CS 106X, Lecture 4
Vectors and Big-O

reading:

Programming Abstractions in C++, Chapter 5.1, 10

Plan For Today

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

Plan For Today

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

What is a stream?

- An *input stream* lets you get data from a source (like user input, a file, a webpage, etc.) and read it in your program.
- An *output stream* lets you take data *from* your program and output it to a source (like the console, a file, etc.).

Reading In A File

1. Open the file for reading
2. Read the file, one chunk at a time
3. Close the file

Reading In A File

```
The animal I really dig,
Above all others is the pig.
Pigs are noble. Pigs are clever,
Pigs are courteous. However,...
-Roald Dahl, "The Three Little Pigs"
```

```
ifstream infile;
promptUserForFile(infile, "Enter a file name: ");

string line;
while (getline(infile, line)) {
    cout << line << endl;
}
infile.close();
```

Writing To A File

```
// Open the file for writing
ofstream outfile;
promptUserForFile(outfile, "Enter a file name: ");

// Write to the file
string word = "my cool string";
int x = 3;
outfile << word << x;

// Close the file
outfile.close();
```

Generic Streams

```
void outputUserData(ostream& outputStream, string name,
int score, double health) {
    outputStream << name << endl;
    outputStream << score << endl;
    ...
}

int main() {
    ...
    outputUserData(cout, name, score, health);
    if (getYesOrNo("Save to file? ")) {
        outputUserData(outfile, name, score, health);
    }
}
```

Grid members (5.1)*

`Grid<type> name(r, c);` `Grid<type> name;`	create grid with given number of rows/cols; empty 0x0 grid if omitted
`g[r][c]` or `g.get(r, c)`	returns value at given row/col
`g.fill(value);`	set every cell to store the given value
`g.inBounds(r, c)`	returns `true` if given position is in the grid
`g.numCols()` or `g.width()`	returns number of columns
`g.numRows()` or `g.height()`	returns number of rows
`g.resize(nRows, nCols);`	resizes grid to new size, discarding old contents
`g[r][c] = value;` or `g.set(r, c, value);`	stores value at given row/col
`g.toString()`	returns a string representation of the grid such as `"{{3, 42}, {-7, 1}, {5, 19}}"`
`ostr << g`	prints, e.g. `{{3, 42}, {-7, 1}, {5, 19}}`

** (a partial list; see http://stanford.edu/~stepp/cppdoc/)*

Const parameters

- What if you want to avoid copying a large variable but don't want to change it?
- Use the **const** keyword to indicate that the parameter won't be changed
  - Usually used with strings and collections
  - Passing in a non-variable (e.g. `printString("hello")`) **does** work

```
void printString(const string& str) {
    cout << "I will print this string" << endl;
    cout << str << endl;
}

int main() {
    printString("This could be a really really long
                string");
}
```

NEW: Constants

- Use the **const** keyword with variables to make them constants at the top of your program file.

```
const int NUM_TURNS = 4;

int main() {
    ... // reference constant here
}

void otherFunc() {
    ... // reference constant here
}
```

STL vs. Stanford

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**.

- **Standard Template Library (STL)**:
  C++ built in standard library of collections.
  - `vector`, `map`, `list`, …
  - Powerful but somewhat hard to use.

- **Stanford C++ library (SPL)**:
  Custom library of collections made for use in CS 106B/X.
  - `Vector`, `Grid`, `Stack`, `Queue`, `Set`, `Map`, …
  - Similar to STL, but simpler interface and error messages.

Grid members (5.1)*

`Grid<type> name(r, c);` `Grid<type> name;`	create grid with given number of rows/cols; empty 0x0 grid if omitted
`g[r][c]` or `g.get(r, c)`	returns value at given row/col
`g.fill(value);`	set every cell to store the given value
`g.inBounds(r, c)`	returns `true` if given position is in the grid
`g.numCols()` or `g.width()`	returns number of columns
`g.numRows()` or `g.height()`	returns number of rows
`g.resize(nRows, nCols);`	resizes grid to new size, discarding old contents
`g[r][c] = value;` or `g.set(r, c, value);`	stores value at given row/col
`g.toString()`	returns a string representation of the grid
`ostr << g`	Prints a string representation of the grid

* *(a partial list; see http://stanford.edu/~stepp/cppdoc/)*

SparseGrid members*

`SparseGrid<type> name(r, c);` `SparseGrid<type> name;`	create grid with given number of rows/cols; empty 0x0 grid if omitted
`g[r][c]` or `g.get(r, c)`	returns value at given row/col
`g.fill(value);`	set every cell to store the given value
`g.inBounds(r, c)`	returns `true` if given position is in the grid
`g.numCols()` or `g.width()`	returns number of columns
`g.numRows()` or `g.height()`	returns number of rows
`g.resize(nRows, nCols);`	resizes grid to new size, discarding old contents
`g[r][c] = value;` or `g.set(r, c, value);`	stores value at given row/col
`g.toString()`	returns a string representation of the grid.
`ostr << g`	Prints a string representation of the grid.

* *(a partial list; see http://stanford.edu/~stepp/cppdoc/)*

Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection can do, not *how* it does it.
  - We could say that both `Grid` and `SparseGrid` implement the operations of the *abstract data type* called "**grid**".
    - other examples of ADTs: stack, queue, set, map, graph

- We don't always know exactly how a given collection is implemented internally, and we don't need to.
  - We just need to understand the idea of the collection and what operations it can perform.

Plan For Today

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

Vectors (Lists)

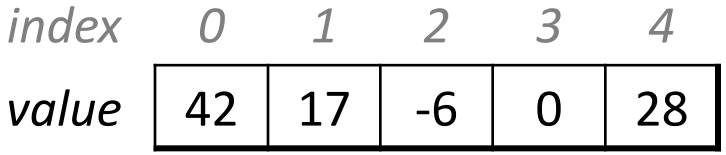
```
#include "vector.h"
```

- **vector** (aka **list**): a collection of elements with 0-based **indexes**
  - like a dynamically-resizing array (Java `ArrayList` or Python `list`)
  - Include the type of elements in the <> brackets

```
// initialize a vector containing 5 integers
//          index   0   1   2   3   4
Vector<int> nums {42, 17, -6,  0, 28};

Vector<string> names;          // {}
names.add("Nick");           // {"Nick"}
names.add("Zach");         // {"Nick", "Zach"}
names.insert(0, "Ed");     // {"Ed", "Nick", "Zach"}
```

Why not arrays?

```
// actual arrays in C++ are mostly awful
int nums[5] {42, 17, -6,  0, 28};
```



- Arrays have fixed **size** and cannot be easily resized.
  - In C++, an array doesn't even *know* its size. (no `.length` field)
- C++ lets you index out of the array **bounds** (garbage memory) *without* necessarily crashing or warning.
- An array does not support many **operations** that you'd want:
  - inserting/deleting elements into the front/middle/back of the array, reversing, sorting the elements, searching for a given value ...

