Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Today's Topics

Recursion!

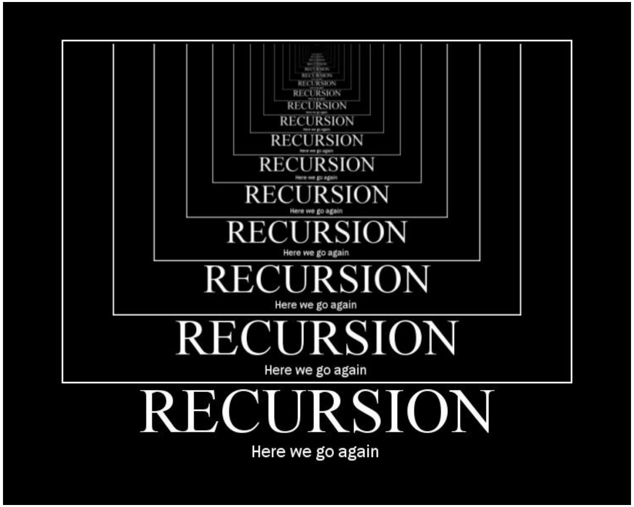
- Functions calling functions

Next time:

- More recursion! It's Recursion Week!
  - Like Shark Week, but more nerdy

Recursion!

The exclamation point isn't there only because this is so exciting; it also relates to our first recursion example….



Factorial!

$$n! = n(n-1)(n-2)(n-3)(n-4)\ldots(3)(2)(1)$$

This could be a really long expression!

Recursion is a technique for tackling large or complicated problems by just "eating" one "bite" of the problem at a time.

Factorial!

$$\boldsymbol{n}! \;=\; \boldsymbol{n}(\boldsymbol{n}-\mathbf{1})(\boldsymbol{n}-\mathbf{2})(\boldsymbol{n}-\mathbf{3})(\boldsymbol{n}-\mathbf{4})\ldots(\mathbf{2})(\mathbf{1})$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n(\boldsymbol{n}-\mathbf{1})! & otherwise \end{cases}$$

Translated to code

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * someFunctionThatKnowsFactorialOfNMinus1();
    }
}
```

Factorial!

$$n! = n(n-1)(n-2)(n-3)(n-4)\ldots(2)(1)$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n(n-1)! & otherwise \end{cases}$$

Translated to code

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

The recursive function pattern

Always two parts:

Base case:

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

Recursive case:

- This problem is still a bit large, let's bite off just one piece, and delegate the remaining work to recursion.

Translated to code

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet! Do 1 piece
        return n * factorial(n - 1);
    }
}
```

The recursive function pattern

Recursive case:

- This problem is still a bit large, **let's bite off just one piece**, and **delegate the remaining work to recursion.**

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {   // Not easy enough to finish yet! Do 1 piece
       return n * factorial(n - 1);
    }
```

This is an example of "one piece" of the problem—just doing <u>one</u> of the many, many multiplications required for factorial.

The recursive function pattern

Recursive case:

- This problem is still a bit large, **let's bite off just one piece**, and **delegate the remaining work to recursion.**

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet! Do 1 piece
        return n * factorial(n - 1);
    }
}
```

This is an example "delegating the remaining work"—all the other multiplications—to the recursive call.

The recursive function pattern

Recursive case:

- This problem is still a bit large, **let's bite off just one piece**, and **delegate the remaining work to recursion.**

