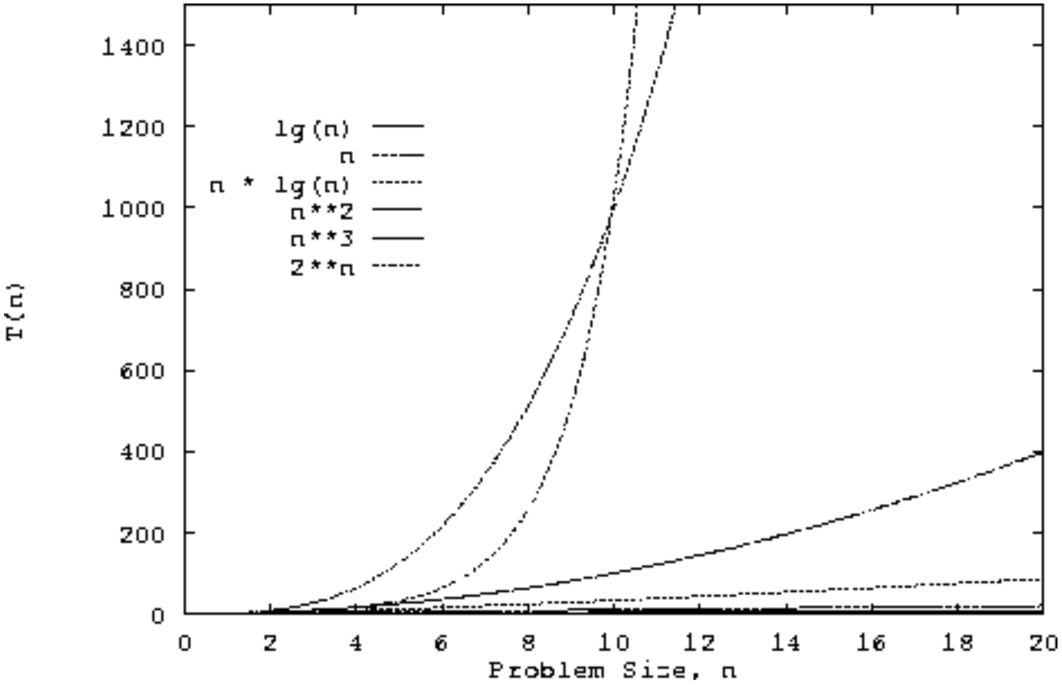
CS 106X
Lecture 3: Big O, Vectors Grids

Friday, January 13, 2017



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

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapters 5.1-5.2, Section 10.2

Today's Topics

- Logistics:
  - Signing up for section
  - Extra Help Sessions Saturday and Sunday
- A note on the honor code
- Introduction to Computational Complexity and "Big O"
- Vectors
- Grids
- Reading Assignment: Chapter 5.1-5.2, Section 10.2

Logistics

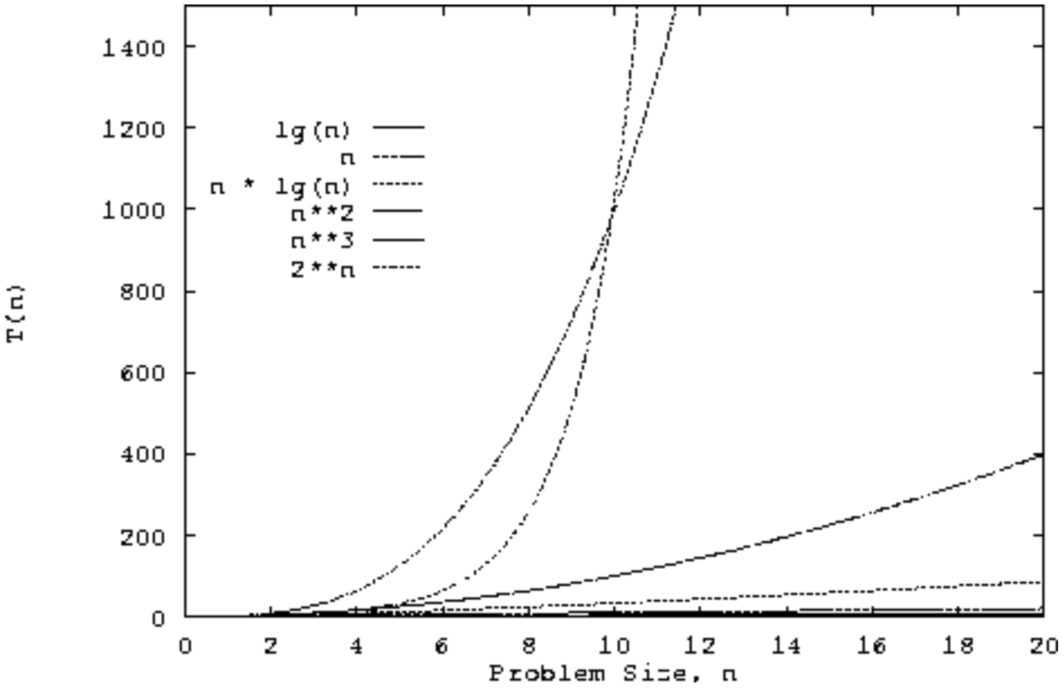
- Signing up for section: you must put your available times by Sunday January 15th at 5pm (opens Thursday at 5pm).
  - Go to cs198.stanford.edu to sign up.

- Extra help sessions: Saturday at 2pm, Sunday at 2pm, Monday at 2pm
- Stop by Gates 191 (text Chris at 857-234-0211 if you can't get into the building)

A Note on the Honor Code

- Honor code handout: http://web.stanford.edu/class/cs106x//handouts/3-HonorCode.pdf

