Programming Abstractions

CS106B

Instructor: Cynthia Bailey

Head TA: Yasmine Alonso

Today's Topics

pollev.com/cs106b

- Recursion!
  - Algorithm performance analysis with Big-O
- Next time:
  - Recursion! The upcoming week is Recursion Week!
  - Like Shark Week, but more nerdy

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A**

How do we measure “faster” in Computer Science?

NOT AS SIMPLE AS YOU MIGHT THINK…

Recall our discussion of performance with the Vector add vs. Insert…

Your turn: Vector performance

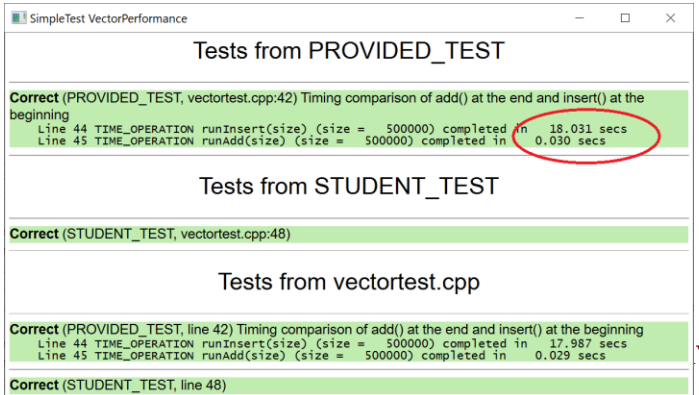
Performance analysis technique 1: **Benchmarking** (timing)

- **Answer: (D) Something else! (about 50x)**
  - In addition to analyzing the code and predicting numbe
    time the code using our Stanford 106B test system.
  - **Check the code bundle for class today for runnable vers**

```
void runInsert(int size)
{
	Vector<int> v;
	for (int i = 0; i < size; i++) {
		v.insert(0, i);
	}
}

void runAdd(int size)
{
	Vector<int> v;
	for (int i = 0; i < size; i++) {
		v.add(i);
	}
}
```

```
/* * * * * * Test Cases * * * * * */
PROVIDED_TEST("Timing comparison")
{
	int size = 500000;
	TIME_OPERATION(size, runInsert(size));
	TIME_OPERATION(size, runAdd(size));
}
```



versity

Your turn: Vector performance

Performance analysis technique 2: **Precise counting** of steps

- **Answer: (D) Something else! (about 50x)**
  - Number of times a number is written in a box
    - OPTION 1:
      - First loop iteration: 1 write
      - Next loop iteration: 2 writes … continued…
      - Formula for sum of numbers 1 to N = (N * (N + 1)) / 2
      - *(don't worry if you don't know this formula, we only expected a ballpark estimate)*
      - 100 * (100 + 1) / 2 = 10,100 / 2 = **5,050**
    - OPTION 2:
      - First loop iteration: 1 write
      - Next loop iteration: 1 write … continued…
      - **100**