Vector members (5.1)

`v.add(value);` or `v += value;` or `v += v1, v2, ..., vN;`	appends value(s) at end of vector
`v.clear();`	removes all elements
`v[i]` or `v.get(i)`	returns the value at given index
`v.insert(i, value);`	inserts given value just before the given index, shifting subsequent values to the right
`v.isEmpty()`	returns `true` if the vector contains no elements
`v.remove(i);`	removes/returns value at given index, shifting subsequent values to the left
`v[i] = value;` or `v.set(i, value);`	replaces value at given index
`v.subList(start, length)`	returns new vector of sub-range of indexes
`v.size()`	returns the number of elements in vector
`v.toString()`	returns a string representation of the vector such as `"{3, 42, -7, 15}"`
`ostr << v`	prints v to given output stream (e.g. `cout << v`)

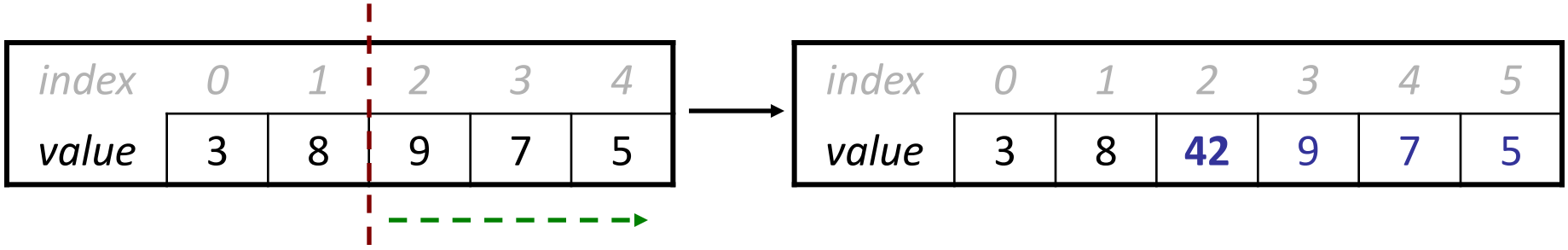
Iterating over a vector

```
Vector<string> names {"Ed", "Hal", "Sue"};

for (int i = 0; i < names.size(); i++) {
    cout << names[i] << endl;   // for loop
}
for (int i = names.size() - 1; i >= 0; i--) {
    cout << names[i] << endl;   // for loop, backward
}

for (string name : names) {
    cout << name << endl;     // "for-each" loop
}
for (string& name : names) {
    name += "!";              // "for-each" by reference
}
cout << names << endl;        // {"Ed!", "Hal!", "Sue!"}
```

Vector insert/remove

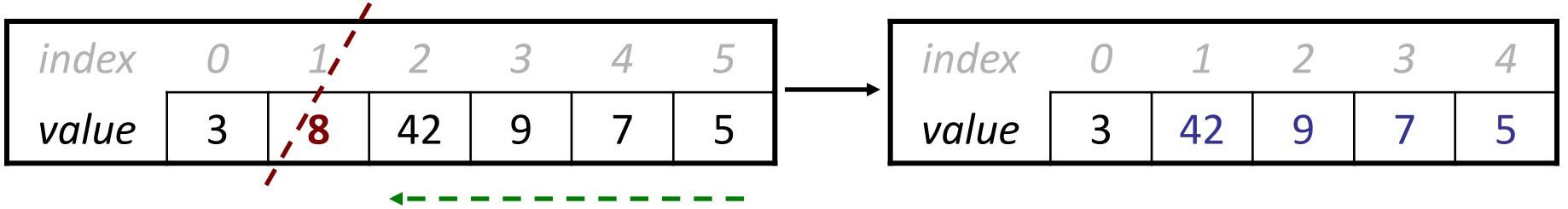
```
v.insert(2, 42);
```

- shift elements right to make room for the new element




```
v.remove(1);
```

- shift elements left to cover the space left by the removed element



(These operations are slower the more elements they need to shift.)

Exercise

countInRange
removeAll

- Write a function **`countInRange`** that accepts a vector of integers along with a min and max integer as parameters, and returns the number of elements in the vector within that range inclusive.
  - Example: if vector v stores:

    `{28, 1, 17, 4, 41, 9, 59, 8, 31, 30, 25}`
  - Then the call of `countInRange(v, 10, 30)` returns `4`.

Exercise solution

```
int countInRange(const Vector<int>& v, int min, int max) {
    int count = 0;
    for (int i = 0; i < v.size(); i++) {
        if (v[i] >= min && v[i] <= max) {
            count++;
        }
    }
    return count;
}
```

Exercise 2

countInRange
removeAll

- Write a function **`removeAll`** that accepts a vector of strings along with an element value string as parameters, and modifies the vector to remove *all* occurrences of that string.
  - Example: removing all occurrences of "b" from `{a, b, c, b, b, a, b}` yields `{a, c, a}`.

Exercise solution

```
void removeAll(Vector<string>& v, string value) {
    for (int i = v.size() - 1; i >= 0; i--) {
        if (v[i] == value) {
            v.remove(i);
        }
    }
}
```