Back to Computational Complexity



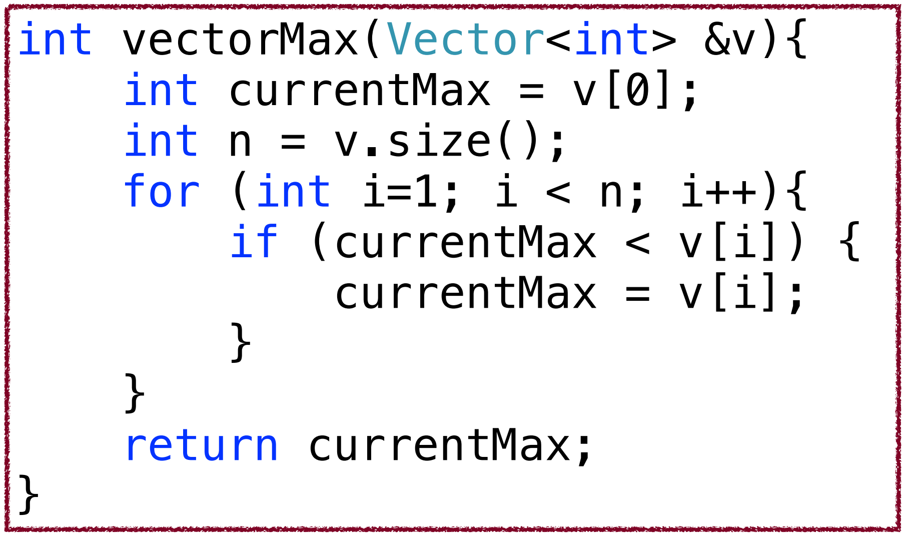
Computational Complexity

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
```

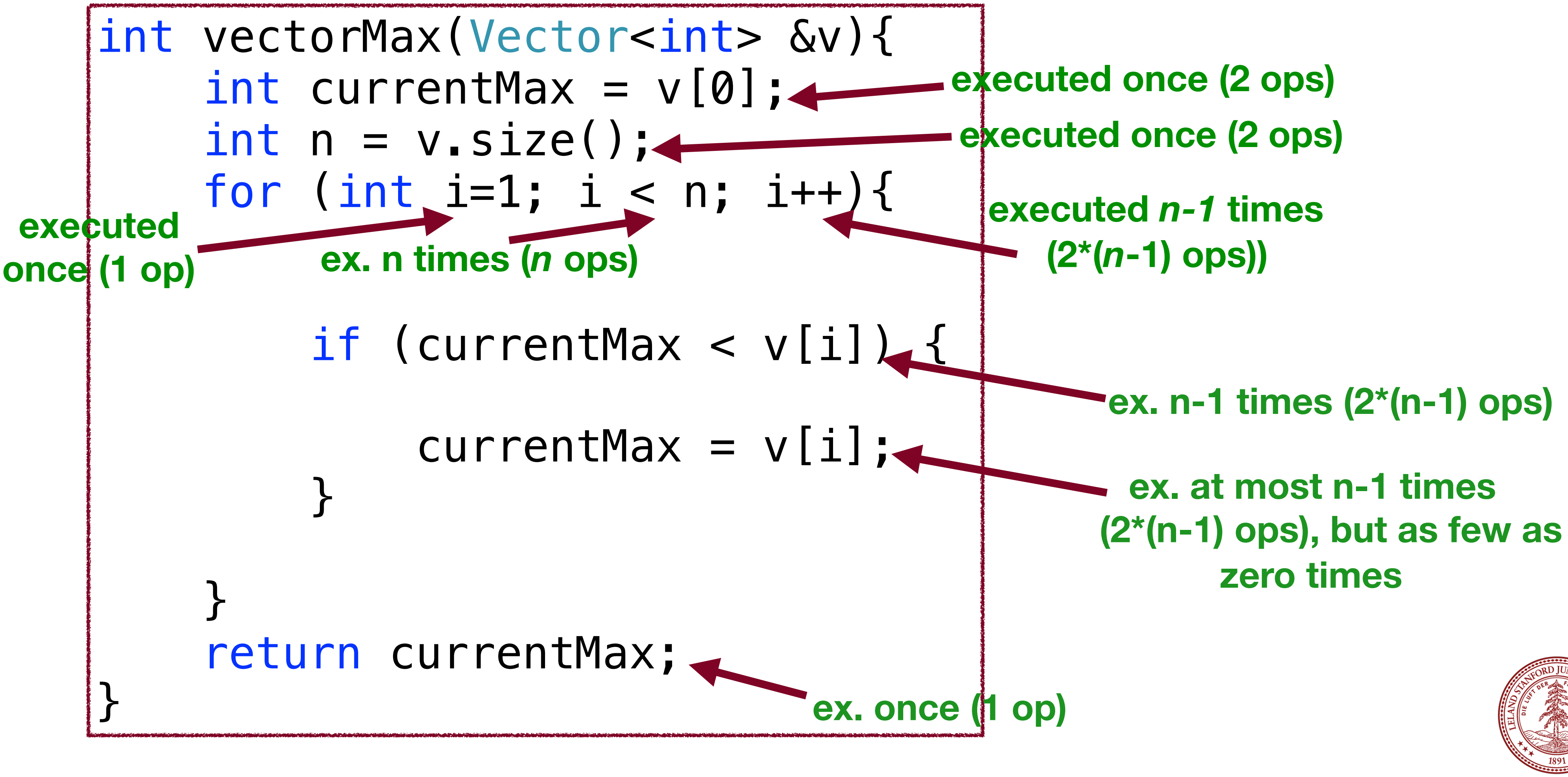
Algorithm Analysis: Primitive Operations

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.

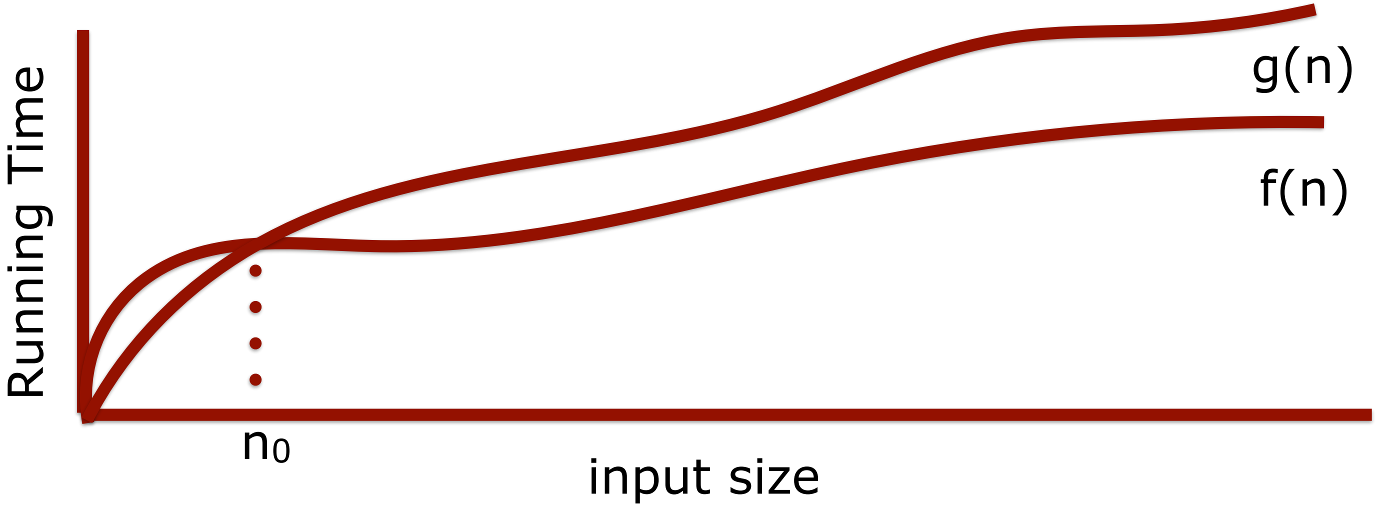
Algorithm Analysis: Big-O

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:

$$5n + 2 \text{ is } O(n)$$

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≥1)	O(a^n) (a>1)

Practice: what is Big-O for this function?

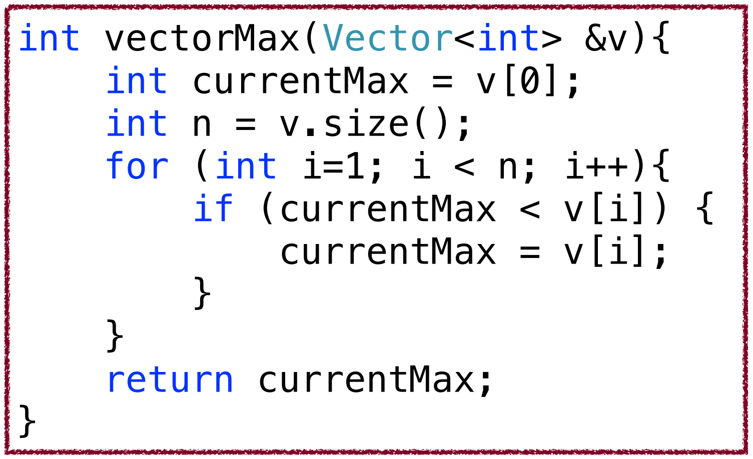
$$2000 \log \mathrm{n} + 7n \log n + 5$$

Answer: O(n log n)

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

Then, find the biggest factor: n log n

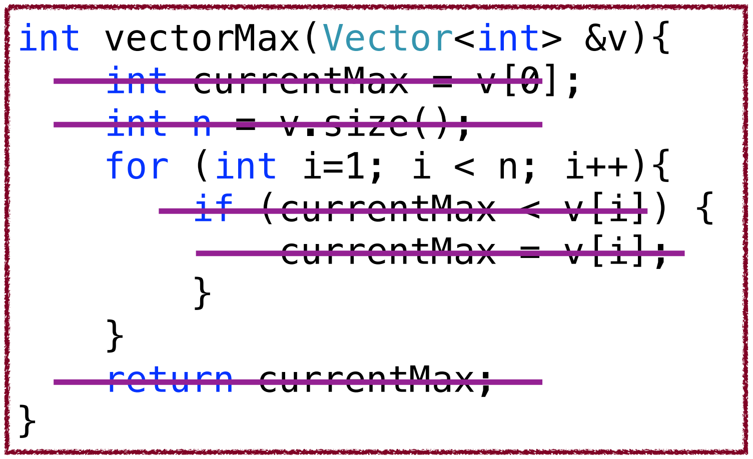
Algorithm Analysis: Back to vectorMax()



