    -0x18(%rbp),%esi
0x000000010014ae40 <+80>: cmp    (%rax),%esi
0x000000010014ae42 <+82>: jge    0x10014ae59 <vectorMax(Vector<int>&)+105>
0x000000010014ae48 <+88>: mov    -0x8(%rbp),%rdi
0x000000010014ae4c <+92>: mov    -0x14(%rbp),%esi
0x000000010014ae4f <+95>: callq  0x10014aea0 <std::__1::basic_ostream<char, std::__1::char_traits<char> >::operator<<(long)+32>
0x000000010014ae54 <+100>:   mov    (%rax),%esi
0x000000010014ae56 <+102>:   mov    %esi,-0xc(%rbp)
0x000000010014ae59 <+105>:   jmpq   0x10014ae5e <vectorMax(Vector<int>&)+110>
0x000000010014ae5e <+110>:   mov    -0x14(%rbp),%eax
0x000000010014ae61 <+113>:   add    $0x1,%eax
0x000000010014ae64 <+116>:   mov    %eax,-0x14(%rbp)
0x000000010014ae67 <+119>:   jmpq   0x10014ae1f <vectorMax(Vector<int>&)+47>
0x000000010014ae6c <+124>:   mov    -0xc(%rbp),%eax
0x000000010014ae6f <+127>:   add    $0x20,%rsp
0x000000010014ae73 <+131>:   pop    %rbp
0x000000010014ae74 <+132>:   retq
```

Algorithm Analysis: Primitive Operations

Instead of those complex steps, we can define *primitive operations* for our C++ code.

- Assigning a value to a variable
- Calling a function
- Arithmetic (e.g., adding two numbers)
- Comparing two numbers
- Indexing into a Vector
- Returning from a function

We assign "1 operation" to each step. We are trying to gather data so we can compare this to other algorithms.

Algorithm Analysis: Primitive Operations

```
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;
}
```

executed once (2 ops)

executed once (2 ops)

executed once (1 op)

ex. n times (*n* ops)

executed *n-1* times (2*(*n*-1) ops))

ex. n-1 times (2*(n-1) ops)

ex. at most n-1 times (2*(n-1) ops), but as few as zero times

ex. once (1 op)

Algorithm Analysis: Primitive Operations

Summary:

Primitive operations for **`vectorMax()`**:

at least: $2 + 2 + 1 + n + 4 * (n - 1) + 1 = 5n + 2$

at most: $2 + 2 + 1 + n + 6 * (n - 1) + 1 = 7n$

i.e., if there are *n* items in the Vector, there are between 5*n*+2 operations and 7*n* operations completed in the function.

Algorithm Analysis: Primitive Operations

Summary:

Primitive operations for **`vectorMax()`**:

best case: $5n + 2$

worst case: $7n$

In other words, we can get a "best case" and "worst case" count

Algorithm Analysis: Simplify!

Do we *really* need this much detail? Nope!

Let's simplify: we want a "big picture" approach.

It is enough to know that **`vectorMax()`** grows

linearly proportionally to n

In other words, as the number of elements increases, the algorithm has to do proportionally more work, and that relationship is linear. 8x more elements? 8x more work.