Performance analysis
technique 3:
Big-O

Big-O Performance Analysis

A WAY TO COMPARE THE NUMBER OF STEPS TO RUN CODE/ALGORITHMS

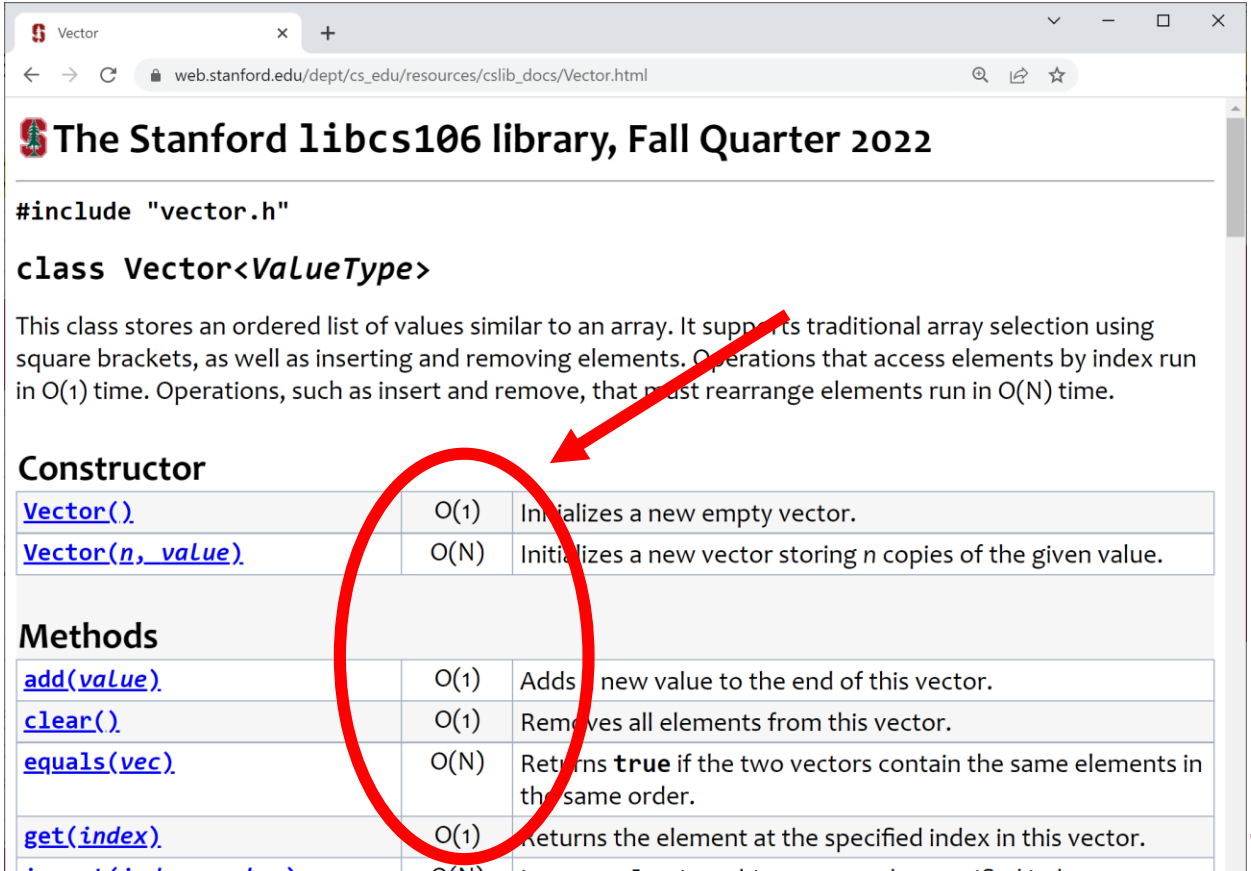
Big-O: programmers' primary algorithm performance analysis technique

- Big-O analysis in computer science is a way of imprecisely counting the number of "steps" needed to complete a task by grouping algorithms into buckets with labels like O(1), O(N), O(N^2)

- Differs from **benchmarking**:
  - Ignores how fast the computer's CPU or other hardware components are
    - *(part of the point is to analyze independent of this)*
- Differs from **precise counting**:
  - Often starts with a precise count, but then ignores "details" to just assign a broad performance category
  - Often combines different kinds of "steps" (writes, less-than < comparisons, etc) into one count, ignoring that some kinds take more time than others

- *Ignores much to focus on making broad comparisons between approaches*

Big-O analysis in computer science



Vector

web.stanford.edu/dept/cs_edu/resources/cslib_docs/Vector.html

The Stanford libcs106 library, Fall Quarter 2022

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

class Vector<*ValueType*>

This class stores an ordered list of values similar to an array. It supports traditional array selection using square brackets, as well as inserting and removing elements. Operations that access elements by index run in O(1) time. Operations, such as insert and remove, that must rearrange elements run in O(N) time.