```
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;~~
}
```

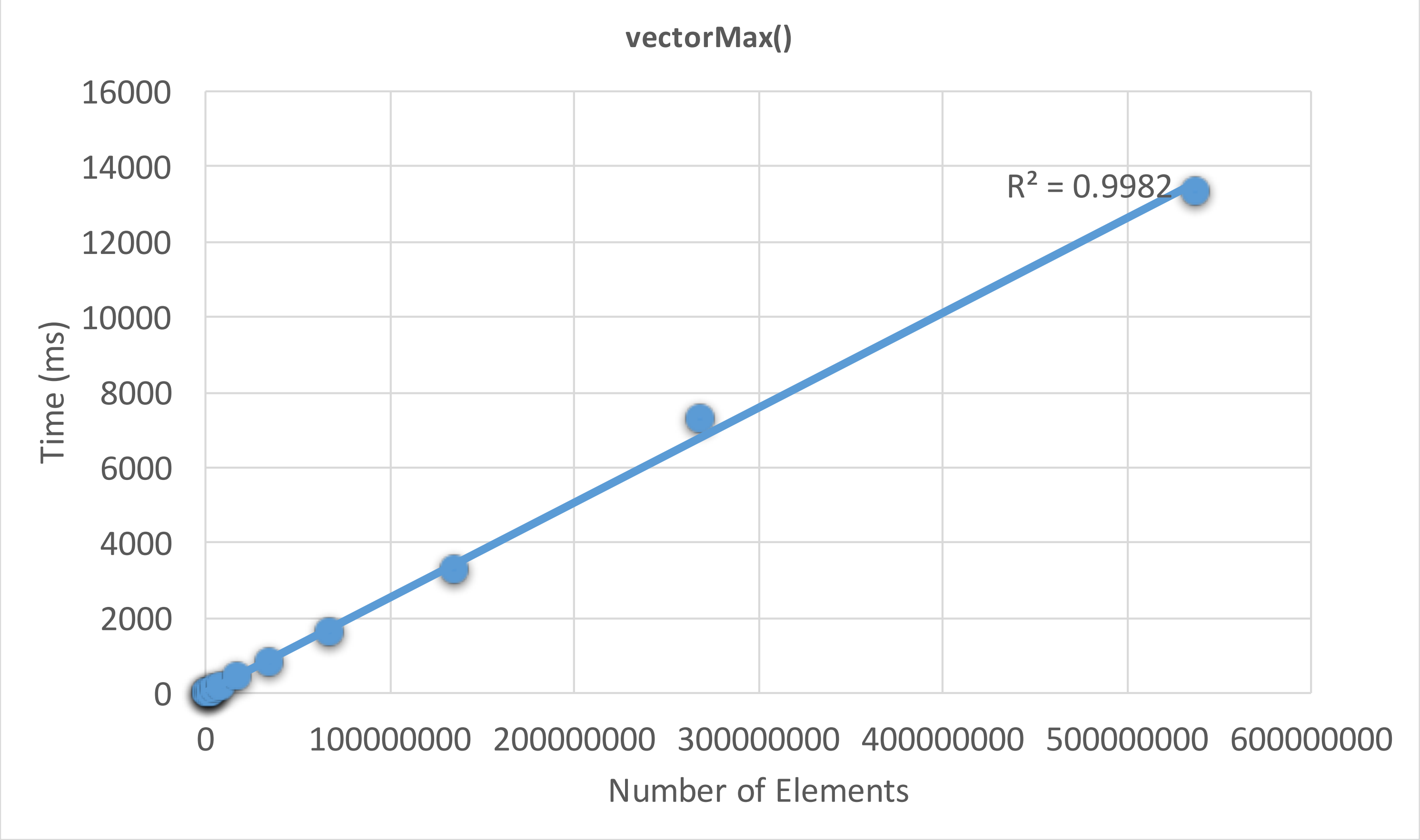
So, for vectorMax(): ignore the original two variable initializations, the return statement, the comparison, and the setting of currentMax in the loop.

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

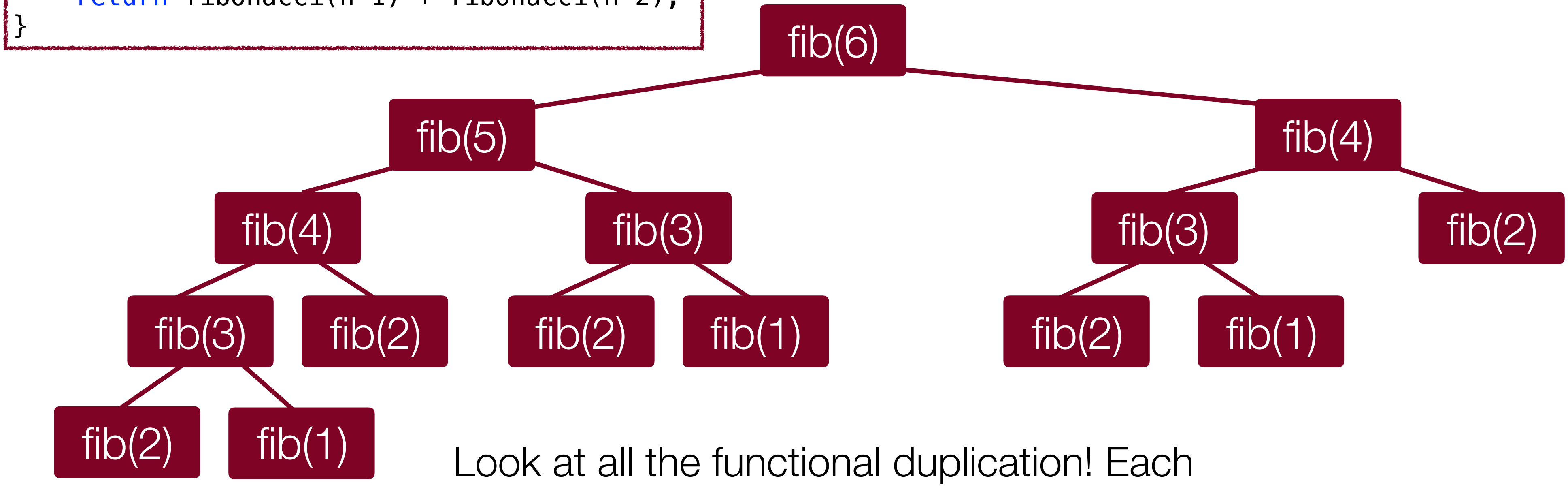
```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