Nested vectors

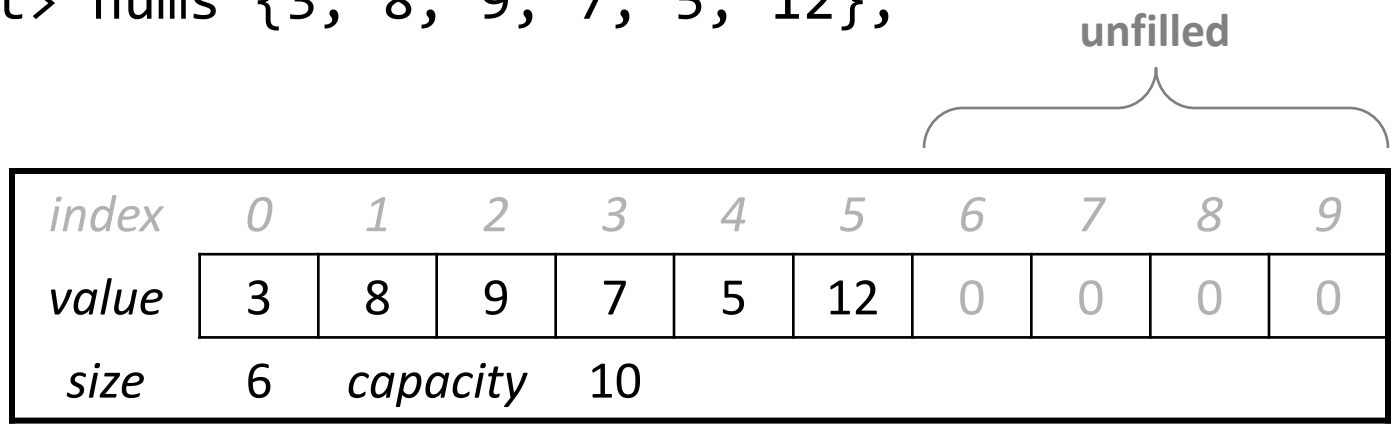
- Collections can contain other collections.
  - *How does the following code contrast with using a `Grid`?*

```
Vector<int> row1 {1};
Vector<int> row2 {2, 3};
Vector<int> row3 {4, 5, 6};
Vector<Vector<int> > vv;
vv.add(row1);
vv.add(row2);
vv.add(row3);
cout << vv << endl;          // {{1}, {2, 3}, {4, 5, 6}}
cout << vv[1][1] << endl;    // 3

// quicker initialization
Vector<Vector<int> > vv {
    {1}, {2, 3}, {4, 5, 6}
};
```

Inside a Vector

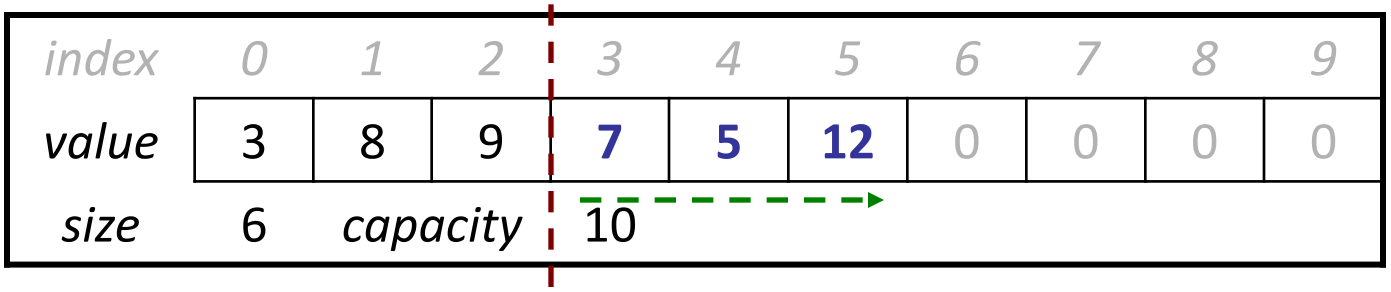
- A `Vector` is implemented using an **array** of values.
  - The vector also stores its **size** and array **capacity**.
  - Array is larger than data so there is room to add elements later *(an "unfilled array")*

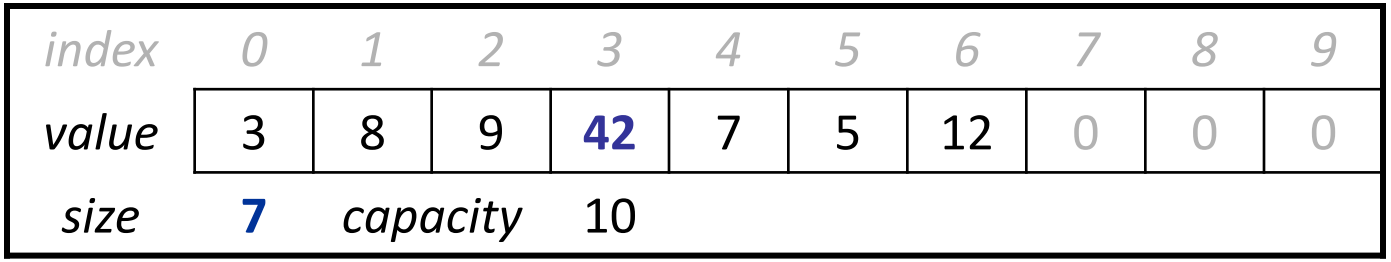
```
Vector<int> nums {3, 8, 9, 7, 5, 12};
```



Vector insert

- How do you insert in the middle of a vector? `v.insert(3, 42);`
  - **shift** elements right to make room for the new element
  - increment *size*





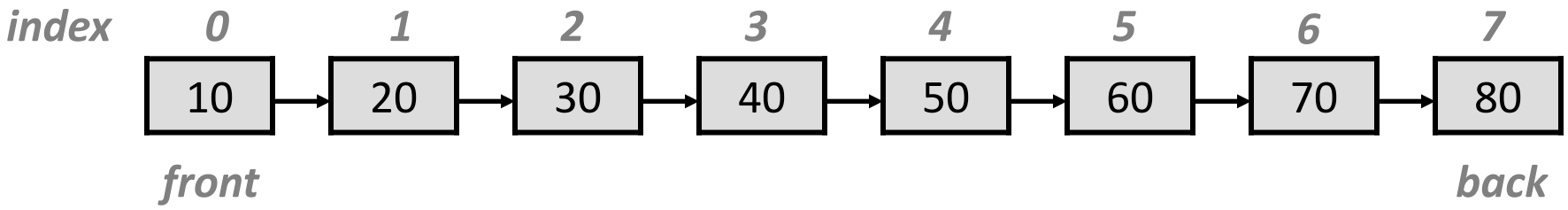
  - *Observation: Insert/remove at the front is slow.*
    - *Runtime is related to the size of the vector (number of elements to shift).*

LinkedList class

- Class **`LinkedList`** provides the same functionality as `Vector`.