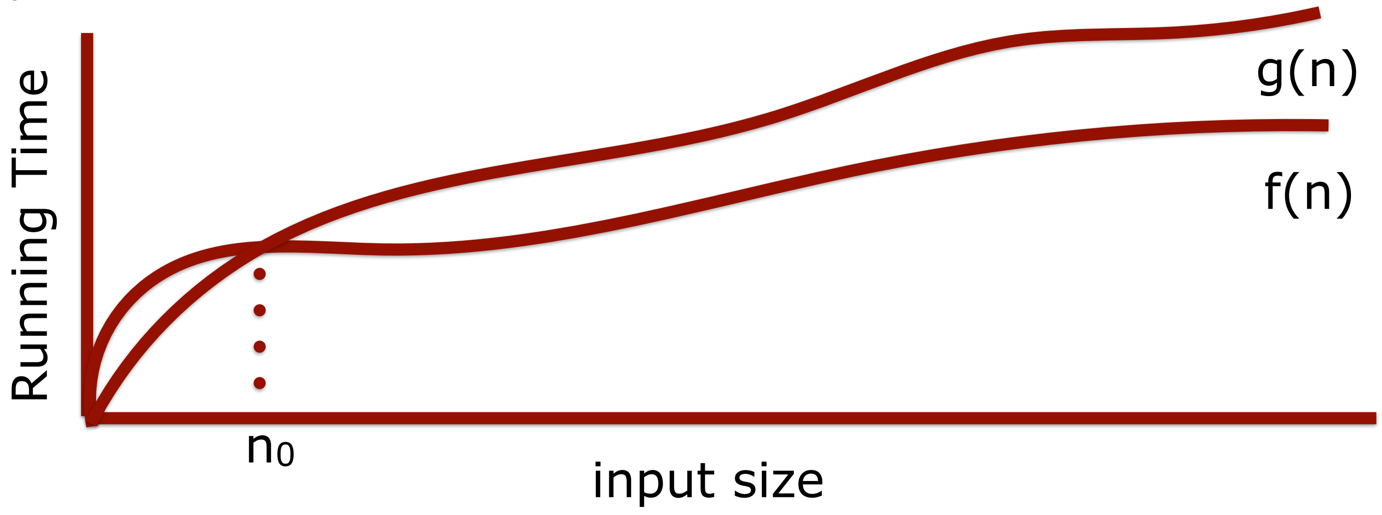
Algorithm Analysis: Big-O

Our simplification uses a mathematical construct known as “Big-O” notation — think “O” as in “on the Order of.”

Wikipedia:
“Big-O notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions.”

Algorithm Analysis: Big-O

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$. This definition is often referred to as the "big-Oh" notation. We can also say, "$f(n)$ is *order* $g(n)$."



Algorithm Analysis: Big-O

Dirty little trick for figuring out Big-O: look at the number of steps you calculated, throw out all the constants, find the "biggest factor" and that's your answer:

$5n + 2$ is O(*n*)

Why? Because constants are not important at this level of understanding.

Algorithm Analysis: Big-O

We will care about the following functions that appear often in data structures:

constant	***logarithmic***	***linear***	***n log n***	***quadratic***	***polynomial (other than n^2)***	***exponential***
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a>1)$

When you are deciding what Big-O is for an algorithm or function, simplify until you reach one of these functions, and you will have your answer.

Algorithm Analysis: Big-O

constant	logarithmic	linear	n log n	quadratic	polynomial (other than n^2)	exponential
$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^k)$ $(k \geq 1)$	$O(a^n)$ $(a > 1)$

Practice: what is Big-O for this function?

$$20n^3 + 10n \log n + 5$$

Answer: $O(n^3)$

First, strip the constants: $n^3 + n \log n$

Then, find the biggest factor: n^3

Algorithm Analysis: Big-O

constant	***logarithmic***	***linear***	***n log n***	***quadratic***	***polynomial*** ***(other than n^2)***	***exponential***
O(1)	***O(log n)***	***O(n)***	***O(n log n)***	***$O(n^2)$***	***$O(n^k)$ ($k \geq 1$)***	***$O(a^n)$ ($a>1$)***

Practice: what is Big-O for this function?

$$2000 \log \mathrm{n} + 7n \log n + 5$$

Answer: O(n log n)

First, strip the constants: log *n* + *n* log *n*

Then, find the biggest factor: *n* log *n*

Algorithm Analysis: Back to vectorMax()

```
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;
}
```

When you are analyzing an algorithm or code for its *computational complexity* using Big-O notation, you can ignore the primitive operations that would contribute less-important factors to the run-time. Also, you always take the *worst case* behavior for Big-O.

Algorithm Analysis: Back to vectorMax()

```
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;~~
}
```

When you are analyzing an algorithm or code for its *computational complexity* using Big-O notation, you can ignore the primitive operations that would contribute less-important factors to the run-time. Also, you always take the *worst case* behavior for Big-O.

So, for vectorMax(): ignore the original two variable initializations, the return statement, the comparison, and the setting of currentMax in the loop.