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet! Do 1 piece
       return n * factorial(n - 1);
    }
```

This is an example of "one piece" of the problem—just doing one of the many, many multiplications required for factorial.

This is an example "delegating the remaining work"—all the other multiplications—to the recursive call.

Recap: the recursive function pattern

- Recursion is a way of taking a big problem and repeatedly breaking it into smaller and smaller pieces until it is so small that it can be so easily solved that it almost doesn't even need solving.
- There are two parts of a recursive algorithm:
  - base case: where we identify that the problem is so small that we trivially solve it and return that result
  - recursive case: where we see that the problem is still a bit too big for our taste, so we chop it into smaller bits and call **ourselves** (the function we are in now) on the smaller bits to find out the answer to the problem we face

Digging deeper in the recursion

Looking at how recursion works “under the hood”

Factorial!

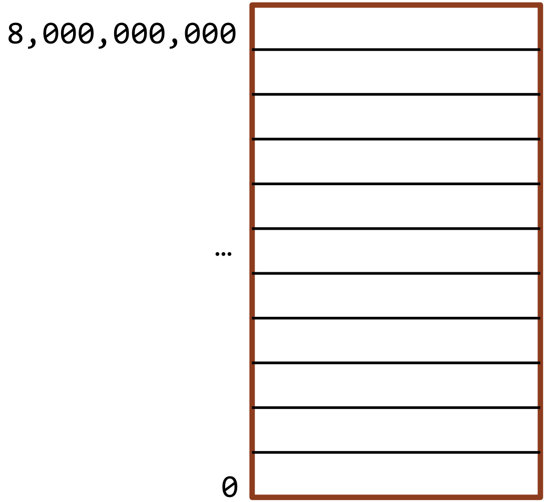
```
int factorial(int n) {
    cout << n << endl;  // **Added for this question**
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet! Do 1 piece