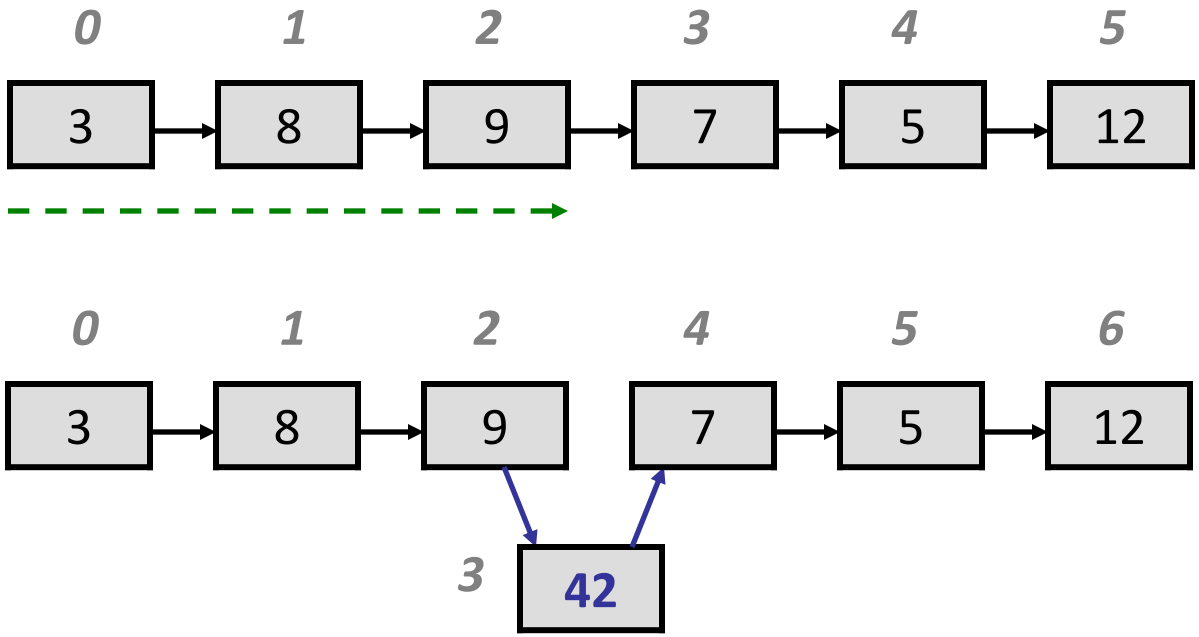
```
LinkedList<int> list;
for (int i = 1; i <= 8; i++) {
    list.add(10 * i);   // {10, 20, 30, 40, 50, 60, 70, 80}
}
```

- **linked list:** Made of *nodes*, each storing a value and link to 'next' node.
  - Internally the list knows its **front** node only (sometimes **back** too), but can go to 'next' repeatedly to reach other nodes.



LinkedList insert

- How do you insert in the middle of a linked list? `l.insert(3, 42);`
  - start at the front, walk to the location of the new element
  - add a new node containing the new element (no "shifting")

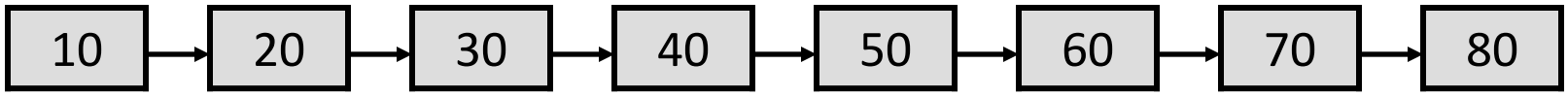


  - What operations are slow/fast for a linked list to perform?

Vector vs LinkedList

- **Q:** Which is faster? **A.** Vector **B.** LinkedList **C.** about the same

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	10	20	30	40	50	60	70	80	0	0
size	8	*capacity*	10							



- removing from the front?
- removing from the back?
- inserting in the middle?
- printing the entire contents of the list?
- filtering out all occurrences of a particular value?

Plan For Today

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

Announcements

- Discussion Section Signups
  - On-time signups have closed; results posted by tomorrow 5PM
  - Sections start Wed.!
  - Late section signups open tomorrow 5PM
- Assignment 1
  - Due Friday at 11AM
  - Use Piazza, LaIR, Office Hours, etc. if you have questions
- Lecture participation
  - If you enrolled recently, you'll be emailed today with your assignment

AfroTech

Stanford may be sponsoring students to the AfroTech conference.

If you're interested in attending, fill out this form!

https://goo.gl/forms/78m3izBgwEF4iBZy2

she++


SHEPLUSPLUS.COM
JOIN US
IN rebranding WHAT IT MEANS TO BE A TECHNOLOGIST.
SHE ++
2018.2019
apply at HTTP://WWW.TINYURL.COM/SHEPLUSPLUS18
>>>>> due sunday, october 7 at 11:59 pm
message us on Facebook with any questions

Brown Institute Showcase

Gates Computer Science Building
AT&T Patio and Lawn

Check out how our Magic Grant teams and Brown Fellows spent the last year developing groundbreaking media technologies and producing award-winning stories.

