



# Algorithm Analysis

CSE 332 Spring 2025  
Lecture 2

# Announcements

Office Hours have started! You can find the schedule on the calendar on the webpage.

Slides and the handout also go up on the calendar (usually before lecture), and inked versions of the slides go up after lecture (usually that afternoon).

# Outline

Wrap up CircularArray Queues and Linked List Queues

How do we analyze code?

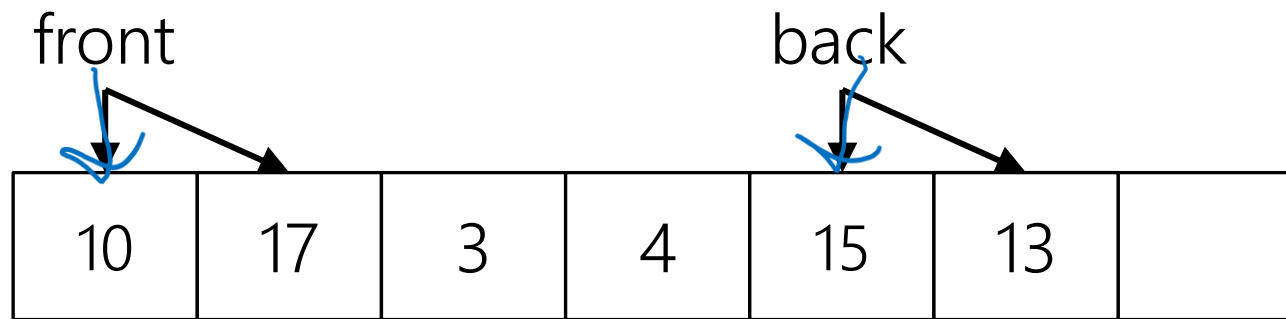
What is big-O formally?

What will big-O proofs be like?

# "Circular" Array

A different queue implementation

Removing elements is expensive. What if we just remember where the next element is?



//sketch only

```
Enqueue(item) {  
    Q[back]=item;  
    back = (back+1) % size;  
}
```

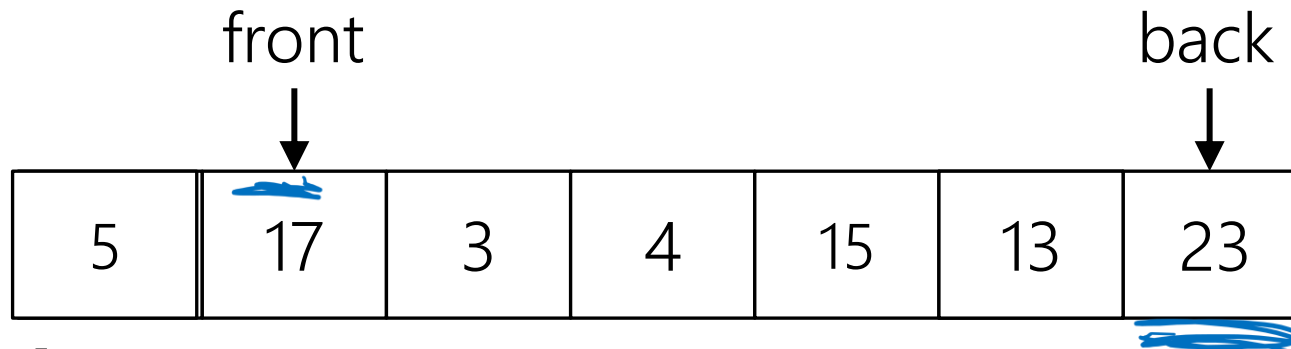
//sketch only

```
Dequeue() {  
    item = Q[front];  
    front = (front+1) % size;  
    return item;  
}
```

# "Circular" Array

What about insertions?