       return n * factorial(n - 1);
    }
}
```

What is the **third** thing **printed** when we call `factorial(4)`?

A. 1
B. 2
C. 3
D. 4
E. Other/none/more

How does this look in memory? A little background…

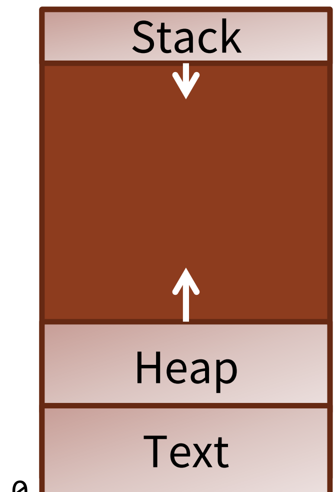
- A computer's memory is like a giant Vector/array, and like a Vector, we start counting at index `0`.
- We typically draw memory vertically (rather than horizontally like a Vector), with index `0` at the bottom.
- A typical laptop's memory has billions of these indexed slots (one byte each)



* Take CS107 to learn much more!!

How does this look in memory?
A little background…

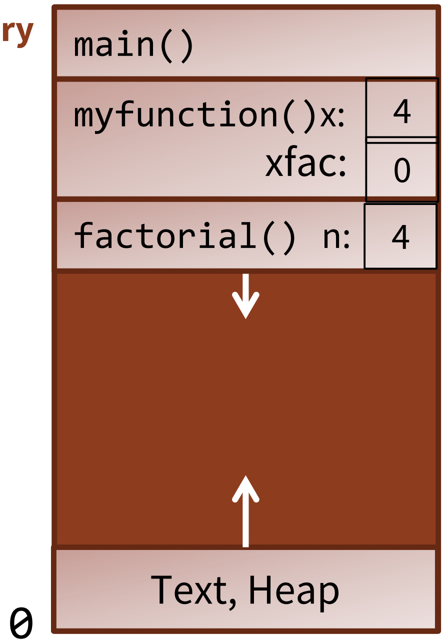
- Broadly speaking, we divide memory into regions:
  - **Text:** the program's own code (needs to be in memory so it can run!)
  - **Heap:** we'll learn about this later in CS106B!
  - **Stack:** this is where local variables for each function are stored.



* Take CS107 to learn much more!!

How does this look in memory?

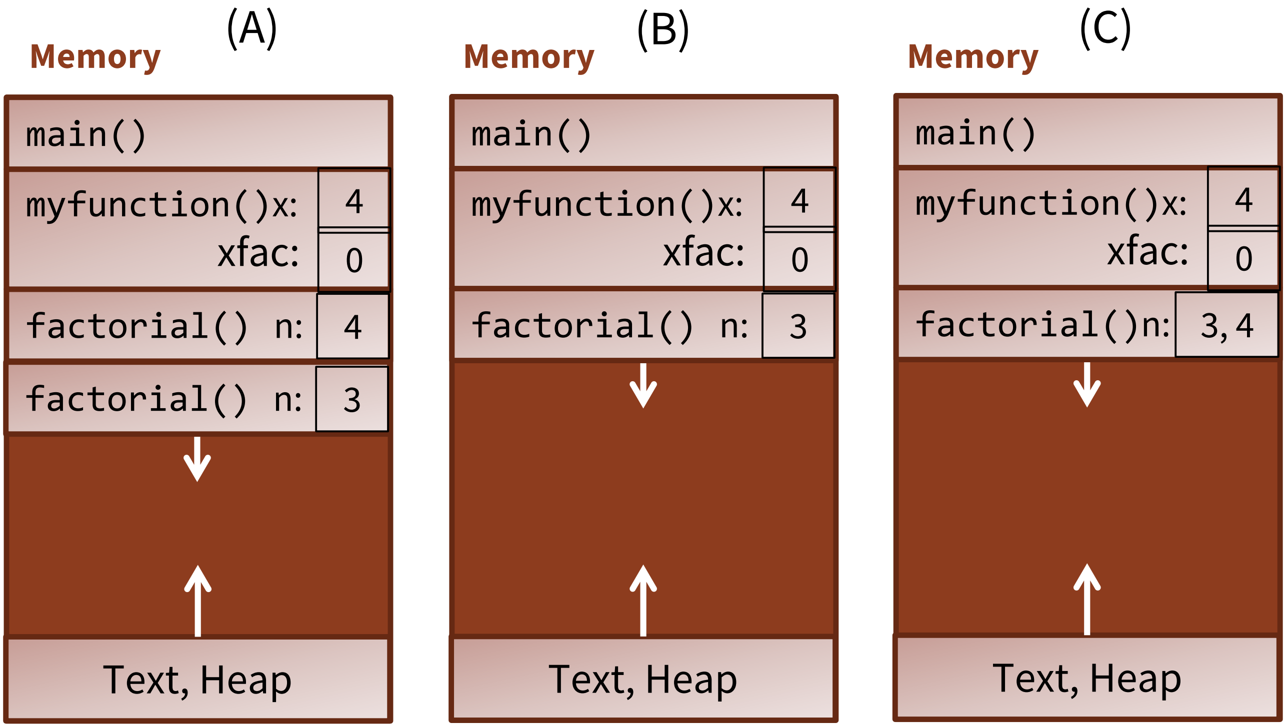
Memory



Recursive code

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

void myfunction(){
     int x = 4;
     int xfac = 0;
     xfac = factorial(x);
}
```