Plan For Today

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

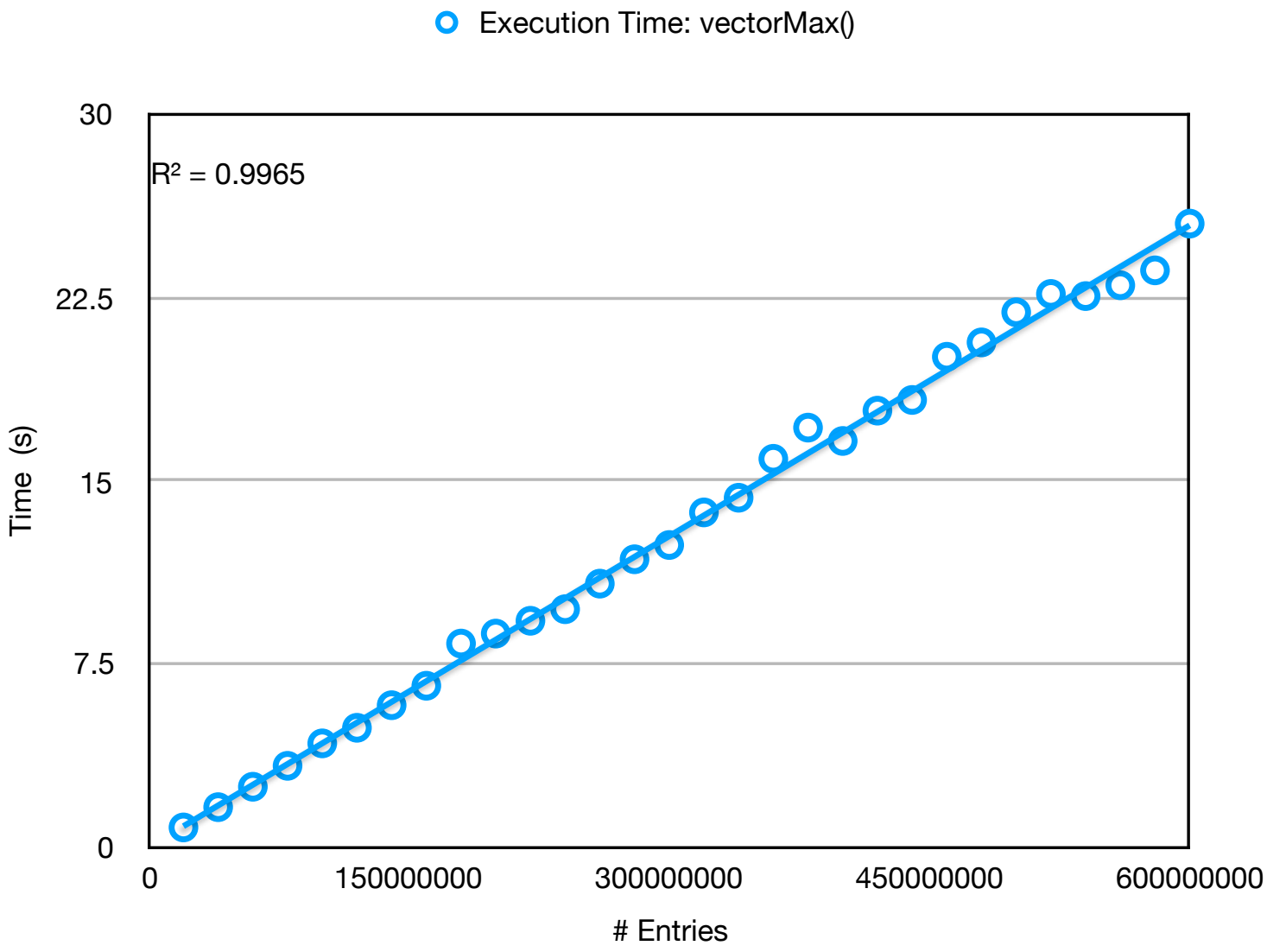
Algorithmic Efficiency

- We would like a way to measure how "fast" our code is.
  - **Seconds?** *This changes on each computer/chip!*
  - **Exact # Operations?** Hard to measure, maybe unnecessarily precise.
- To measure algorithmic efficiency, we need to narrow down why efficiency is important in the first place.
- What most significantly impacts an algorithm's speed? **Data size**
  - Most algorithms run fast when processing little data. But at Facebook, difference between 1 hr. and 1 day for machine learning matters!
  - Fun fact: Facebook scanned ~105TB of data per hour….6 years ago!
- Let's develop a way to measure an algorithm's speed that depends on the amount of data being processed.

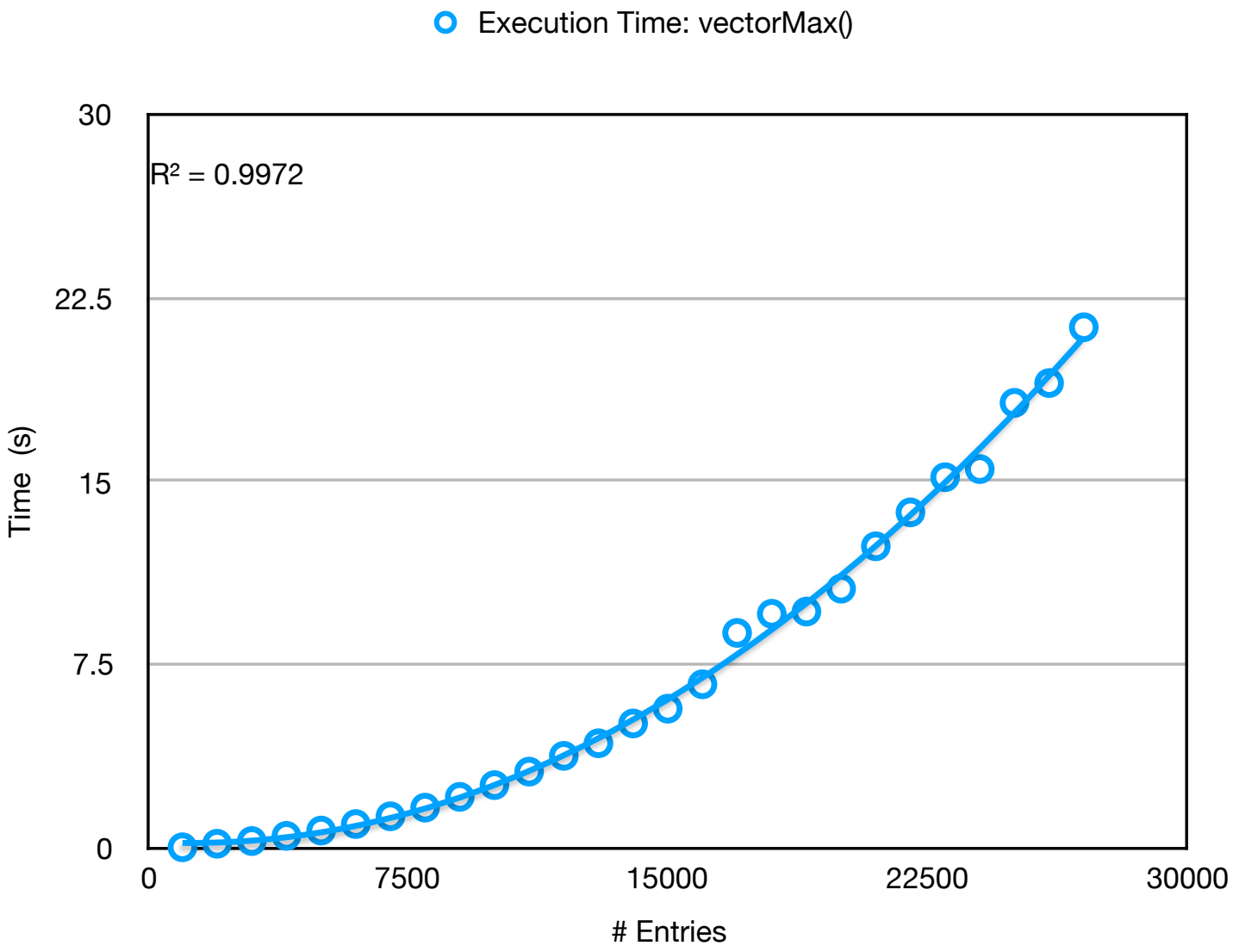
Example: vectorMax

```
int vectorMax(const Vector<int>& v) {
    int currentMax = v[0];
    for (int element : v) {
        if (currentMax < element) {
            currentMax = element;
        }
    }

    return currentMax;
}
```

vectorMax()