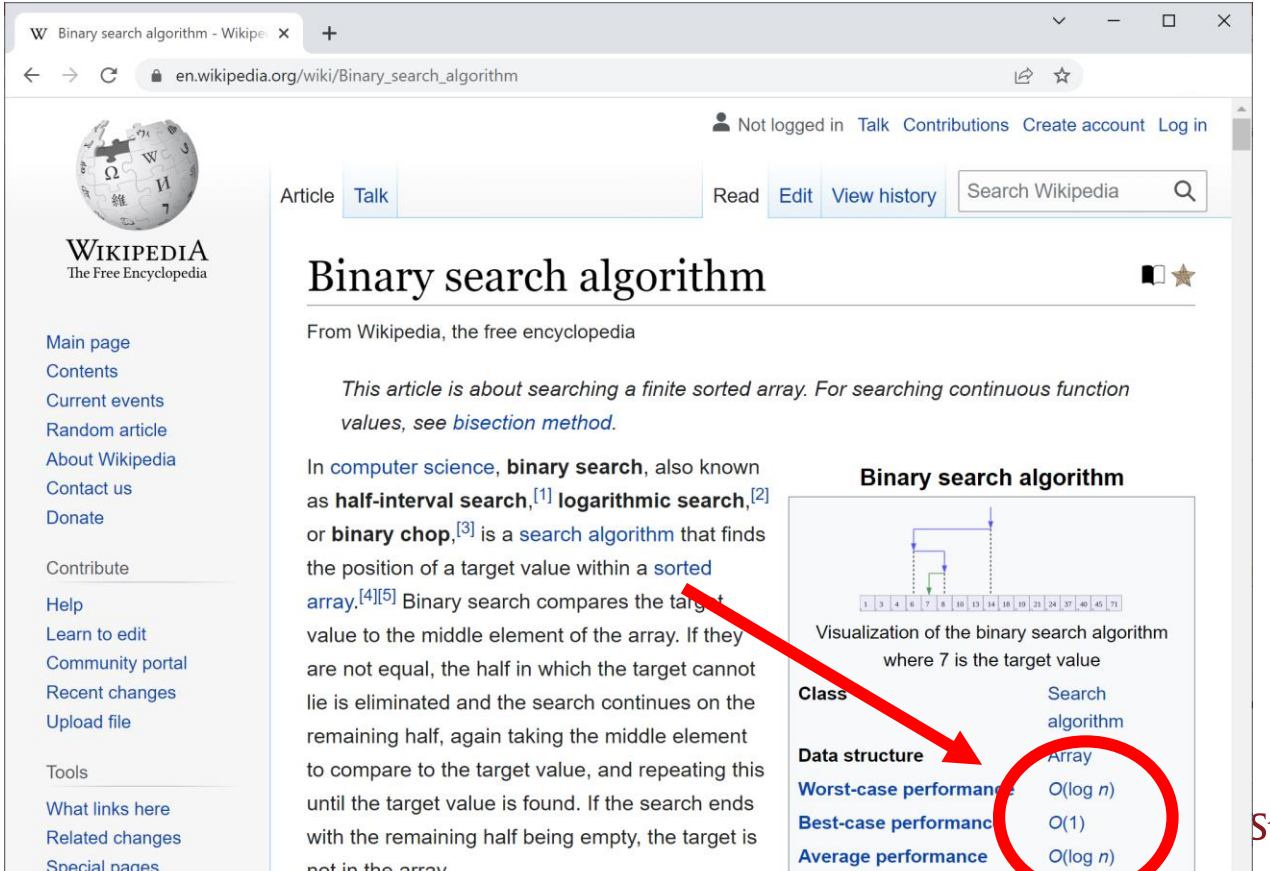
Constructor

Vector()	O(1)	Initializes a new empty vector.
Vector(*n*, *value*)	O(N)	Initializes a new vector storing *n* copies of the given value.

Methods

add(*value*)	O(1)	Adds a new value to the end of this vector.
clear()	O(1)	Removes all elements from this vector.
equals(*vec*)	O(N)	Returns **true** if the two vectors contain the same elements in the same order.
get(*index*)	O(1)	Returns the element at the specified index in this vector.