(D) Other/none of the above

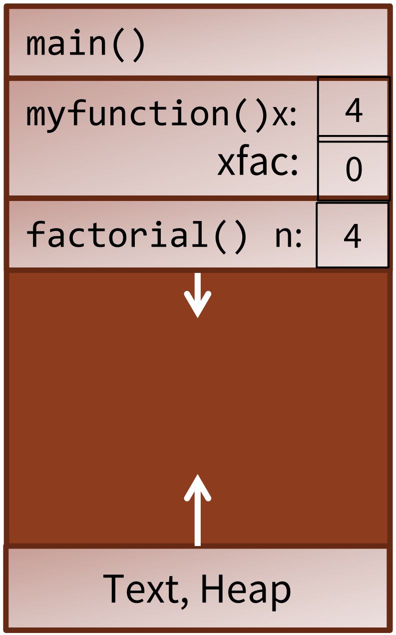
Fun fact:
The “stack” part of memory is a stack

Function **call** = **push** a stack frame

Function **return** = **pop** a stack frame

* Take CS107 to learn much more!!

The “stack” part of memory is a stack

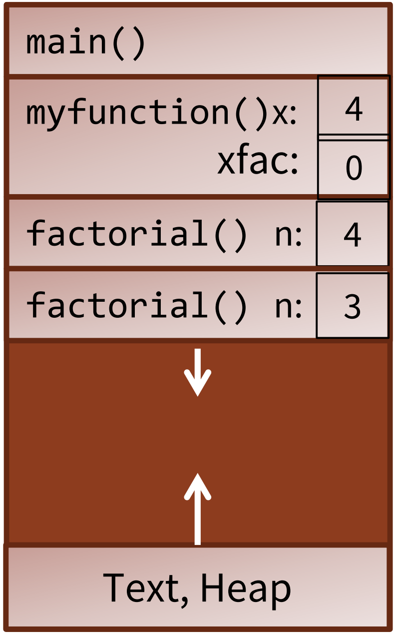


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack

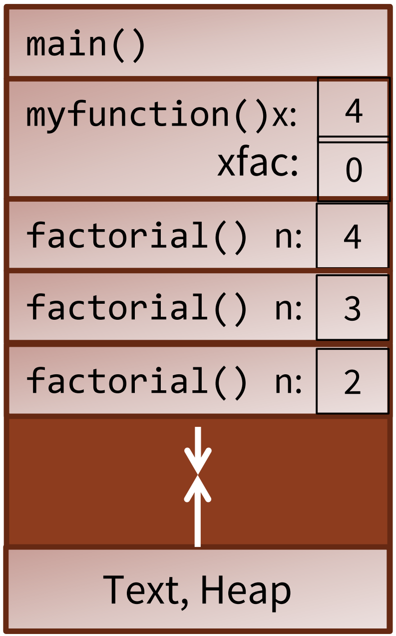


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



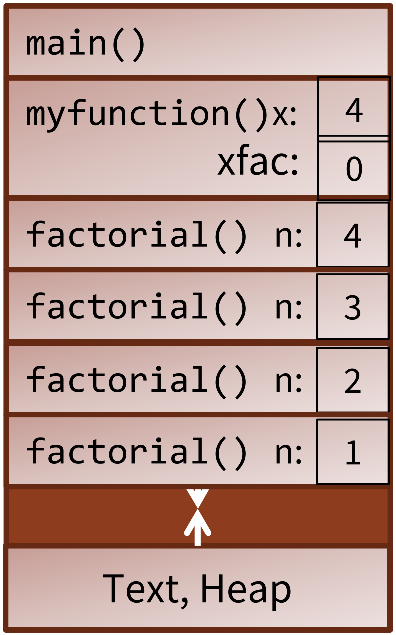
Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

Answer: 3rd thing printed is 2

The "stack" part of memory is a stack



Recursive code

```
int factorial(int n) {
     cout << n << endl;
     if (n == 1) return 1;
     else return n * factorial(n – 1);
}

void myfunction(){
     int x = 4;
     int xfac = 0;
     xfac = factorial(x);
}
```

Factorial!

What is the **fourth** value ever **returned** when we call `factorial(4)`?

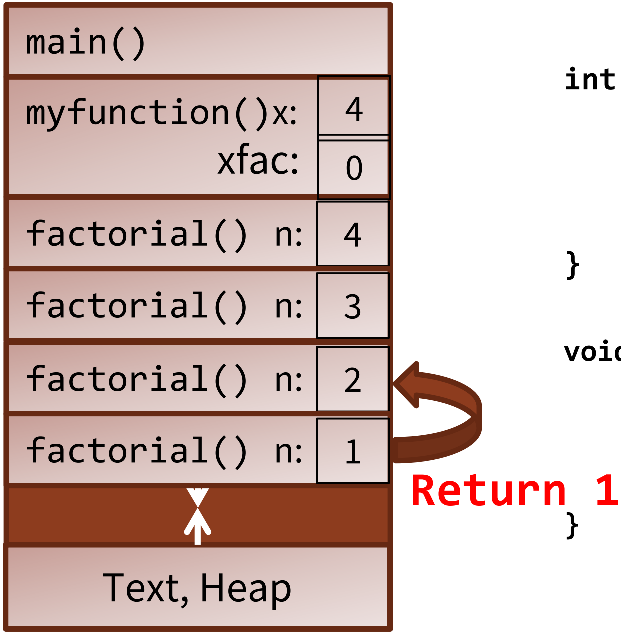
A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

Recursive code

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

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The "stack" part of memory is a stack