Algorithm Analysis: Back to vectorMax()

```
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;~~
}
```

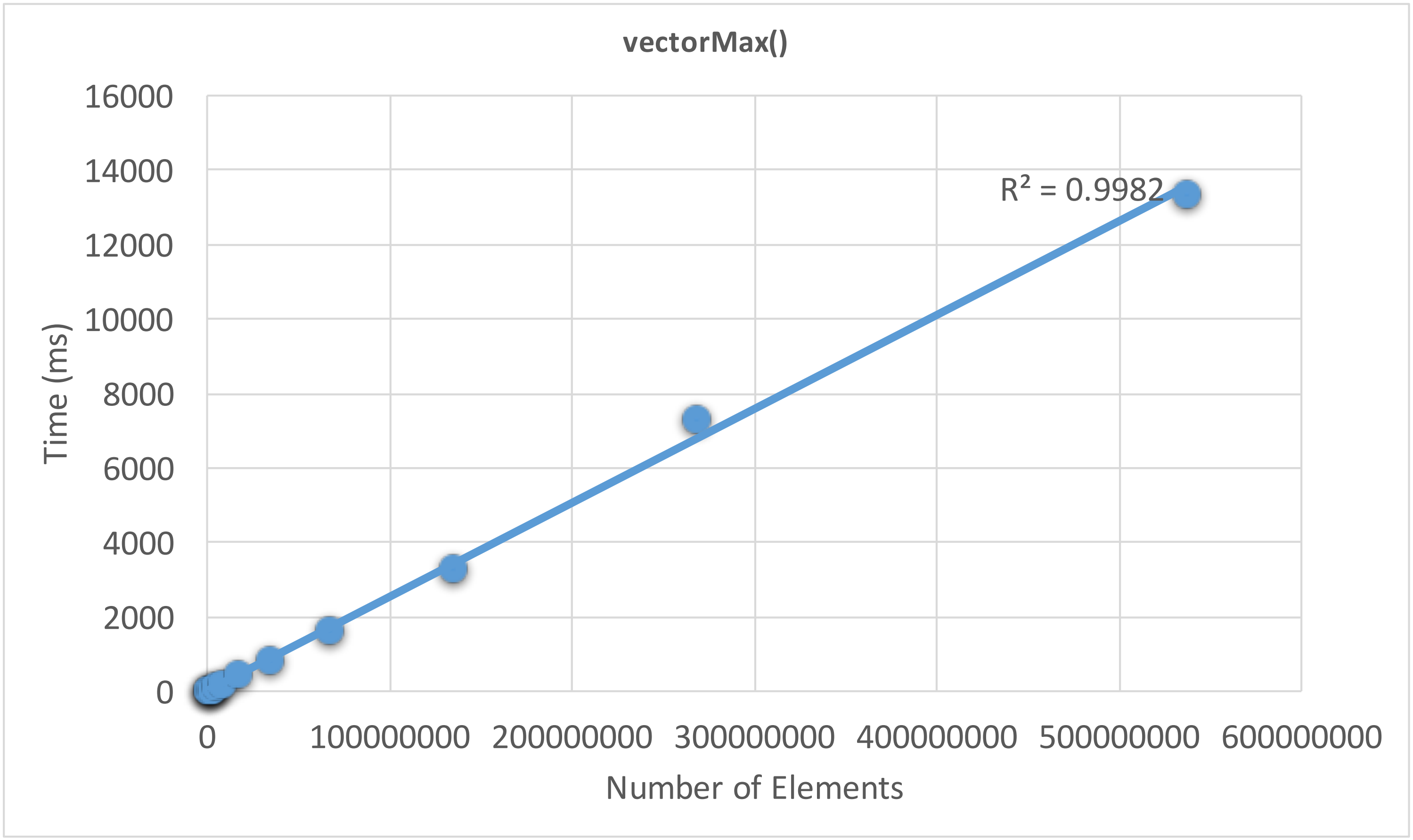
So, for vectorMax(): ignore the original two variable initializations, the return statement, the comparison, and the setting of currentMax in the loop.

Notice that the important part of the function is the fact that the loop conditions will change with the size of the array: for each extra element, there will be one more iteration. This is a *linear* relationship, and therefore O(*n*).