We can "wrap around" (At least until the array is completely full. )



//sketch only

```
Enqueue(item) {  
    Q[back]=item;  
    back = (back+1) % size;  
}
```

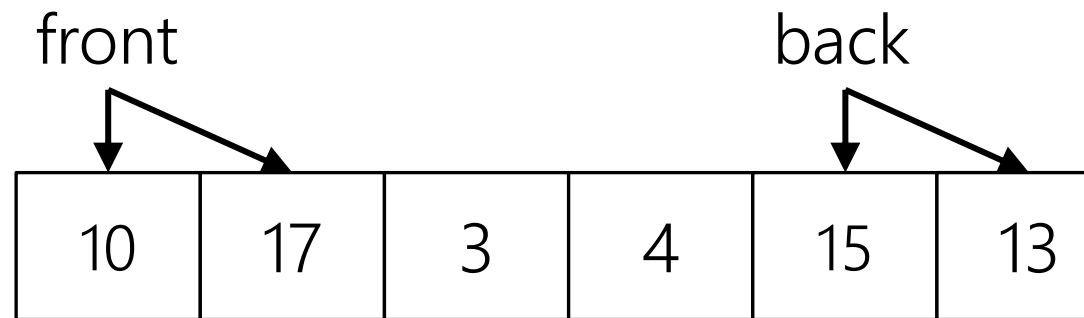
//sketch only

```
Dequeue() {  
    item = Q[front];  
    front = (front+1) % size;  
    return item;  
}
```

# "Circular" Array

A different queue implementation

Removing elements is expensive. What if we just re  
element is?



//sketch only

```
Enqueue(item) {  
  Q[back]=item;  
  back = (back+1) % size;  
}
```

//sketch only

```
Dequeue() {  
  item = Q[front];  
  front = (front+1) % size;  
  return item;  
}
```

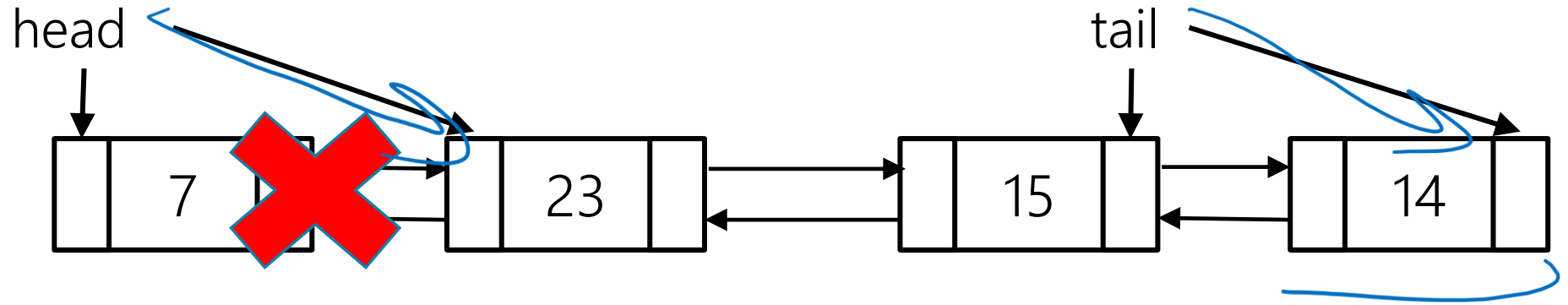
length of array

Sketches don't handle:

- What if queue is empty (for either enqueue or dequeue)
  - How do you test if it's empty
- What if array is full?
- Keeping size up to date
- Etc.

# Linked List

What do Enqueue and Dequeue look like? What are their time complexities?

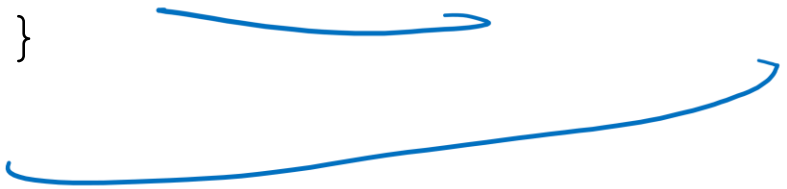


# Linked List

What do Enqueue and Dequeue look like? What are their time complexities?

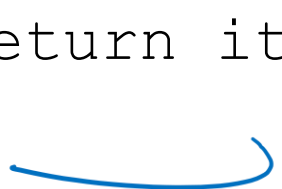
//sketch only

```
Enqueue(item) {  
    tail.next=new Node(item);  
    tail=tail.next;  
}
```



//sketch only

```
Dequeue() {  
    item = head.data;  
    head = head.next;  
    return item;  
}
```



Still just sketches!  
Ask the same questions from  
the circular array.  
Notice that sometimes the  
answer is "don't worry about  
it"!



# Tradeoffs

What makes the circular queue implementation different from the linked list implementation? In what ways is one more desirable than the other?