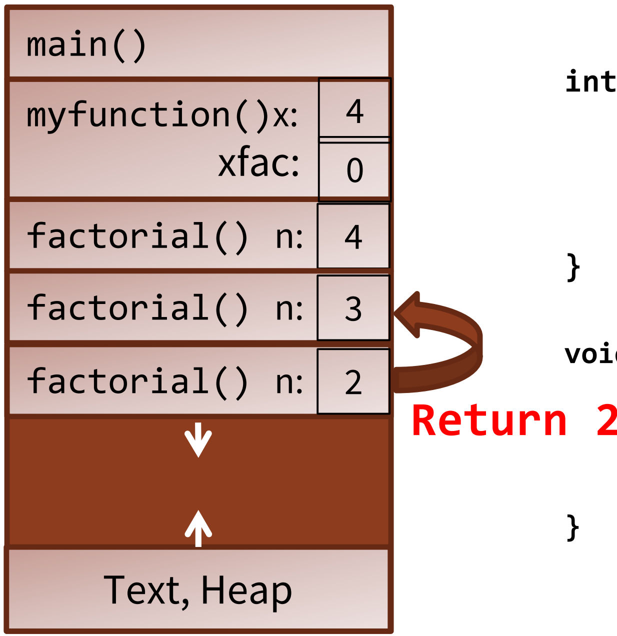


Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack

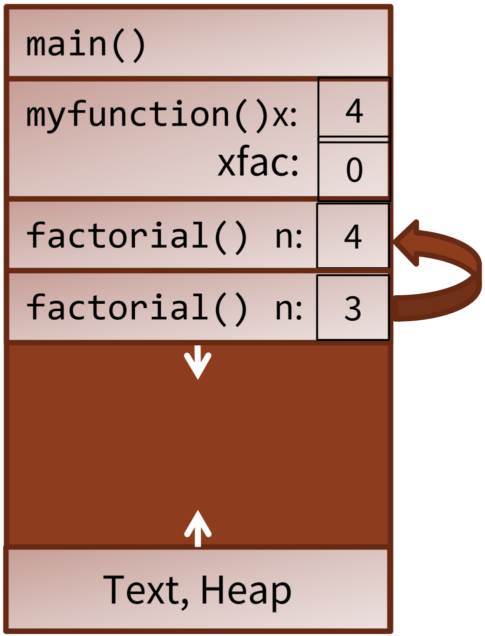


Recursive code

```
int factorial(int n) {
     cout << n << endl;
     if (n == 1) return 1;
     else return n * factorial(n – 1);
}

void myfunction(){
     int x = 4;
     int xfac = 0;
     xfac = factorial(x);
}
```

The "stack" part of memory is a stack



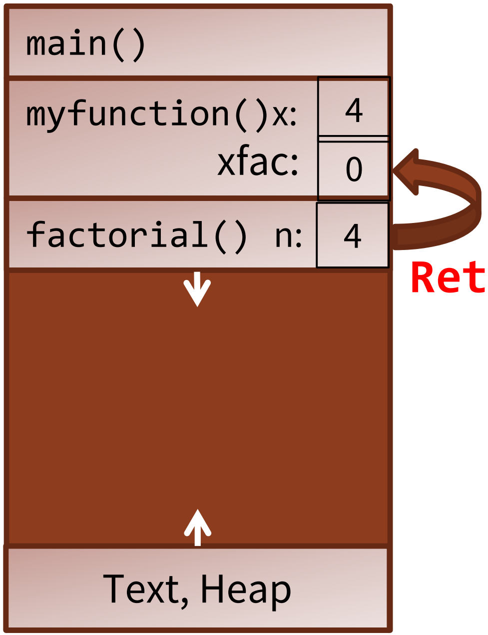
Recursive code

```
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}
```

Return 6

```
void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



Return 24

Recursive code

```
int factorial(int n) {
     cout << n << endl;
     if (n == 1) return 1;
     else return n * factorial(n – 1);
}
```

Answer: 4th thing returned is 24

```
void myfunction(){
     int x = 4;
     int xfac = 0;
     xfac = factorial(x);
}
```

Factorial!

Iterative version

```
int factorial(int n) {
    int f = 1;
    while (n > 1) {
        f = f * n;
        n = n – 1;
    }
    return f;
}
```

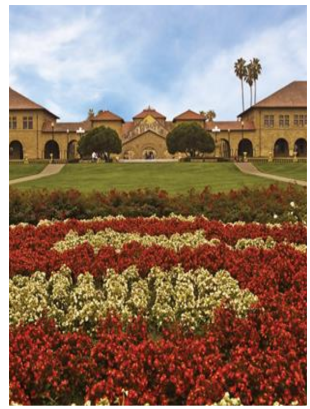
Recursive version

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

NOTE: sometimes **iterative can be much faster** because it doesn't have to push and pop stack frames. Method calls have overhead in terms of space *and* time (to set up and tear down).

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

- **Answer: (D) Something else! (about 50x)**
  - In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
  - **Check the code bundle for class today for runnable version!**