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):



fib(6)

fib(5) fib(4)

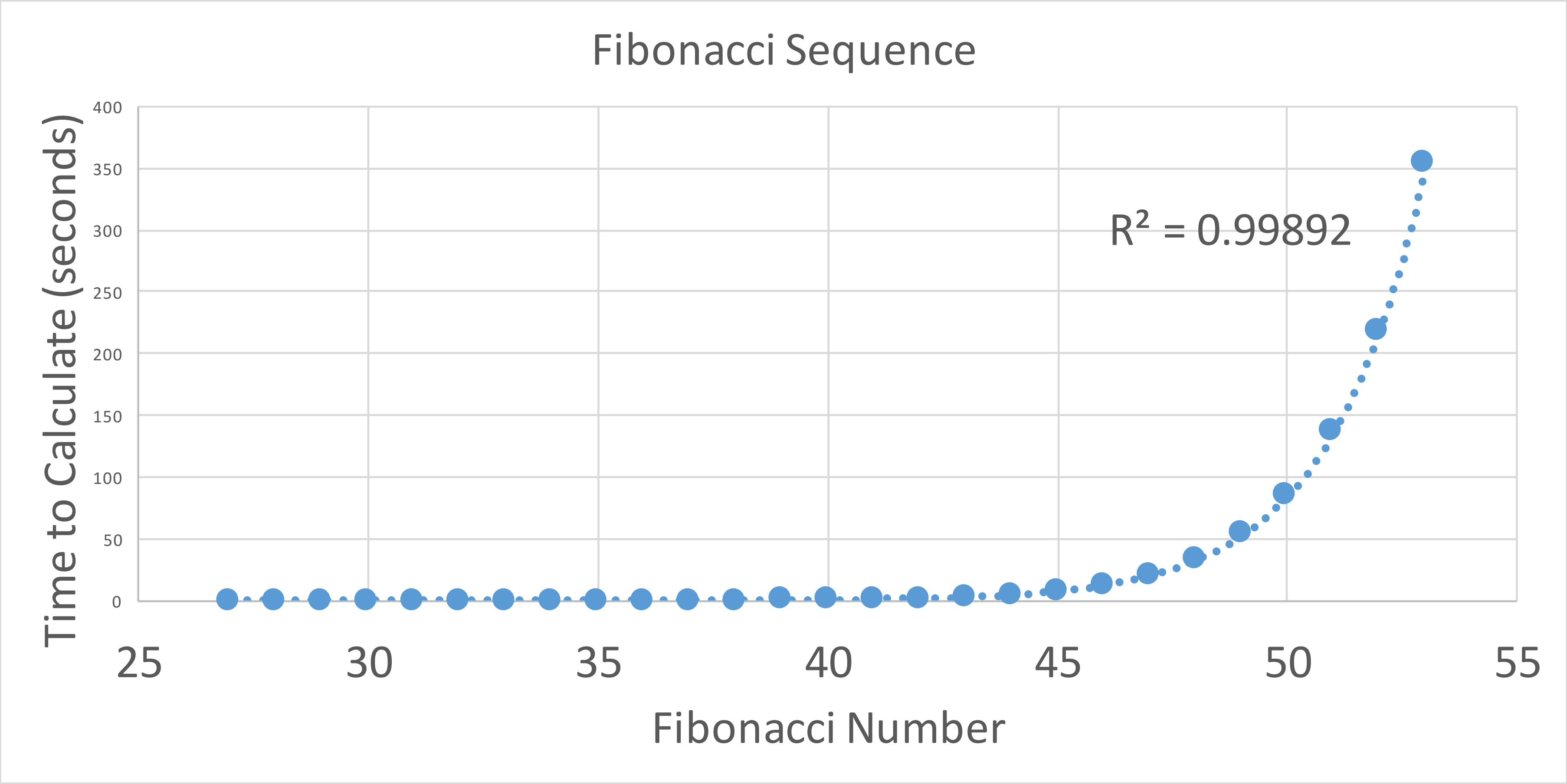
fib(4) fib(3) fib(3) fib(2)

fib(3) fib(2) fib(2) fib(1) fib(2) fib(1)

fib(2) fib(1)

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***

Vectors and Grids

- One of the most powerful aspects of C++ is the ability to have a "collection":

Vector Grid Map

Stack Queue Set

- We will talk about all of these as we go through CS 106X, but you will need to use the Vector and Grid classes for Fauxtoshop.