Array resized when full

Linked List extra space

Array cache behavior

# Tradeoffs

## LINKED LIST

$O(1)$  enqueue and dequeue

Pointers lead to slower constants and worse cache behavior (see CSE351)

More space used per element (pointers take space)

Not in Queue ADT, but if you want to insert at position  $k$ , must traverse  $k$  elements first (or last  $n-k$ ).

## CIRCULAR ARRAY

$O(1)$  enqueue and dequeue unless the array is already full.

Faster constants, but when you resize you get a **very** slow insert.

Less space when full, but potentially **a lot** of wasted space if queue gets huge then tiny.

Not in Queue ADT, but if you want to insert at position  $k$ , must shift last  $n-k$  elements (or first  $k$ ).

# Tradeoffs

This class is built on tradeoffs

If data structure A beats data structure B in every way, you'll choose A.

Usually, data structure A is better in some ways, and B is better in others.

Knowing the tradeoffs will help you frame the choice.

If you know the size in advance, you might choose the array (no resize worries)

If any one insertion being slow would be very bad, you might choose the linked list

# Common Tradeoffs

Often there are multiple unavoidable tradeoffs

- Time vs. space
- One operation vs. another being as fast as possible
- Generality vs. simplicity vs. performance

These tradeoffs are why there are many data structures.

And well-trained computer scientists have internalized those tradeoffs to pick.



# Asymptotic Analysis

---

# Algorithm Analysis

- staff time  
→ What we're running on might not be representative

I have some problem I need solved.

I ask Alysa and Yafqa. They both have different ideas for how to solve the problem. How do we know which is better? — random chance

Easy. Have them both write the code and run it and see which is faster.

## THIS IS (often) A TERRIBLE IDEA

How can we analyze their suggestions before they have to write the code, and in a way that is independent of their machines?

# Algorithm Analysis

“Just code it up and see what happens” isn’t a great strategy for code analysis.

Running time of actual code depends on the computer you’re running on (CPU power, but also OS, other programs running in the background), details of the implementation.

You probably won’t consider all potential inputs in testing

- What if you missed a really bad case?
- You probably won’t be able to explain patterns clearly

Wasteful if you decide this isn’t the right implementation

# Comparing Algorithms

We want to know when one algorithm will be better than another

- Better might mean faster.
- Or using less memory.

We really care about large inputs.

- If  $n=15$ , any algorithm will probably finish in less than a second anyway...

Want our answer to be independent of computer speed or programming language.

And we want an answer that's mathematically rigorous.

- In a 311-like sense. We should have a proposition that we can prove true or false.



# Algorithm Analysis

Usually, define a function  $f: \mathbb{N} \rightarrow \mathbb{N}$

Domain: size of the input to the code (e.g., number of elements in our array, number of characters in our string)

Co-Domain: Counts of resources used (e.g., number of basic operations [time], number of bytes of memory used, etc.)

Be sure you're clear on the units of your domain and co-domain

- It won't make a big difference for this class, but in complexity theory (e.g. CSE 431, some of 421) bits of input vs. number of elements as input can make a big difference.

# What Are We Counting?

## Worst case analysis

- What's the  $f(N)$  [running time, memory, etc.] for the **worst** state our data structure can be in or the **worst** input we can give of size  $N$ ? (i.e. the biggest  $f$  could be on an input size  $N$ )

## Best case analysis

- What is  $f(N)$  for the best state of our structure and the best question of size  $N$ ? (the smallest  $f(N)$  could be)

## Average case analysis

- What is the value of  $f(N)$  on average over all possible inputs of size  $N$ ?
- Have to ask this question very carefully to get a meaningful answer

We usually do worst case analysis.

# Analyzing Code

Assume basic operations take the same constant amount of time.

What's a basic operation?

- Adding ints or doubles

- Assignment

- Incrementing a variable

- A return statement

- Accessing an array index or an object field

What's not a basic operation?

- Making a method call.

This is a LIE but it's a very useful lie.

# Example

What is the worst case number of simple operations for this piece of code?  
Let  $A$  have  $n$  entries.

## Linear search

```
int linearSearch(int[] A, int target){  
    for(int i = 0; i < A.length; i++){  
        if(A[i] == target)  
            return i;  
    }  
    return -1;  
}
```

target is  
not in array

$$3n + 2$$

$$5n + 2$$

$$2n + 3$$

# Asymptotic Notation

That's a nice formula. But does everything in it matter?