Big-O analysis in computer science



Formal definition of big-O

We say a function $f(n)$ is "big-O" of another function $g(n)$
(i.e., $f(n)$ is $O(g(n))$)

if and only if

there exist positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

c says "we don't care about constant coefficients"

n_0 says "we only care about performance on big data sets"

Formal definition of big-O

If this whole mess looks scary, don't worry!! Just ignore it and learn our simple steps to applying Big-O! ☺

We say a function $f(n)$ is "big-O" of $g(n)$ (i.e., $f(n) \in O(g(n))$) if and only if there exist positive constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

c says "we don't care about constant coefficients"

n_0 says "we only care about performance on big data sets"

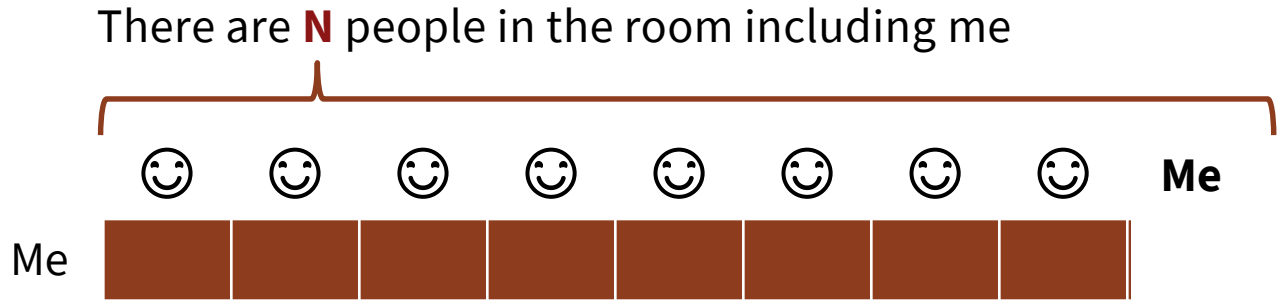
Warm-up Big-O Example

LET'S GET INTRODUCED!

Before we start, let's get introduced

Lets say I want to meet each of you today with a handshake and ***you*** tell ***me*** your name…

How many introductions need to happen?



But do I need to shake hands with myself, or tell myself my name?

Precise counting says: N-1 introductions

Putting this in Big-O terms

Big-O is a way of categorizing amount of work to be done in general terms, with a focus on:

- ***Rate of growth*** as a function of the problem size N
- What that rate looks like ***on the horizon*** (i.e., for large N)

Therefore, we don't really care about an insignificant ±1

Putting this in Big-O terms

For the first handshake problem, the rate N is important and the -1 constant is not, so a precise count of **N – 1** introductions becomes:

$$O(N-1) \rightarrow O(N)$$

Ignoring constants is a general rule!

So, similarly, if we said that each introduction **takes 3 seconds on laptopA and 5 seconds on laptopB**, benchmarking would measure the amount of time in seconds as **3(N – 1) = 3N – 3**, and **5(N – 1) = 5N – 5,** but we disregard constants:

$$O(3N-3) \rightarrow O(N)$$

$$O(5N-5) \rightarrow O(N)$$

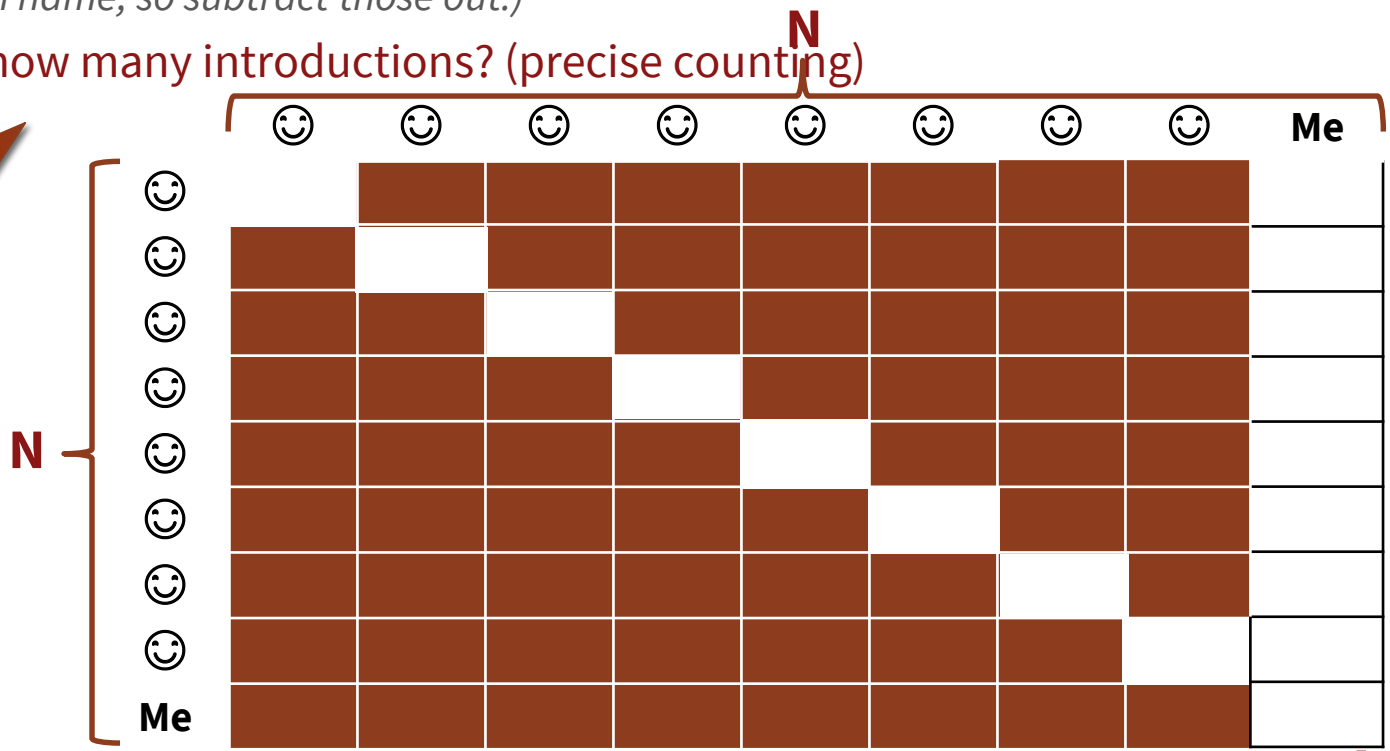
Before we start, let's get introduced

What if I not only want you to be introduced to me, but to each other also?

- *(Note: I don't need to tell anyone my name, and nobody needs to tell themselves their own name, so subtract those out.)*

Q: Now how many introductions? (precise counting)

Go to pollev.com/cs106b to respond!