Algorithm Analysis: Back to vectorMax()

Data: In the lecture code, you will find a test program for vectorMax(), which runs the function on an increasing (by powers of two) number of vector elements. This is the data I gathered from my computer.

As you can see, it's a linear relationship!



Algorithm Analysis: Nested Loops

```
int nestedLoop1(int n){
        int result = 0;
        for (int i=0;i<n;i++){
                for (int j=0;j<n;j++){
                        result++;
                }
        }
        return result;
}
```

Also go through the outer loop n times

Inner loop complexity: *O(n)*

Total complexity: *$O(n^2)$* *(quadratic)*

In general, we don't like $O(n^2)$ behavior! Why?

As an example: let's say an $O(n^2)$ function takes 5 seconds for a container with 100 elements. How much time would it take if we had 1000 elements?

500 seconds! This is because 10x more elements is (10^2)x more time!

Algorithm Analysis: Nested Loops

```
int nestedLoop1(int n){
        int result = 0;
        for (int i=0;i<n;i++){
                for (int j=0;j<n;j++){
                        for (int k=0;k<n;k++)
                                result++;
                }
        }
        return result;
}
```

What would the complexity be of a 3-nested loop?

Answer: n^3 (polynomial)

In real life, this comes up in 3D imaging, video, etc., and it is **slow**!
Graphics cards are built with hundreds or thousands of processors to tackle this problem!

Algorithm Analysis: Linear Search

```
void linearSearchVector(Vector<int> &vec, int numToFind){
    int numCompares = 0;
    bool answer = false;
    int n = vec.size();

    for (int i = 0; i < n; i++) {
        numCompares++;
        if (vec[i]==numToFind) {
            answer = true;
            break;
        }
    }
    cout << "Found? " << (answer ? "True" : "False") << ", "
         << "Number of compares: " << numCompares << endl << endl;
}
```

Best case? O(1)

Worst case? O(*n*)

Complexity: O(n) (linear, worst case)

You have to walk through the entire vector one element at a time.

Algorithm Analysis: *Binary* Search

There is another type of search that we can perform on a list that is in order: binary search (as seen in 106A!)

If you have ever played a "guess my number" game before, you will have implemented a binary search, if you played the game efficiently!

The game is played as follows:

- one player thinks of a number between 0 and 100 (or any other maximum).
- the second player guesses a number between 1 and 100
- the first player says "higher" or "lower," and the second player keeps guessing until they guess correctly.

Algorithm Analysis: *Binary* Search

The most efficient guessing algorithm for the number guessing game is simply to choose a number that is between the high and low that you are currently bound to. Example:
bounds: 0, 100
guess: 50 (no, the answer is lower)
new bounds: 0, 49
guess: 25 (no, the answer is higher)
new bounds: 26, 49
guess: 38
etc.

With each guess, the search space is *divided into two*.

Algorithm Analysis: Binary Search

```
void binarySearchVector(Vector<int> &vec, int numToFind) {
    int low=0;
    int high=vec.size()-1;
    int mid;
    int numCompares = 0;
    bool found=false;
    while (low <= high) {
        numCompares++;
        //cout << low << ", " << high << endl;
        mid = low + (high - low) / 2; // to avoid overflow
        if (vec[mid] > numToFind) {
            high = mid - 1;
        }
        else if (vec[mid] < numToFind) {
            low = mid + 1;
        }
        else {
            found = true;
            break;
        }
    }
    cout << "Found? " << (found ? "True" : "False") << ", " <<
    "Number of compares: " << numCompares << endl << endl;
}
```

Best case? O(1)

Worst case? O(log *n*)

Complexity: O(log *n*) (logarithmic, worst case)

Technically, this is O($\log_2 n$), but we will not worry about the base.

The general rule for determining if something is logarithmic: if the problem is one of "divide and conquer," it is logarithmic. If, at each stage, the problem size is cut in half (or a third, etc.), it is logarithmic.