Multiplying by constant factors doesn't matter – let's just ignore them.

Lower order terms don't matter – delete them.

Gives us a "simplified big-O"

$$10n \log n + 3n$$

$$5n^2 \log n + 13n^3$$

$$20n \log \log n + 2n \log n$$

$$2^{3n}$$

$$O(n \log n)$$

$$O(n^3)$$

$$O(n \log n)$$

$$O(8^n)$$

# Asymptotic Notation

That's a nice formula. But does everything in it matter?

Multiplying by constant factors doesn't matter – let's just ignore them.

Lower order terms don't matter – delete them.

Gives us a “simplified big-O”

$$10n \log n + 3n \qquad O(n \log n)$$

$$5n^2 \log n + 13n^3 \qquad O(n^3)$$

$$20n \log \log n + 2n \log n \qquad O(n \log n)$$

$$2^{3n} \qquad O(8^n)$$

# Formally Big-O

We wanted to find an upper bound on our algorithm's running time, but

- We don't want to care about constant factors.
- We only care about what happens as  $n$  gets large.

The formal, mathematical definition is Big-O.

## Big- $O$

$f(n)$  is  $O(g(n))$  if there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,

$$f(n) \leq c \cdot g(n)$$

We also say that  $g(n)$  "dominates"  $f(n)$ .

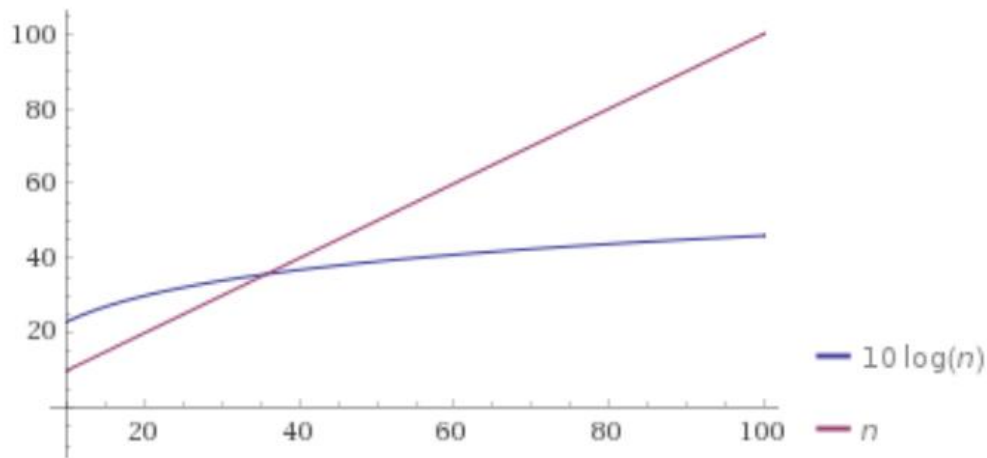
# Why is that the definition?

## Big- $O$

$f(n)$  is  $O(g(n))$  if there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,  
$$f(n) \leq c \cdot g(n)$$

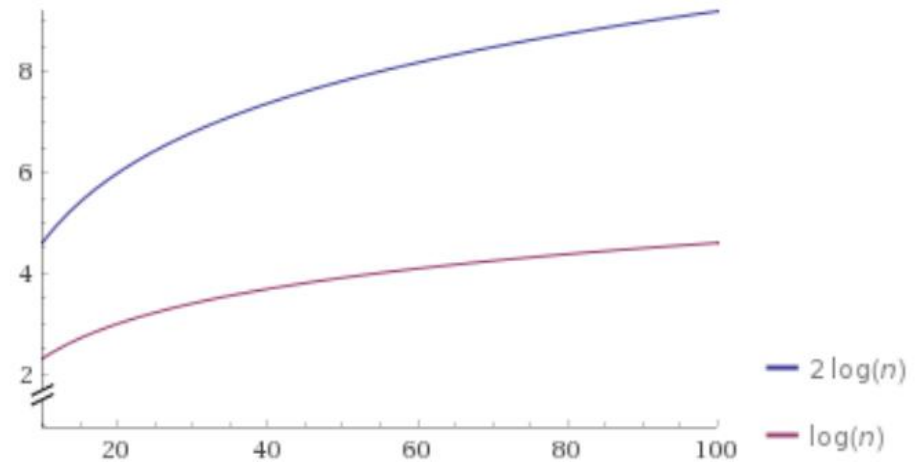
Why  $n_0$ ?