```
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));
}
```

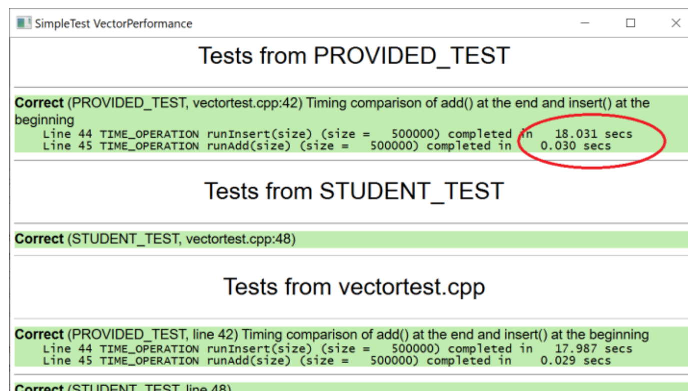
Your turn: Vector performance

- **Answer: (D) Something else! (about 50x)**
  - In addition to analyzing the code and predicting number of writes needed, we can also time the code using our Stanford 106B test system.
  - **Check the code bundle for class today for runnable version!**

```
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));
}
```



Your turn: Vector performance

- **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**

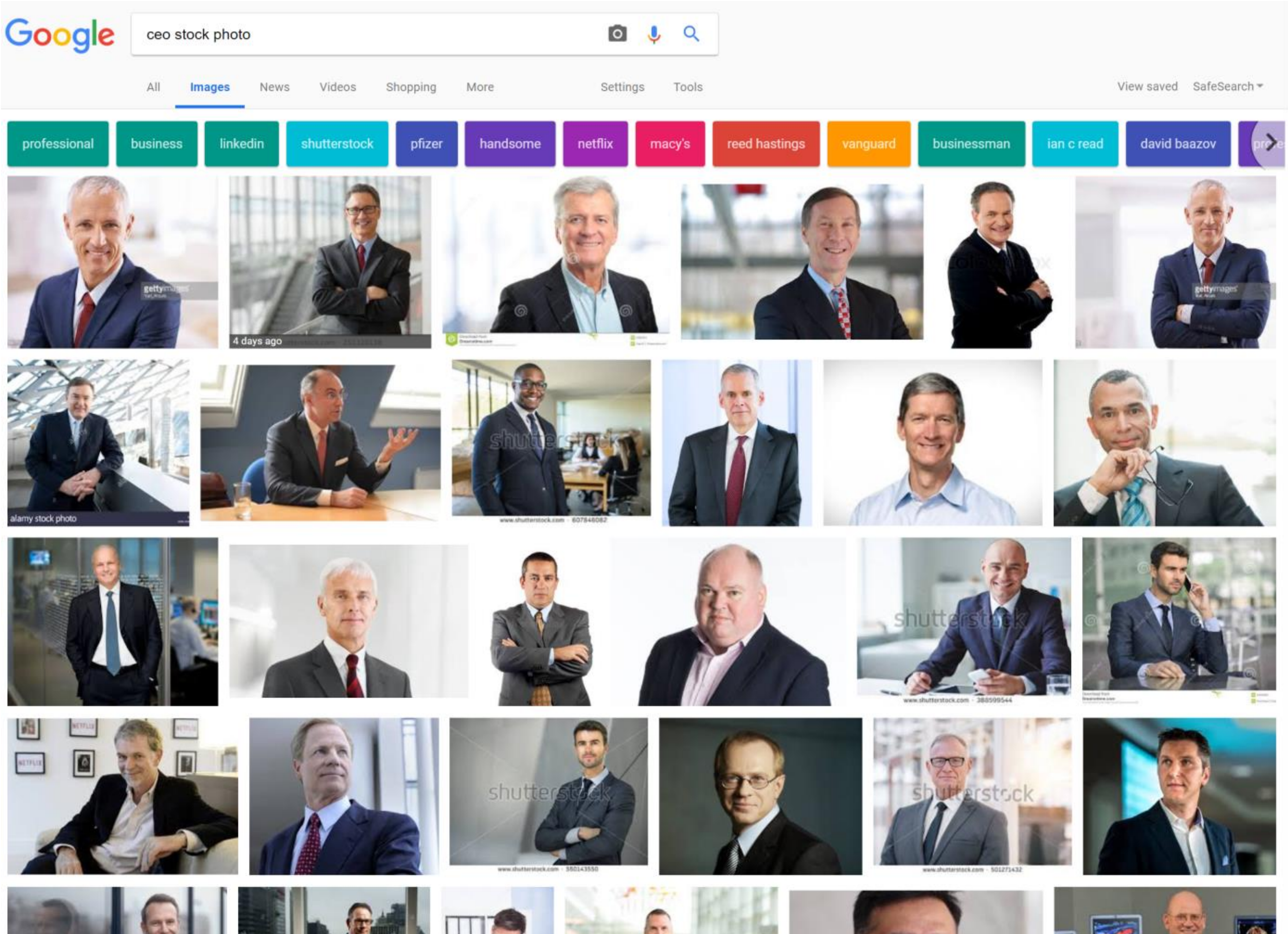
Big-O

- Big-O analysis in computer science is a way of counting the number of “steps” needed to complete a task
  - Doesn’t really consider how “big” each step is
  - Doesn’t consider how fast the computer’s CPU or other hardware components are
  - Doesn’t involve any actual measurement of the time elapsed for any real code in any way