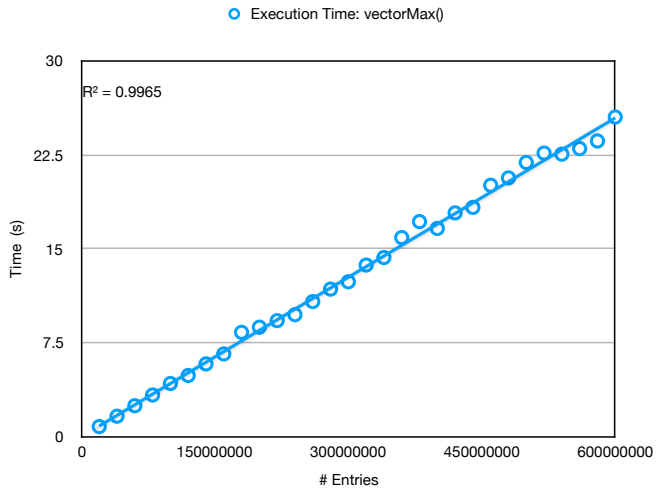

Execution Time: vectorMax()
R² = 0.9965
30
22.5
15
7.5
0
Time (s)
0
150000000
300000000
450000000
600000000
Entries

Inefficient vectorMax()



Exploring Runtime

- The runtime of the first implementation seems to be linearly proportional to the data size. So **2x more data means 2x longer to run.**



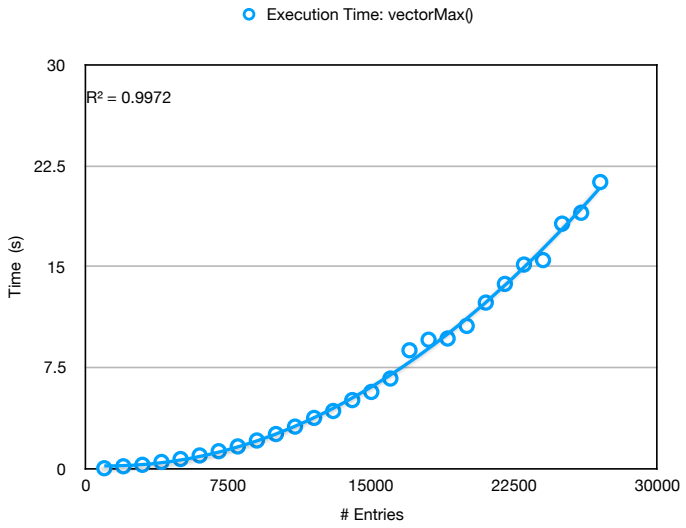

```
int vectorMax(const Vector<int>& v) {
    int currentMax = v[0];
    for (int element : v) {
        if (currentMax < element) {
            currentMax = element;
        }
    }

    return currentMax;
}
```

For each element, this loop will execute once. Doubling the data means that this loop will execute twice as many times.

We say this runtime is O(N): "on the order of N operations".

Exploring Runtime

- The runtime of the second implementation seems to be quadratically (^{2}) proportional to the data size. So **2x more data means 4x longer to run.**




```
int vectorMaxQuadratic(const Vector<int>& v) {
    Vector<int> copy(v);
    selectionSort(copy);
    return copy[copy.size() - 1];
}
```

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Exploring Runtime

```
void selectionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        // walk across the array looking for the smallest value
        int smallestIndex = i;
        for (int j = i+1; j < v.size(); j++) {
            if (v[j] < v[smallestIndex]) {
                smallestIndex = j;
            }
        }
        // swap v[i] with v[smallestIndex]
        swap(v, i, smallestIndex);
    }
}
```

We say this runtime is $O(N^2)$.

For each element, the first loop will execute once. Additionally, for each element, *all inner loops* will execute once. Doubling the data means that the outer loop will execute twice as many times, *and* each inner loop will execute twice as many times, totaling 4x more time.

Exploring Runtime

```
int vectorMaxQuadratic(const Vector<int>& v) {
    Vector<int> copy(v);
    selectionSort(copy);

    // Paranoid check to see if it's correct
    int max = copy[copy.size() - 1];
    bool isCorrect = true;
    for (int i = 0; i < copy.size(); i++) {
        if (copy[i] > max) {
            isCorrect = false;
        }
    }

    if (isCorrect) {
        return max;
    } else {
        error("Internal error");
    }
}
```

We said this runtime is $O(N^2)$.

Due to this loop, this runtime is $O(N)$.

Technically the total runtime is $O(N^2 + N)$, but we say that this is just $O(N^2)$ because as N gets extremely large, this is the only term that matters.

Exploring Runtime

```
int vectorMaxQuadratic(const Vector<int>& v) {
    // Show the user we are doing something
    for (int i = 0; i < 100000000; i++) {
        cout << "Calculating..." << endl;
    }

    Vector<int> copy(v);
    selectionSort(copy);
    return copy[copy.size() - 1];
}
```

We say this runtime is constant O(1) relative to N.

We said this runtime is $O(N^2)$.

We could say the total runtime is $O(N^2 + 100M)$, but we don't care about constants (even large ones). If N = 100T, then it's insignificant! *We just care about how the algorithm performs relative to the data size.* So it's $O(N^2)$.

Runtime Rules

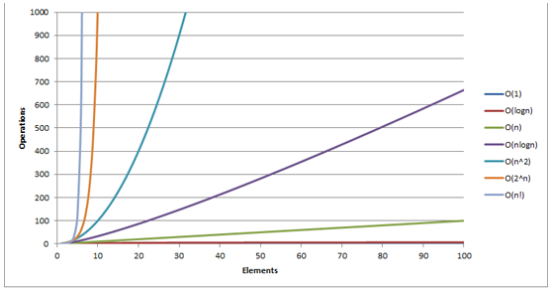
- If you have terms adding together, drop all but the **most dominant** term.
  - Code at the same indentation level adds
- We only care about terms that depend on the data size – constant terms (O(1)) are insignificant.
- Multiplying terms together *does* matter (e.g. selection sort).
  - E.g. a loop over all the data, and inside that loop you loop again each time.
  - Nested code multiplies
- Work from the innermost indented code out
- Realize that some code statements are more costly than others
  - It takes $O(N^2)$ time to call a function with runtime $O(N^2)$, even though calling that function is only one line of code

What is the Big O?

```
int sum = 0;
for (int i = 1; i < 100000; i++) {
    for (int j = 1; j <= i; j++;) {
        for (int k = 1; k <= N; k++) {
            sum++;
        }
     }
}
Vector<int> v;
for (int x = 1; x <= N; x += 2) {
    v.insert(0, x);
}
cout << v << endl;
```

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size "N".