Plot:



Why  $c$ ?

Plot:





# Why Are We Doing This?

You already intuitively understand what big-O means.

Who needs a formal definition anyway?

- We will.

Your intuitive definition and my intuitive definition might be different.

We're going to be making more subtle big-O statements in this class.

- We need a mathematical definition to be sure we're on the same page.

Once we have a mathematical definition, we can go back to intuitive thinking.

- But when a weird edge case, or subtle statement appears, we can figure out what's correct.

# Edge Cases

True or False:  $10n^2 + 15n$  is  $O(n^3)$

[this is an edge case]

It's true! – it fits the definition.

Big-O is just an upper bound. It doesn't have to be a good upper bound.

If we want the best upper bound, we'll ask you for a **tight** big-O bound.

$O(n^2)$  is the tight bound for this example.

It is (usually) technically correct to say your code runs in time  $O(n^{n!})$ .

-DO NOT TRY TO PULL THIS TRICK ON AN EXAM. Or in an interview.

# O, Omega, Theta [oh my?]

Big-O is an **upper bound**

-My code uses at most this many resources (e.g. runs in at most this much time)

Big-Omega is a lower bound

## Big-Omega

$f(n)$  is  $\Omega(g(n))$  if there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,  
$$f(n) \geq c \cdot g(n)$$

Big Theta is “equal to”

## Big-Theta

$f(n)$  is  $\Theta(g(n))$  if  
 $f(n)$  is  $O(g(n))$  and  $f(n)$  is  $\Omega(g(n))$ .

# Viewing $O$ as a class

Sometimes you'll see big- $O$  defined as a family or set of functions.

## Big- $O$ (alternative definition)

$O(g(n))$  is the set of all functions  $f(n)$  such that there exist positive constants  $c, n_0$  such that for all  $n \geq n_0$ ,  $f(n) \leq c \cdot g(n)$

For that reason, some people write  $f(n) \in O(g(n))$  where we wrote " $f(n)$  is  $O(g(n))$ ".

Other people write " $f(n) = O(g(n))$ " to mean the same thing.

We never write  $O(5n)$  instead of  $O(n)$  – they're the same thing!

It's like writing  $\frac{6}{2}$  instead of 3. It just looks weird.

# Useful Vocab

The most common running times all have fancy names:

$O(1)$  constant

$O(\log n)$  logarithmic

$O(n)$  linear

$O(n \log n)$  "n log n"

$O(n^2)$  quadratic

$O(n^3)$  cubic

$O(n^c)$  polynomial (where  $c$  is a constant)

$O(c^n)$  exponential (where  $c$  is a constant)

# What's the base of the log?

If I write  $\log n$ , without specifying a base, I mean  $\log_2 n$ .

But does it matter for big-O?

Suppose we found an algorithm with running time  $\log_5 n$  instead.

Is that different from  $O(\log_2 n)$ ?

No!

$\log_c n = \frac{\log_2 n}{\log_2 c}$  If  $c$  is a constant, then  $\log_2 c$  is just a constant, and we can hide it inside the  $O()$ .

# $O$ , $\Omega$ , $\Theta$ vs. Best, Worst, Average

It's a common misconception that  $\Omega()$  is "best-case" and  $O()$  is "worst-case". This is a misconception!!

$O()$  says "the complexity of this algorithm is at most" (think  $\leq$ )

$\Omega()$  says "the complexity of this algorithm is at least" (think  $\geq$ )

You can use  $\leq$  on worst-case or best case; you can use  $\geq$  on worst-case or best-case.

Best/Worst/Average say "what function  $f$  am I analyzing?"

$O$ ,  $\Omega$ ,  $\Theta$  say "let me summarize what I know about  $f$ , it's  $\leq$ ,  $\geq$ ,  $=$ ..."