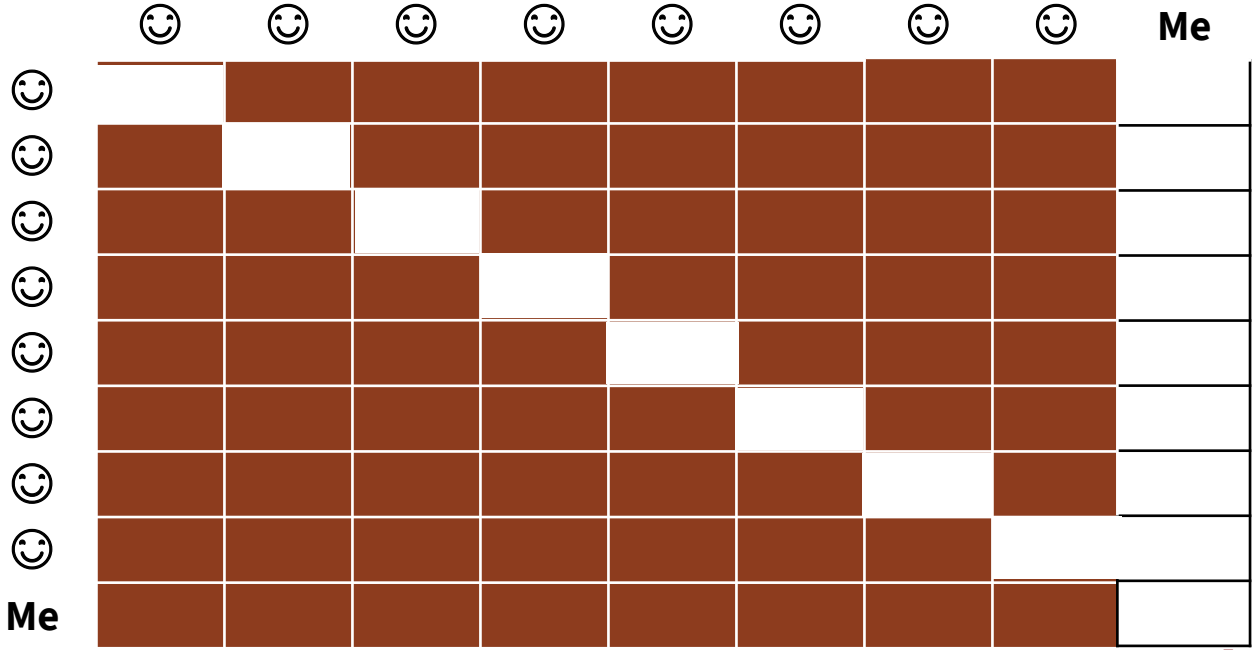


Before we start, let's get introduced

What if I not only want you to be introduced to me, but to each other also?

- *(Note: I don't need to tell anyone my name, and nobody needs to tell themselves their own name, so subtract those out.)*

Q: Now how many introductions? A: $(N - 1)^2 = N^2 - 2N + 1$



Putting this in Big-O terms

For the second handshake problem, the precise count of the number of introductions was **N^2 - 2N + 1**.

But Big-O wants us to simplify, and only focus on the biggest contributor to the rate of growth, so:

$$O(\quad N^2 - 2N + 1\ ) \rightarrow O(N^2)$$

- We cut out 2N even though N is definitely not a constant!
- **For Big-O, we only keep the largest term of the polynomial**

Big-O Procedure

IN SUMMARY

In Summary: Big-O Procedure

Bookmark or take a photo of this slide!

1. **Do a precise(ish) count**
   - *Note: as you get comfortable with Big-O, you'll learn you can do shortcuts on a true precise count, since you'll be stripping out the details anyway!*
2. **Keep only the largest term of the polynomial**
   - $O(1) < O(\log_2 N) < O(N) < O(N^2) < O(N^3) < O(N^4) \ldots < O(2^N) < O(N!)$
3. **Strip off any constant coefficient**
4. **Throw a O(__) around the result, and done!**

Big-O Example Analyzing C++ Code

THIS IS WHAT YOU'LL USUALLY BE DOING!

Revisiting Vector Insert Memory Writes, with Big-O

```
void runInsert(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.insert(0, i);
     }
}

void runAdd(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.add(i);
     }
}
```

1. **Do a precise count (of mem writes)**
   - *Last time we decided it was* **(N(N+1))/2** *and* **N** *(100 for N=100), respectively.*
2. **Keep only the largest term of the polynomial**
   - $(N(N+1))/2 = \frac{1}{2} N^2 + \frac{1}{2}N \rightarrow \frac{1}{2} N^2$
   - $N \rightarrow N$
3. **Strip off any constant coefficient**
   - $\frac{1}{2} N^2 \rightarrow N^2$
   - $N \rightarrow N$
4. **Throw a O(__) around the result, and done!**
   - **$O(N^2)$**
   - **$O(N)$**

Revisiting Vector Insert Memory Writes, with Big-O

```
void runInsert(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.insert(0, i);
     }
}

void runAdd(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.add(i);
     }
}
```