Algorithm Analysis: Constant Time

When an algorithm's time is *independent* of the number of elements in the container it holds, this is *constant time* complexity, or O(1). We love O(1) algorithms! Examples include (for efficiently designed data structures):

- Adding or removing from the *end* of a Vector.
- Pushing onto a stack or popping off a stack.
- Enqueuing or dequeuing from a queue.
- Other cool data structures we will cover soon (*hint:* one is a "hash table"!)

Algorithm Analysis: Exponential Time

There are a number of algorithms that have *exponential* behavior. If we don't like quadratic or polynomial behavior, we *really* don't like exponential behavior.

Example: what does the following beautiful recursive function do?

```
long mysteryFunc(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return mysteryFunc(n-1) + mysteryFunc(n-2);
}
```

This is the *fibonacci sequence!* 0, 1, 1, 2, 3, 5, 8, 13, 21 …

Algorithm Analysis: Exponential Time

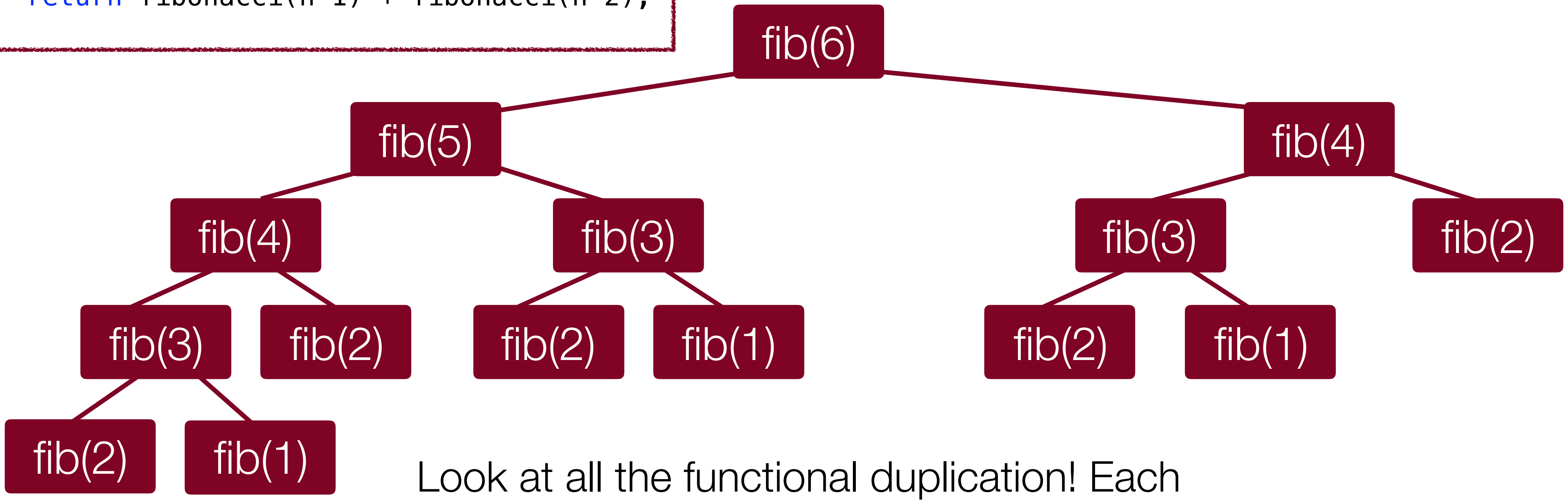
```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Beautiful, but a flawed algorithm! Yes, it works, but why is it flawed? Let's look at the call tree for fib(6):

Algorithm Analysis: Exponential Time

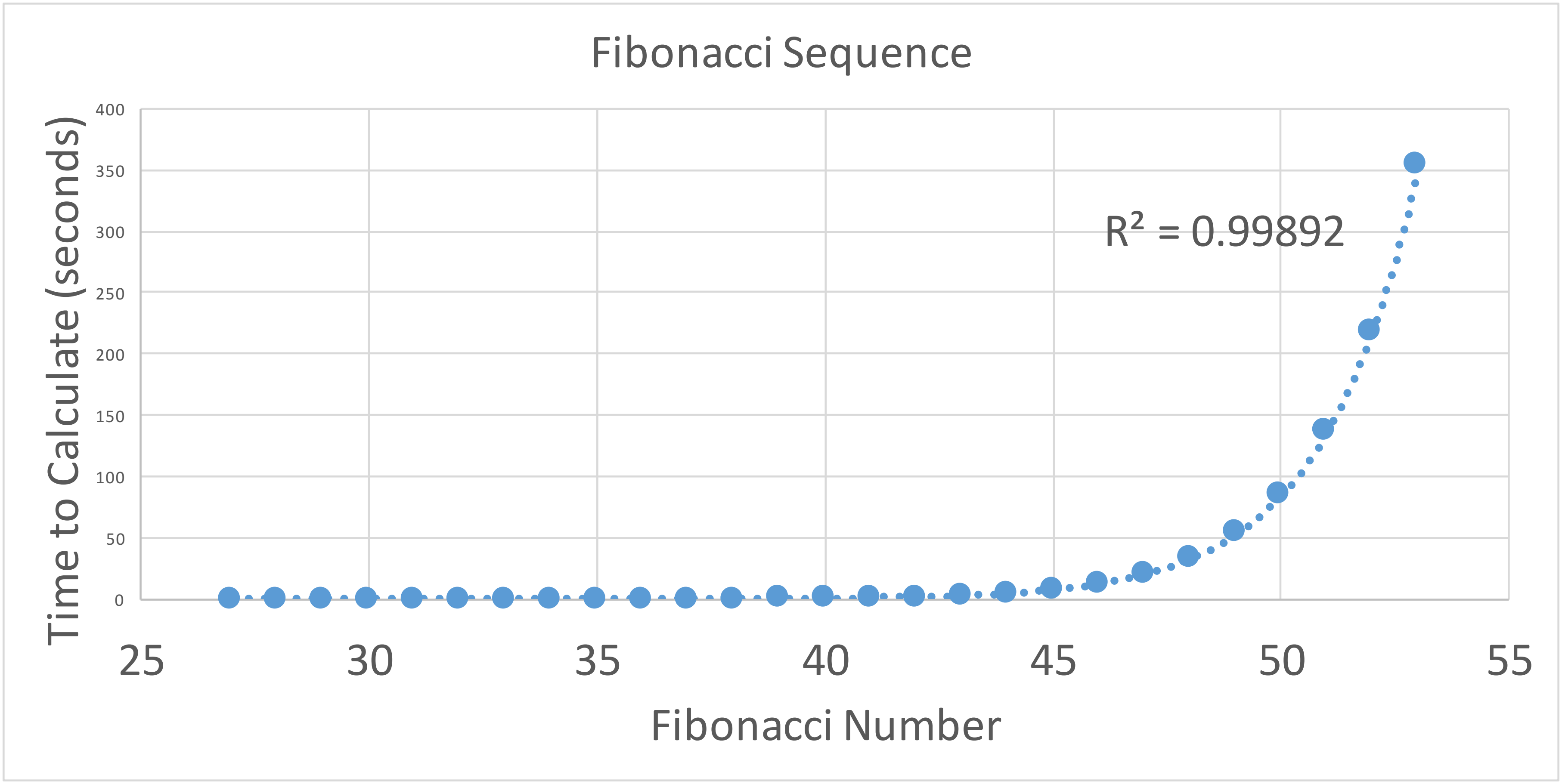
```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Beautiful, but a flawed algorithm! Yes, it works, but why is it flawed? Let's look at the call tree for fib(6):



Look at all the functional duplication! Each call (down to level 3) has to make two recursive calls, and many are duplicated!