Vector

- What is it?
  - ArrayList<type>
  - A list of elements that can grow and shrink.
  - Each element has a place (or index) in the list.
  - Advanced array.
- Important Details
  - Constructor creates an empty list.
  - Bounds checks.
  - Knows its size.
  - Include “vector.h”
- Why not use arrays?

Vector

- What is it?
  - ArrayList<*type*>
  - A list of elements that can grow and shrink.
  - Each element has a place (or index) in the list.
  - Advanced array.
- Important Details
  - Constructor creates an empty list.
  - Bounds checks.
  - Knows its size.
  - Include “vector.h”
- Why not use arrays?

Creating a Vector

```
Vector<int> vec;
```

or

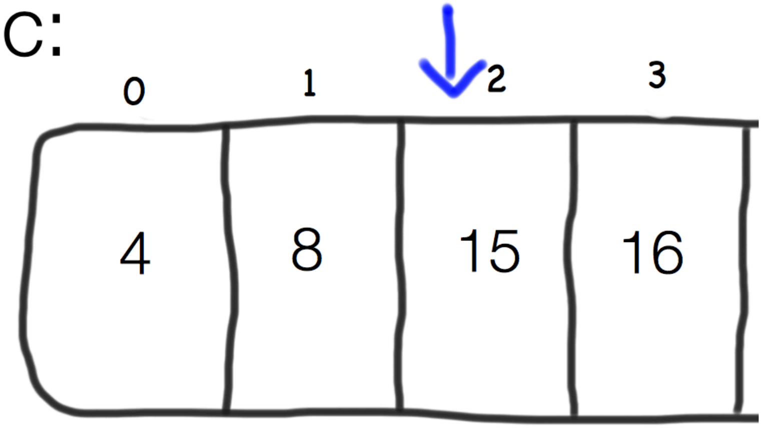
```
Vector<int> vec();
```

You must specify the type of your vector.
When a vector is created it is initially empty.

Vectors are just arrays under the hood!

```
Vector<int> magic;
magic.add(4);
magic.add(8);
magic.add(15);
magic.add(16);
cout << magic[2] << endl;
```

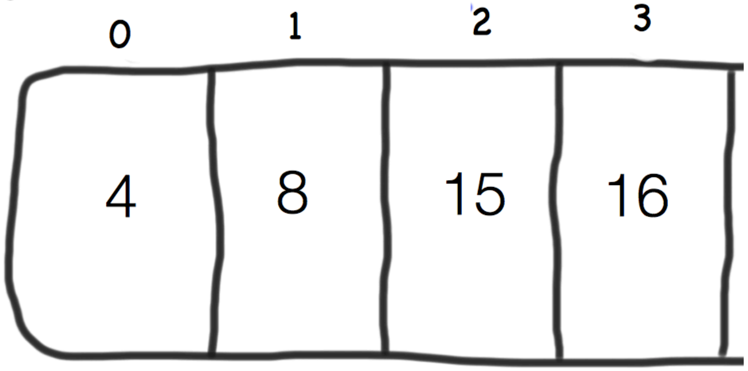
magic:



But they have useful functions, like size()

```
for(int i = 0; i < magic.size(); i++) {
   cout << magic[i];
 }
```

magic:



Output: 4

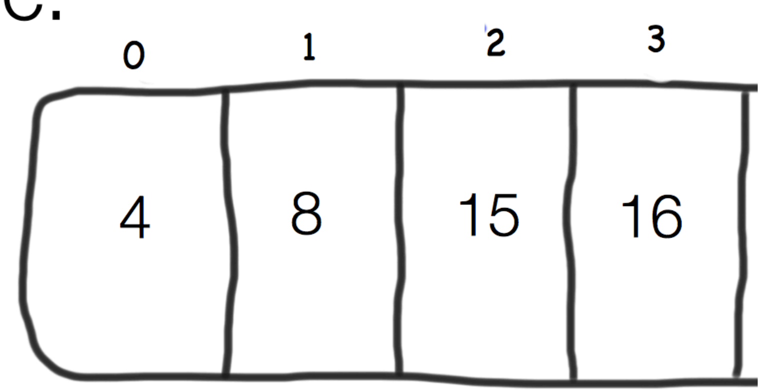
You can use "for each" loops on containers:

```
for(int value : magic) {
    cout << value << endl;
}
```

Output:

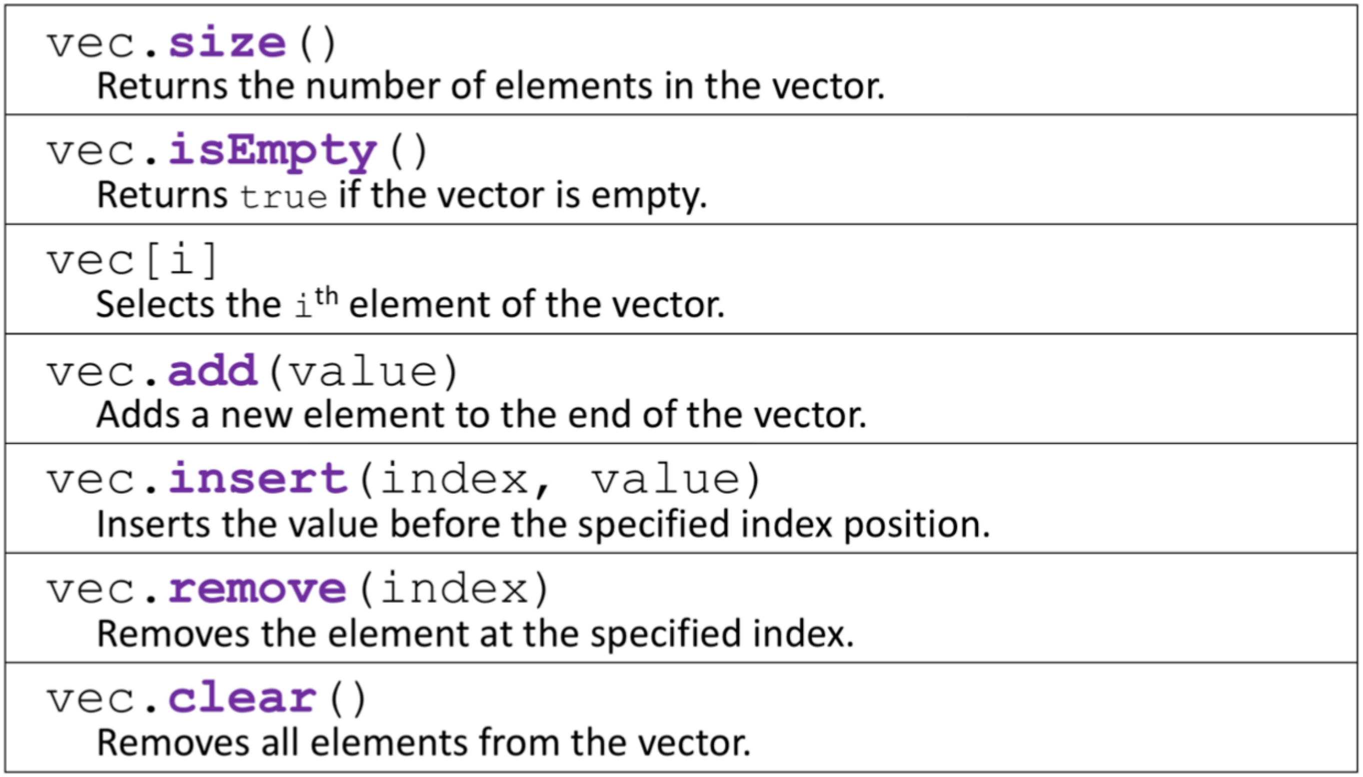
```
4
8
15
16
```

magic:



0	1	2	3
4	8	15	16

Vector Methods



`vec.size()`
Returns the number of elements in the vector.

`vec.isEmpty()`
Returns `true` if the vector is empty.

`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 position.

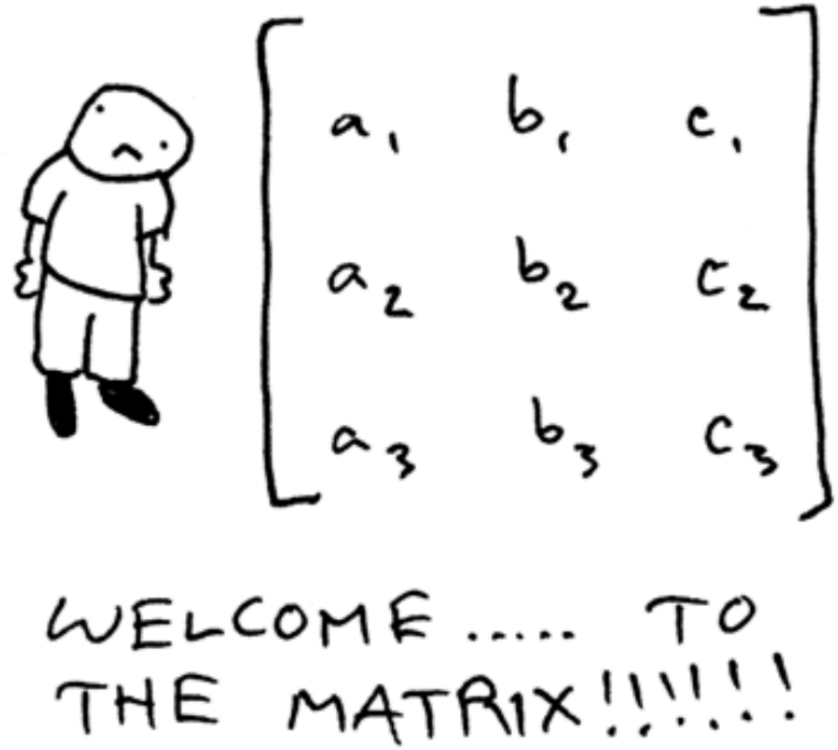
`vec.remove(index)`
Removes the element at the specified index.

`vec.clear()`
Removes all elements from the vector.

For the exhaustive list check out:
http://stanford.edu/~stepp/cppdoc/Vector-class.html

The Grid Container

Grid<type>



$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$

Grid

- What is it?
  - Advanced **2D** array.
  - Think spread sheets, game boards

- Important Details
  - Default constructor makes a grid of size 0
  - Doesn’t support “ragged right”.
  - Bounds checks
  - Knows its size.

- We *could* use a combination of Vectors to simulate a 2D matrix, but a Grid is easier!

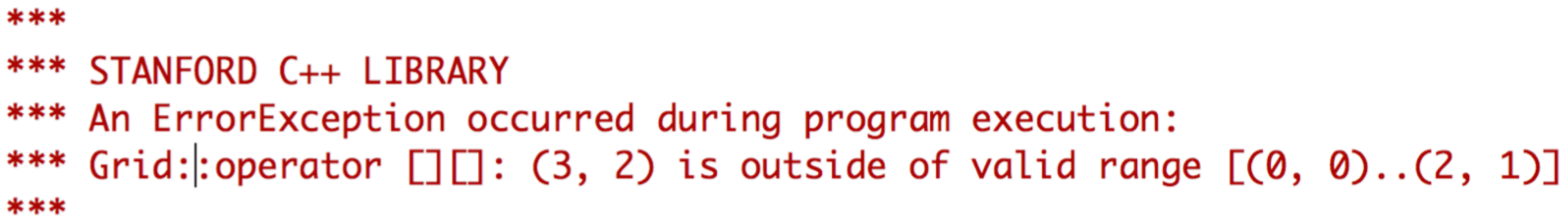
Creating a Grid

```
Grid<int> matrix(2,2);
matrix[0][0] = 42;
matrix[0][1] = 6;
matrix[1][0] = matrix[0][1];
cout << matrix.numRows() << endl;
cout << matrix[0][1] << endl;
cout << matrix[1][1] << endl;
cout << matrix[2][3] << endl;
```