# Some Example Sentences

	$O$	$\Omega$	$\Theta$
Best-Case Analysis	In the best-case, linear search will take at most as much time as binary search ever takes. That is, it's $O(\log n)$ .	In the best case, linear search still has to look at array index 0; those operations still take time, so you will take at least $\Omega(1)$ time.	In the best case, linear-search is both $O(1)$ and $\Omega(1)$ so it is $\Theta(1)$ .
Worst-Case Analysis	In the worst-case, binary search will take at most as much time as linear search ever takes. That is, it's $O(n)$ .	In the worst-case, linear search will check at least as many locations in the array as binary search does in the worst-case, so it will take at least $\Omega(\log n)$ time	In the worst-case, binary search takes $\Theta(\log n)$ time.  In the worst-case, linear search takes $\Theta(n)$ time.



# Why Might you use it?

	$O$	$\Omega$	$\Theta$
Best-Case Analysis	In the best case, my algorithm is pretty good, it takes at most $O(time)$	Even in the best-case, this algorithm still takes a while; it takes at least $\Omega(time)$	In the best case, my algorithm takes exactly $\Theta(time)$
Worst-Case Analysis	Even in the worst-case my algorithm, isn't that bad! It takes at most $O(time)$ time in the worst-case	In the worst-case, there's still a lot of work the algorithm has to do; it takes at least $\Omega(time)$	In the worst case, my algorithm takes exactly $\Theta(time)$



## Some Log Review

---

# Logs

$$\log(n^k) = k \cdot \log(n).$$

$$\log(2^n) = n \text{ (logs and exponents are inverse functions.)}$$

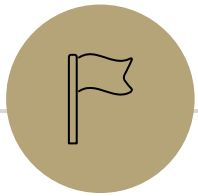
$\log(\log(x))$  is usually written  $\log \log x$

Grows **VERY** slowly (as slowly as  $2^{(2^x)}$  grows quickly)

$$\log_2(\log_2(\# \text{ atoms in universe})) = \log_2(\log_2(10^{80})) = \log_2(80 \cdot \log_2(10)) \approx 8.054$$

Don't confuse with  $\log(x) \cdot \log(x)$  usually written  $\log^2(x)$ .

$\log \log x \ll \log x \ll \log^2(x)$  (where  $\ll$  is "asymptotically less than")



# Writing Big-O Proofs

---

# Proving Big-O, Formally

Big-O is an  $\exists c, n_0 \forall n$  statement.

I.e., an exists statement with a “forall” inside.

How do you prove an exists statement?

How do you prove a for-all statement?

# Proving Big-O, Formally

Big-O is an  $\exists c, n_0 \forall n$  statement.

I.e., an exists statement with a “forall” inside.

How do you prove an exists statement?

Show the  $c, n_0$  that will work. Give specific values.

How do you prove a for-all statement?

Introduce an arbitrary  $n$ .

# Writing Proofs

Claim: For every odd integer  $y$ , there exists an even integer  $x$ , such that  $x > y$ .

Proof:

Let  $y$  be an arbitrary odd integer. By definition,  $y = 2z + 1$  for some integer  $z$ .

Consider  $x = 2(z + 1)$ .

$x$  is even (since it can be written as 2 times some integer) and

$x = 2(z + 1) = 2z + 2 > 2z + 1 = y$ . So  $x$  meets both of the required properties. □

# Writing Proofs

Where did that  $x = 2(z + 1)$  come from?

You probably came up with that even integer first, before you started writing the proof.

That was some “scratch work” – the insight isn’t explained in the final proof

- You just say “Consider”

Don’t try to skip the scratch work when drafting your big-O proofs.

- But it won’t appear in your final version.



# Using the Definition

Let's show:  $10n^2 + 15n$  is  $O(n^2)$

# Using the Definition

Let's show:  $10n^2 + 15n$  is  $O(n^2)$

Recreation of whiteboard:

Scratch work:

$$10n^2 \leq 10n^2$$

$$15n \leq 15n^2 \text{ for } n \geq 1$$

$$10n^2 + 15n \leq 25n^2 \text{ for } n \geq 1$$

Proof:

Take  $c = 25$  and  $n_0 = 1$ .

The inequality  $10n^2 \leq 10n^2$  is always true. The inequality  $15n \leq 15n^2$  is true for  $n \geq 1$ , as the right hand side is a factor of  $n$  more than the right hand side.

As long as both inequalities are true we can add them, thus

$$10n^2 + 15n \leq 25n^2 \text{ holds as long as } n \geq 1.$$

This is exactly the inequality we needed to show.