Fibonacci Sequence Time to Calculate Recursively



Ramifications of Big-O Differences

Some numbers:

If we have an algorithm that has 1000 elements, and the O(log n) version runs in 10 nanoseconds…

constant	***logarithmic***	***linear***	***n log n***	***quadratic***	***polynomial (n^3)***	***exponential (a==2)***
O(1)	***O(log n)***	***O(n)***	***O(n log n)***	***$O(n^2)$***	***$O(n^k)$ ($k \geq 1$)***	***$O(a^n)$ ($a>1$)***
1ns	***10ns***	***1microsec***	***10microsec***	***1millisec***	***1 sec***	***10^{292} years***

Ramifications of Big-O Differences

Some numbers:

If we have an algorithm that has 1000 elements, and the O(log n) version runs in 10 milliseconds…

constant	***logarithmic***	***linear***	***n log n***	***quadratic***	***polynomial (n^3)***	***exponential (a==2)***
O(1)	***O(log n)***	***O(n)***	***O(n log n)***	***$O(n^2)$***	***$O(n^k)$ ($k \geq 1$)***	***$O(a^n)$ ($a>1$)***
1ms	***10ms***	***1sec***	***10sec***	***17 minutes***	***277 hours***	***heat death of the universe***

Summary of Big-O Functions

- **Constant, O(1)**: not dependent on *n*
- **Linear, O(n)**: at each step, reduce the problem by a constant amount like 1 (or two, three, etc.)
- **Logarithmic, O(log n)**: cut the problem by 1/2, 1/3, etc.
- **Quadratic, O(n^2)**: doubly nested things
- **O(n^3)**: triply nested things
- **Exponential, O(a^n)**: reduce a problem into two or more subproblems of a smaller size.

Recap (functions)

- Fauxtoshop is out — start early!
- There are plenty of opportunities to get help before the deadline next Friday!
- Functions are to C++ as methods are to Java (and very, very similar)
- The Qt Creator debugger can show you real-time details of what your program is doing, and it will come in super handy when you are trying to find tricky bugs in your code.
- You must declare function prototypes before using them in C++!
- There are lots of pre-written functions (e.g., <cmath> and the Stanford Library functions) that have been written already. Use them!
- Value semantics: pass by "value" means that you get a copy of a variable, *not the original!*
- Reference semantics: using the `&` in a parameter definition will give the function access to *the original variable*. This can be tricky until you get used to it.

Recap (Big O)

- Asymptotic Analysis / Big-O / Computational Complexity
  - We want a "big picture" assessment of our algorithms and functions
  - We can ignore constants and factors that will contribute less to the result!
  - We most often care about *worst case* behavior.
  - We love O(1) and O(log n) behaviors!
- Big-O notation is useful for determining how a particular algorithm behaves, but be careful about making comparisons between algorithms -- sometimes this is helpful, but it can be misleading.
- Algorithmic complexity can determine the difference between running your program over your lunch break, or waiting until the Sun becomes a Red Giant and swallows the Earth before your program finishes -- that's how important it is!

References and Advanced Reading (Functions)

- **References (in general, not the C++ references!):**
  - Textbook Chapters 2 and 3
  - <cmath> functions: http://en.cppreference.com/w/cpp/header/cmath
  - Obfuscated C contest: http://www.ioccc.org
  - Code from class: see class website (https://cs106x.stanford.edu)

- **Advanced Reading:**
  - Wikipedia article on C++ References: https://en.wikipedia.org/wiki/Reference_(C%2B%2B)
  - More information on C++ references: http://www.learncpp.com/cpp-tutorial/611-references/
  - C++ Newton's Method question on StackOverflow: http://codereview.stackexchange.com/questions/43456/square-root-approximation-with-newtons-method
  - If you are super-brave, look at the square root C++ function in the C library: http://osxr.org:8080/glibc/source/sysdeps/ieee754/dbl-64/e_sqrt.c?v=glibc-2.14#0048

References and Advanced Reading (Big O)

- **References:**
  - Wikipedia on BigO: https://en.wikipedia.org/wiki/Big_O_notation
  - Binary Search: https://en.wikipedia.org/wiki/Binary_search_algorithm
  - Fibonacci numbers: https://en.wikipedia.org/wiki/Fibonacci_number

- **Advanced Reading:**
  - Big-O Cheat Sheet: http://bigocheatsheet.com
  - More details on Big-O: http://web.mit.edu/16.070/www/lecture/big_o.pdf
  - More details: http://dev.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm
  - GPUs and GPU-Accelerated computing: http://www.nvidia.com/object/what-is-gpu-computing.html
  - Video on Fibonacci sequence: https://www.youtube.com/watch?v=Nu-lW-Ifyec
  - Fibonacci numbers in nature: http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html