Creating a Grid

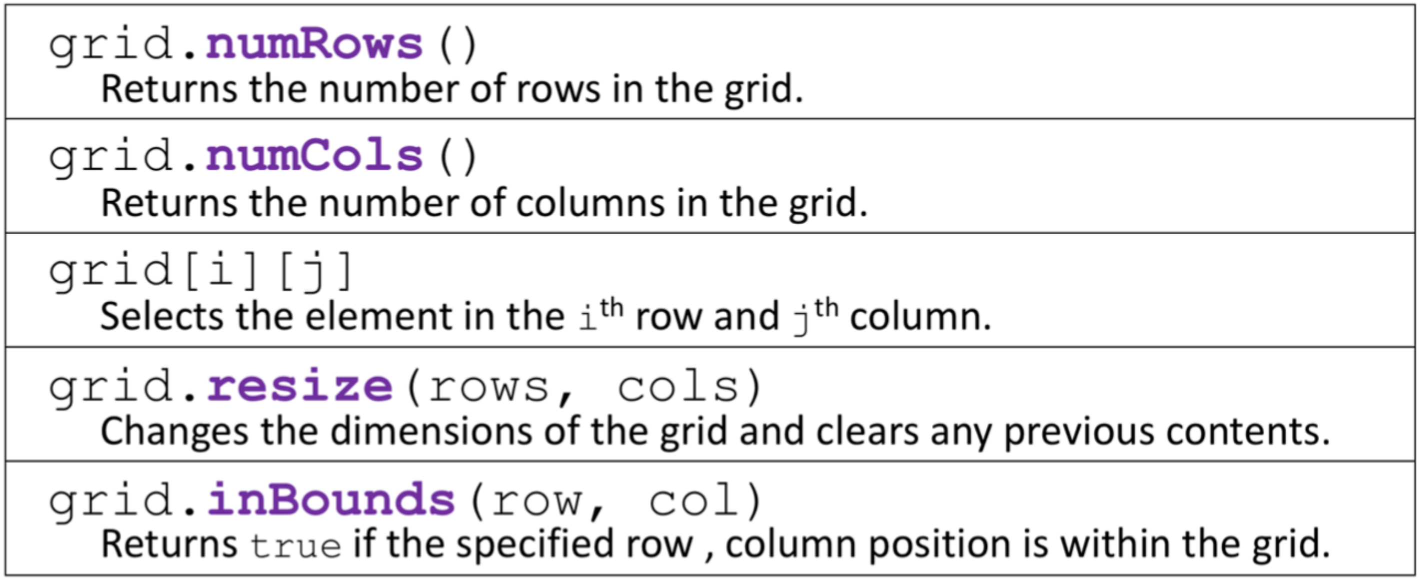
```
Grid<int> matrix(2,2);
matrix[0][0] = 42;
matrix[0][1] = 6;
matrix[1][0] = matrix[0][1];
cout << matrix.numRows() << endl
cout << matrix[0][1] << endl;
cout << matrix[1][1] << endl;
cout << matrix[2][3] << endl;
```

	0	1
0	42	6
1	6	0



```
***
*** STANFORD C++ LIBRARY
*** An ErrorException occurred during program execution:
*** Grid::operator [][]: (3, 2) is outside of valid range [(0, 0)..(2, 1)]
***
```

Grid Methods



`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 i^{th} row and j^{th} column.

`grid.resize(rows, cols)`

Changes the dimensions of the grid and clears any previous contents.

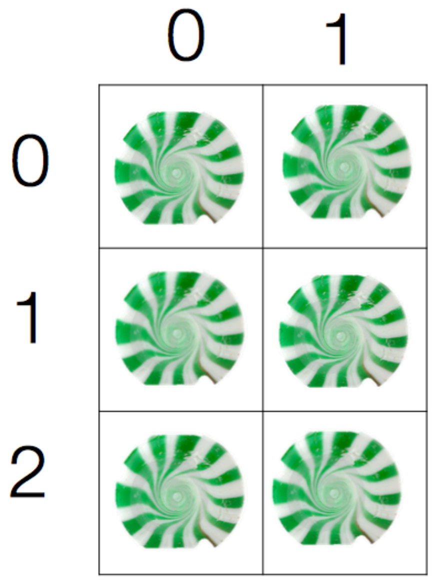
`grid.inBounds(row, col)`

Returns `true` if the specified row , column position is within the grid.

For the exhaustive list check out:
http://stanford.edu/~stepp/cppdoc/Grid-class.html

Grid Example: Traversing a Grid

```
void printGrid(Grid<Candy> & grid) {
    for(int r = 0; r < grid.numRows(); r++) {
        for(int c = 0; c < grid.numCols(); c++) {
            throwCandy(grid[r][c]);
        }
    }
}
```



Collections

1. Defined as Classes
   This means they have constructors and member functions

2. Templatized
   They have a mechanism for collecting different variable types

3. Deep copy assignment
   Often pass them by reference!

Collection Common Pitfalls

1. **`Vector numbers;`**
   Needs a type! Should be: **`Vector<int> numbers;`**

2. **`void myFunction(Grid<bool> gridParam);`**
   Two issues: (a) if you want **`gridParam`** to be changed in the calling function, you're out of luck. (b) inefficient because you have to make a copy of **`gridParam`**.

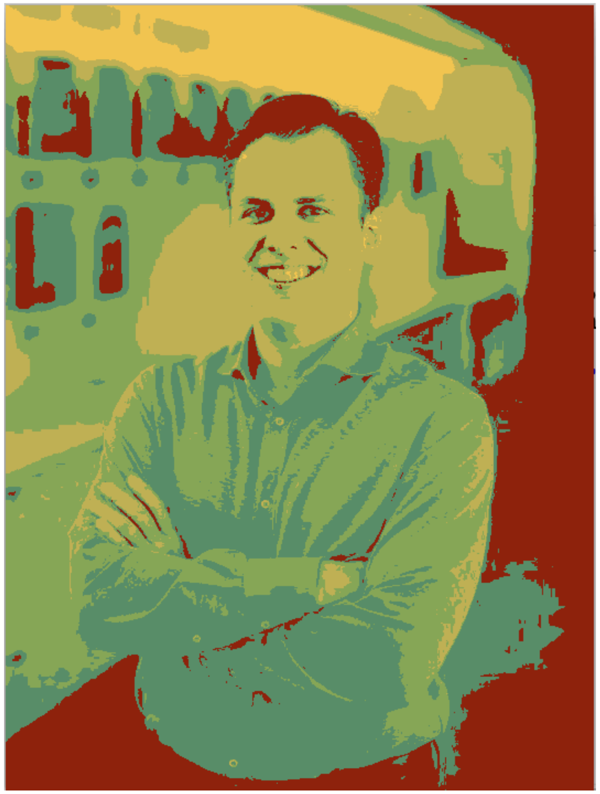
Collection Common Pitfalls

3.
```
void cout(Grid<bool> & grid) {
for(int i = 0; i < grid.numRows(); i++) {
    for(int j = 0; j < grid.numCols(); j++) {
        cout << grid[j][i];
    }
}
```