- But despite all that, really useful for making broad comparisons between different approaches

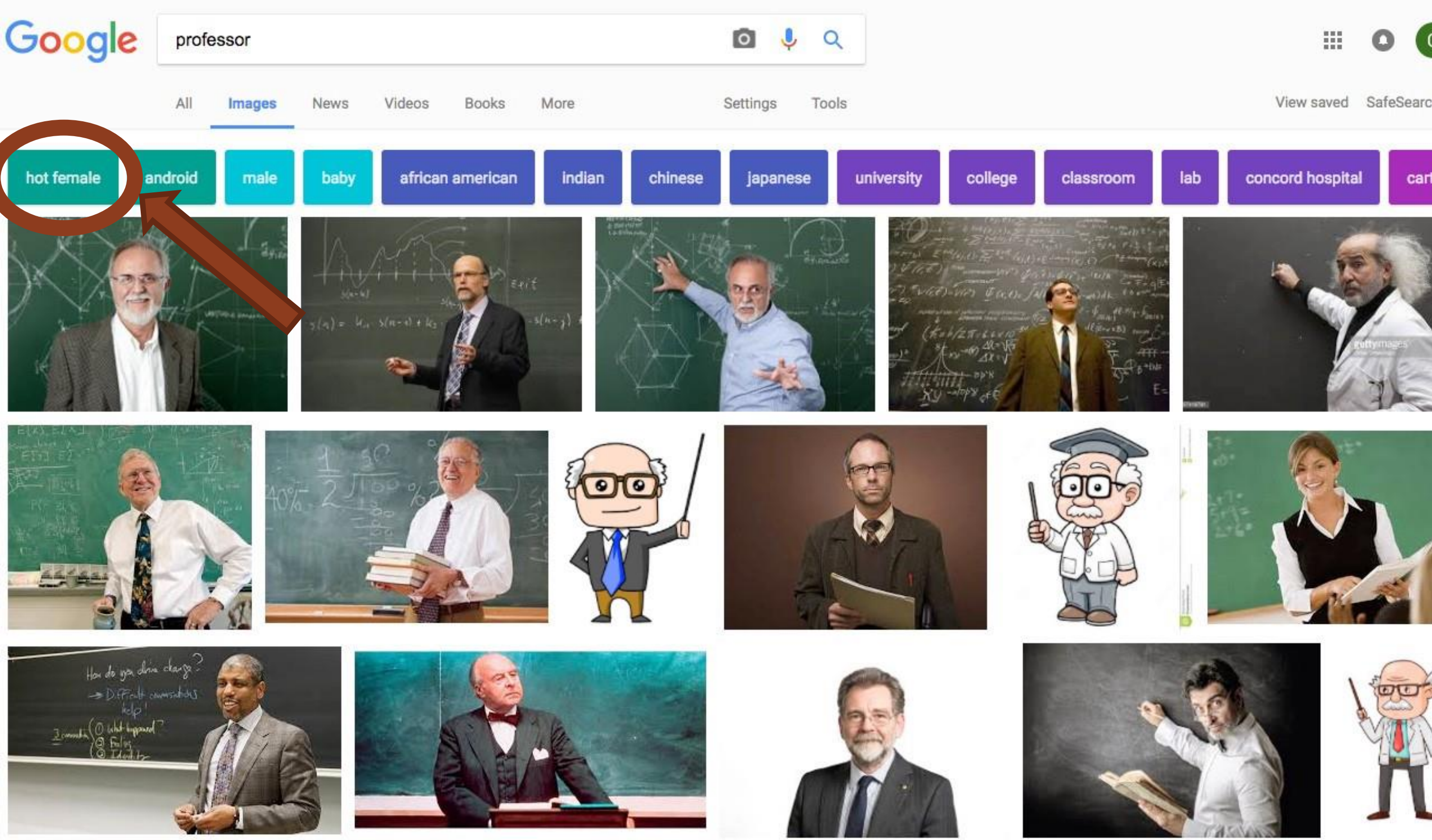
Efficiency as a virtue?

- In computer science, we tend to obsess about **efficiency**, but it's worth taking a step back and asking ourselves, is efficiency always a virtue?
  - Racing to be first to the finish line, but with an answer that's wrong, isn't helpful!
  - That might seem obvious, but it happens *all the time* in real tech products

Google image search

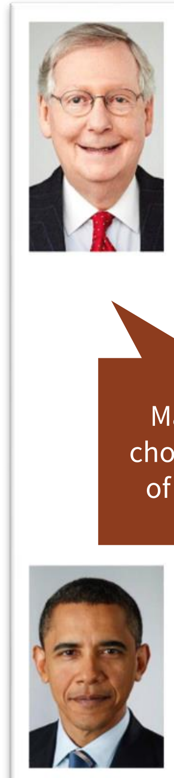


Another example…



The danger of a cheap solution: Twitter cropping

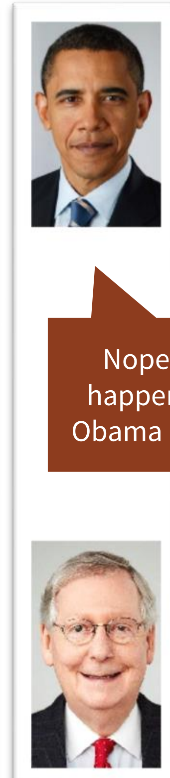
- In the summer of 2020, Twitter users noticed something strange about Twitter's new photo cropping algorithm
- Given a too-tall image, it selects which part to show
- It picked the Senator McConnell (the white man), not President Obama



Maybe it just chooses the top of the photo?

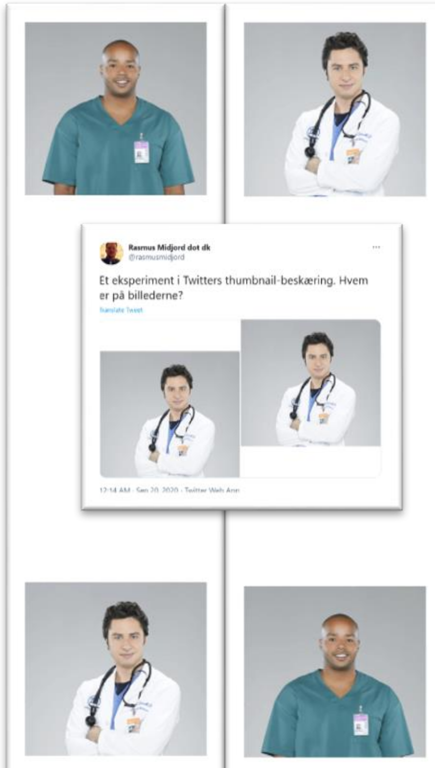
The danger of a cheap solution: Twitter cropping

- In the summer of 2020, Twitter users noticed something strange about Twitter's new photo cropping algorithm
- Given a too-tall image, it selects which part to show
- It picked the Senator McConnell (the white man), not President Obama



Nope! It still happens when Obama is on top!


Cow Momma
I'm doing a test.
I want you all to tell me what you see in the preview for both pics.


Rasmus Midjord dot dk
@rasmusmidjord
Et eksperiment i Twitters thumbnail-beskæring. Hvem er på billederne?


Jordan Simonovski
I wonder if Twitter does this to fictional characters too.
Lenny
Carl
CARL

Efficiency as a virtue?

- In each of these cases, companies chose an algorithm that would be most *efficient*, but came up with answers that were "wrong" (problematic) in ways that are significant for society
- How can we balance cost (which is what efficiency is really about in capitalism) with correctness and justice for society?
  - Reflect on this in your Assignment 2!