1. **Do a precise count (of mem writes)**
   - *Last time we decide… was* **(N(N+1))/2** *and* **N** *(100 for… respectively.*
2. **Keep o… polyno…**
   - (N(N…
   - N →
3. **Strip o…**
   - ½ N²
   - N →
4. **Throw … done!**
   - **O(N²)**
   - **O(N)**

NOTE!! In this example, we are focusing on **memory writes**, *since that's what we did last time.*

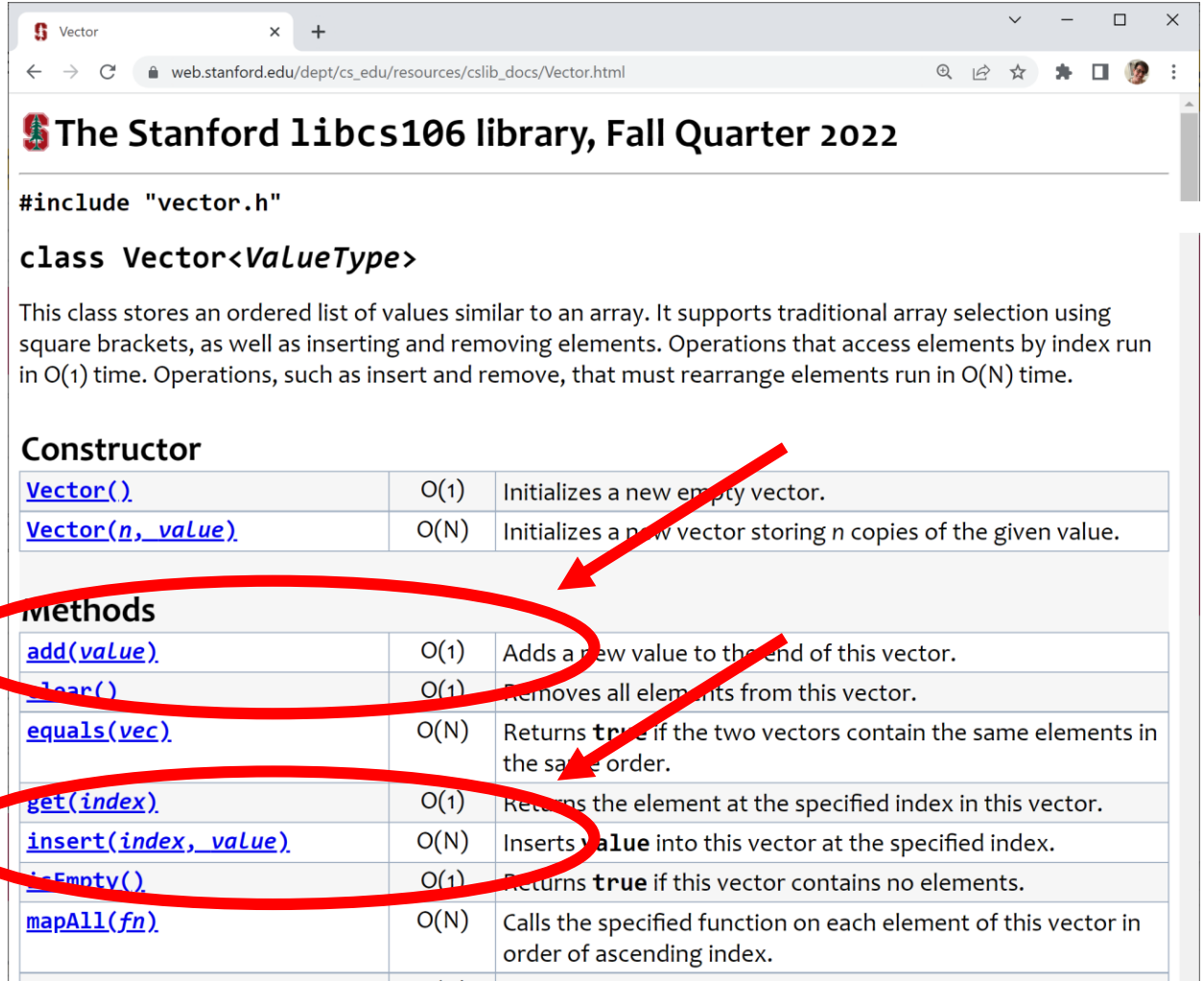
In general for Big-O, you should count **all operations** *and lines of code, not just memory writes (unless otherwise directed).*