Class	Big-Oh	If you double N, ...	Example
constant	O(1)	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	O(N)	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	11 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
quad-linear	$O(N^2 \log_2 N)$	slightly more than quadruple	8 min
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years
factorial	O(N!)	multiplies drastically	10^{200} years

More Examples

```
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!

More Examples

Practice: what is Big-O for this function?

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

More Examples

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

More Examples

Practice: what is Big-O for this function?

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

More Examples

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: $n \log n$

Preparing for the Worst

```
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;
}
```

Preparing for the Worst

```
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;
}
```

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

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

Preparing for the Worst

```
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.

Preparing for the Worst

- In general, we always examine the *worst case* runtime.
- Sometimes, when explicitly mentioned, we examine the best case or average case. But we always assume we are discussing worst case unless explicitly mentioning otherwise.

Vector efficiency

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	42	7	5	12	0	0	0
size	7	capacity		10						

Member	Big-Oh *
`v.add(value);`	O(1)
`v.get(i)` or `v[i]`	O(1)
`v.insert(i, value);`	O(N)
`v.remove(i);`	O(N)
`v.set(i, val)` or `v[i]=`	O(1)
`v.size()`, `v.isEmpty()`	O(1)
`v.toString()`, `cout << v`	O(N)

– Functions that must *loop over* or *shift* the internal array are slow.

* *average-case* runtime

Big-Oh question

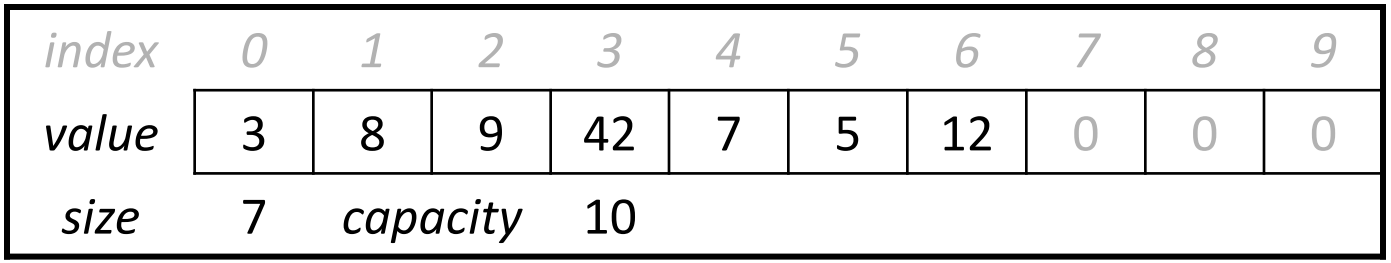
```
Vector<int> v1;                          // 1)
for (int i = 0; i < N; i++) {
    v1.add(i);
}
for (int i = 0; i < N; i++) {
    v1.remove(v1.size() - 1);
}
```

```
Vector<int> v2;                          // 2)
for (int i = 1; i <= N; i++) {
    v2.insert(0, i);   // insert value i at index 0, twice
    v2.insert(0, i);
}
v2.clear();
```

Q: In which complexity class does each piece of code above belong?

A. O(log N)
B. O(N)
C. O(N log N)
D. $O(N^2)$
E. $O(N^3)$

Vector/LinkedList runtime



Member	Vector	LinkedList
`add(value);`	O(1)	O(1)
`get(i) or [i]`	**O(1)**	**O(N)***
`insert(i, value);`	**O(N)***	**O(N)***
`remove(i);`	**O(N)***	**O(N)***
`set(i, val) or [i]=`	**O(1)**	**O(N)***
`size(), isEmpty()`	O(1)	O(1)
`toString(), cout << v`	**O(N)**	**O(N)**

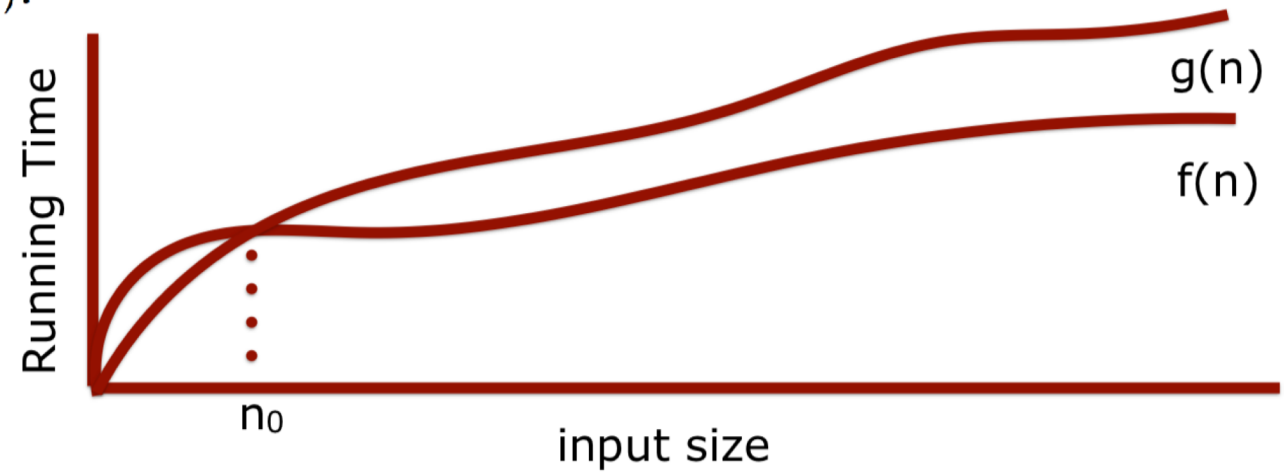
* *average-case* runtime;

Vector = O(1) at end, worst at front;

LinkedList = O(1) at front and end, worst in middle

The Math Behind It

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* $g(n)$."



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"!)

What is N?