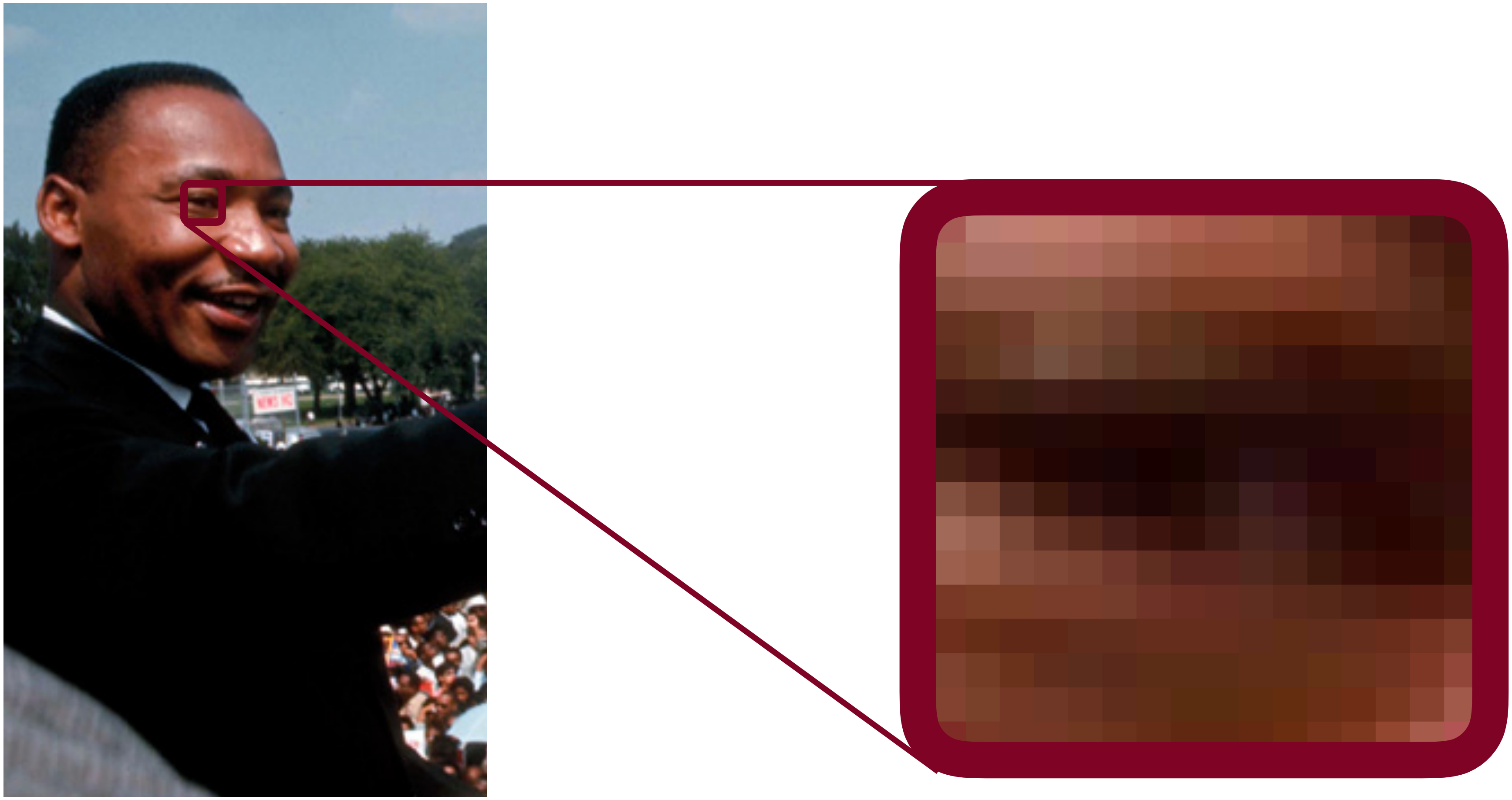
Watch your variable ordering! Better to use r for rows, c for columns.

Let's Code Instagram!

Mike Krieger, Stanford Class of 2008
Founder of Instagram



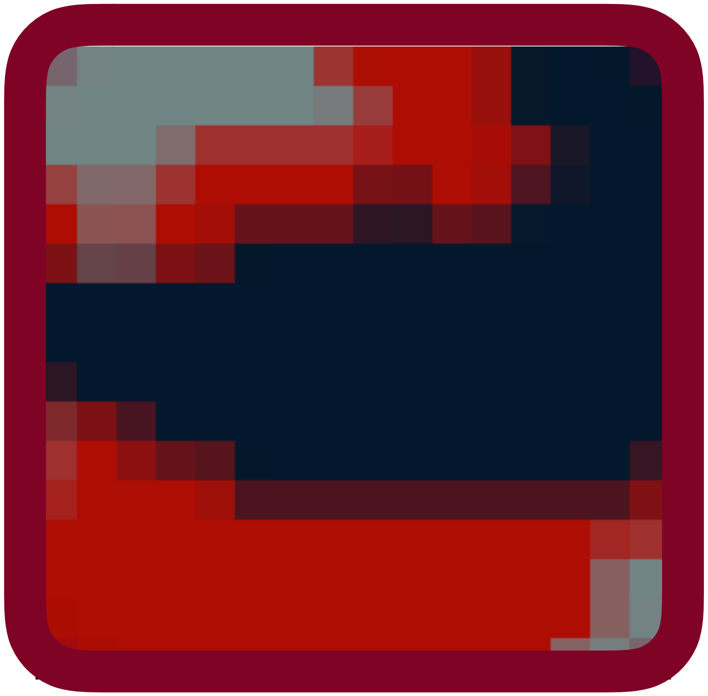
A Color is an int, and and Image is just a Grid<int>!



A Color is an int, and and Image is just a Grid<int>!

Original

Filtered



New Palette:

Let's Code!

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 (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

References and Advanced Reading (Vectors and Grids)

- **References:**

  Stanford Vector Class: http://stanford.edu/~stepp/cppdoc/Vector-class.html

  Stanford Grid Class: http://stanford.edu/~stepp/cppdoc/Grid-class.html

- **Advanced Reading:**
  - Standard Template Library vector class (some different functions!): http://www.cplusplus.com/reference/vector/vector/
  - Adobe Photoshop on Wikipedia: https://en.wikipedia.org/wiki/Adobe_Photoshop