Revisiting Vector Insert Memory Writes, with Big-O *and* Stanford Library Documentation

```
void runInsert(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.insert(0, i);
     }
}

void runAdd(int N)
{
     Vector<int> v;
     for (int i = 0; i < N; i++) {
          v.add(i);
     }
}
```

1. **Do a precise(ish) count (of mem writes)**
   - *Observe that the for-loop runs N times*
   - `insert` *or* `add` *runs once each time the for-loop runs*
   - *So our cost will be N * (cost of* `insert` *or* `add`*)*
   - ***Look up the cost of*** `insert` ***and*** `add` ***in the Stanford library documentation! O(N) and O(1)***
   - *So our costs are N*N = N^2, and N*1 = N*
2. **Keep only the largest term of the polynomial**
3. **Strip off any constant coefficient**
4. **Throw a O(__) around the result, and done!**
   - **$O(N^2)$**
   - **O(N)**


Vector
web.stanford.edu/dept/cs_edu/resources/cslib_docs/Vector.html
The Stanford libcs106 library, Fall Quarter 2022
#include "vector.h"
class Vector<ValueType>
This class stores an ordered list of values similar to an array. It supports traditional array selection using square brackets, as well as inserting and removing elements. Operations that access elements by index run in O(1) time. Operations, such as insert and remove, that must rearrange elements run in O(N) time.
Constructor
Vector() O(1) Initializes a new empty vector.
Vector(n, value) O(N) Initializes a new vector storing n copies of the given value.
Methods
add(value) O(1) Adds a new value to the end of this vector.
clear() O(1) Removes all elements from this vector.
equals(vec) O(N) Returns true if the two vectors contain the same elements in the same order.
get(index) O(1) Returns the element at the specified index in this vector.
insert(index, value) O(N) Inserts value into this vector at the specified index.
isEmpty() O(1) Returns true if this vector contains no elements.
mapAll(fn) O(N) Calls the specified function on each element of this vector in order of ascending index.

Tips: Precise(-ish) Counting for Big-O

Bookmark or take a photo of this slide!

1. Do a precise(ish) count

- Some tips for this step:
  - Lines of code that one after the other:
    - **Sum** their costs
  - Loops:
    - **Multiply** number of times the loop runs * cost of the loop body
  - Nested loops: *same loops principle applies*
    - **Multiply** number of times the outer loop runs * number of times the inner loop runs * cost of the inner loop body
  - ADT function calls:
    - **Look up** cost in Stanford Library Documentation!

Big-O Examples Analyzing C++ Code

YOUR TURN!!

Code Example 1: Your turn!

1. Do a precise(ish) count
2. Keep only the largest term of the polynomial
3. Strip off any constant coefficient
4. Throw a O(__) around the result, and done!

```
int mystery1(int N)
{
     N++;
     N *= 7;
     N -= 3;
     return N;
}
```

What is the Big-O cost of this code?

Go to pollev.com/cs106b to respond!

Code Example 2: Your turn!

1. Do a precise(ish) count
2. Keep only the largest term of the polynomial
3. Strip off any constant coefficient
4. Throw a O(__) around the result, and done!

```
void mystery2(int N)  // assume N > 3
{
     Grid<int, int> g(N, N);
     for (int row = 0; row < g.numRows(); row++) {
         for (int col = 0; col < 3; col++) {
             g[row][col]++;
         }
     }
}
```

What is the Big-O cost of this code?

Go to pollev.com/cs106b to respond!