- N represents the “data size”: this could mean:
  - The number of elements in a data structure
  - The number passed in to a function representing # times to loop
- It depends on what the algorithm is (but there is likely only 1 clear choice)

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 …

Exponential Time

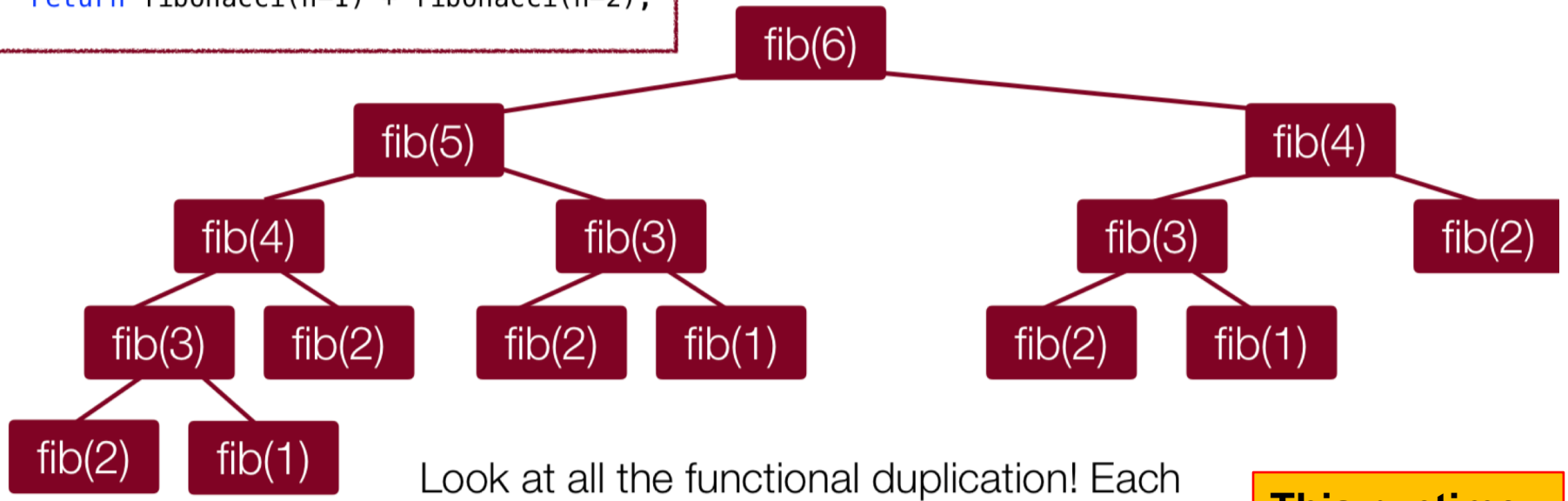
```
long fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

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

Exponential Time

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

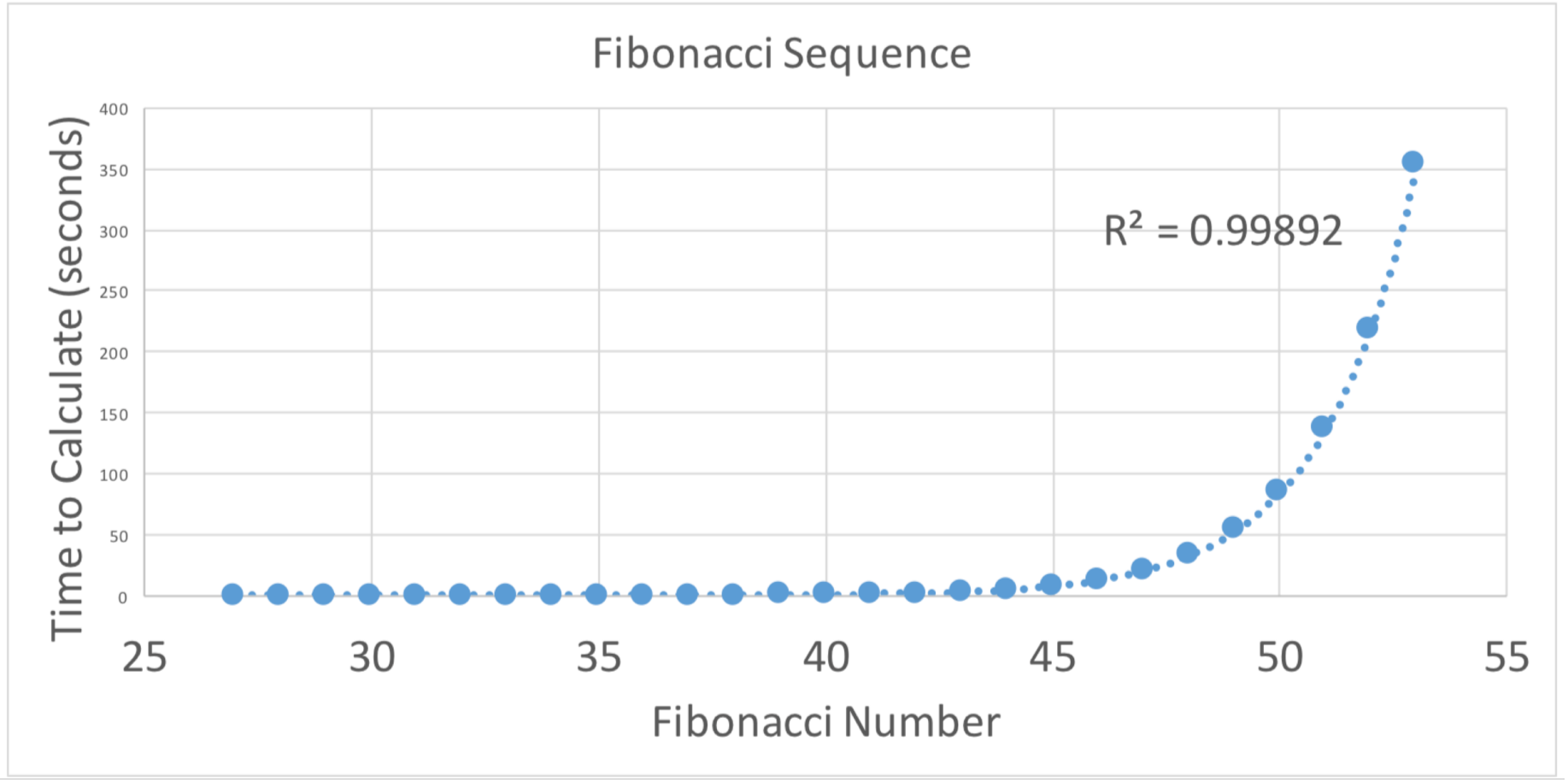
Beautiful, but a flawed algorithm! Yes, it works, but why is it flawed? Let's look at the call tree for fib(6):



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

This runtime is $O(2^N)$!

Exponential Time



Recap

- Recap: C++ Streams and Grid
- ADTs: Vector
- Announcements
- Efficiency and Big-O

Next time: ADTs: Stacks and Queues