Please fill out the Poll at- pollev.com/21sp373

Lecture 4: Asymptotic Analysis

CSE 373: Data Structures and Algorithms

Announcements

HW 0 – 143 Review Project

- Live on website
- Due Wednesday April 7th

Find a partner by Wednesday April 7th

- Groups of 3 are ok

Questions?

Queues

Review: What is a Queue?

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

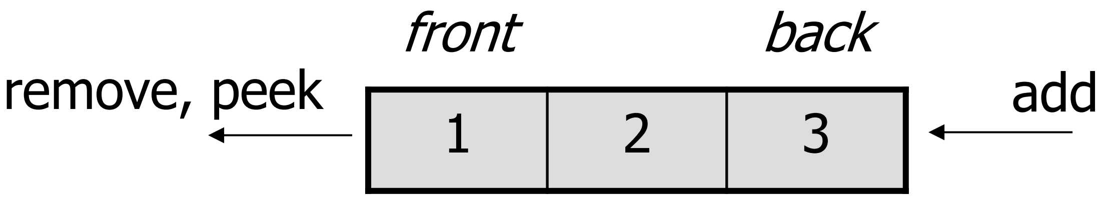
Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?



supported operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Implementing a Queue with an Array

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

ArrayQueue<E>

state

```
data[]
Size
front index
back index
```

behavior

```
add - data[size] = value, if
out of room grow data
remove - return data[size -
1], size-1
peek - return data[size - 1]
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(N) linear if you have to resize O(1) otherwise

```
add(5)
add(8)
add(9)
remove()
```

0	1	2	3	4
5	8	9		

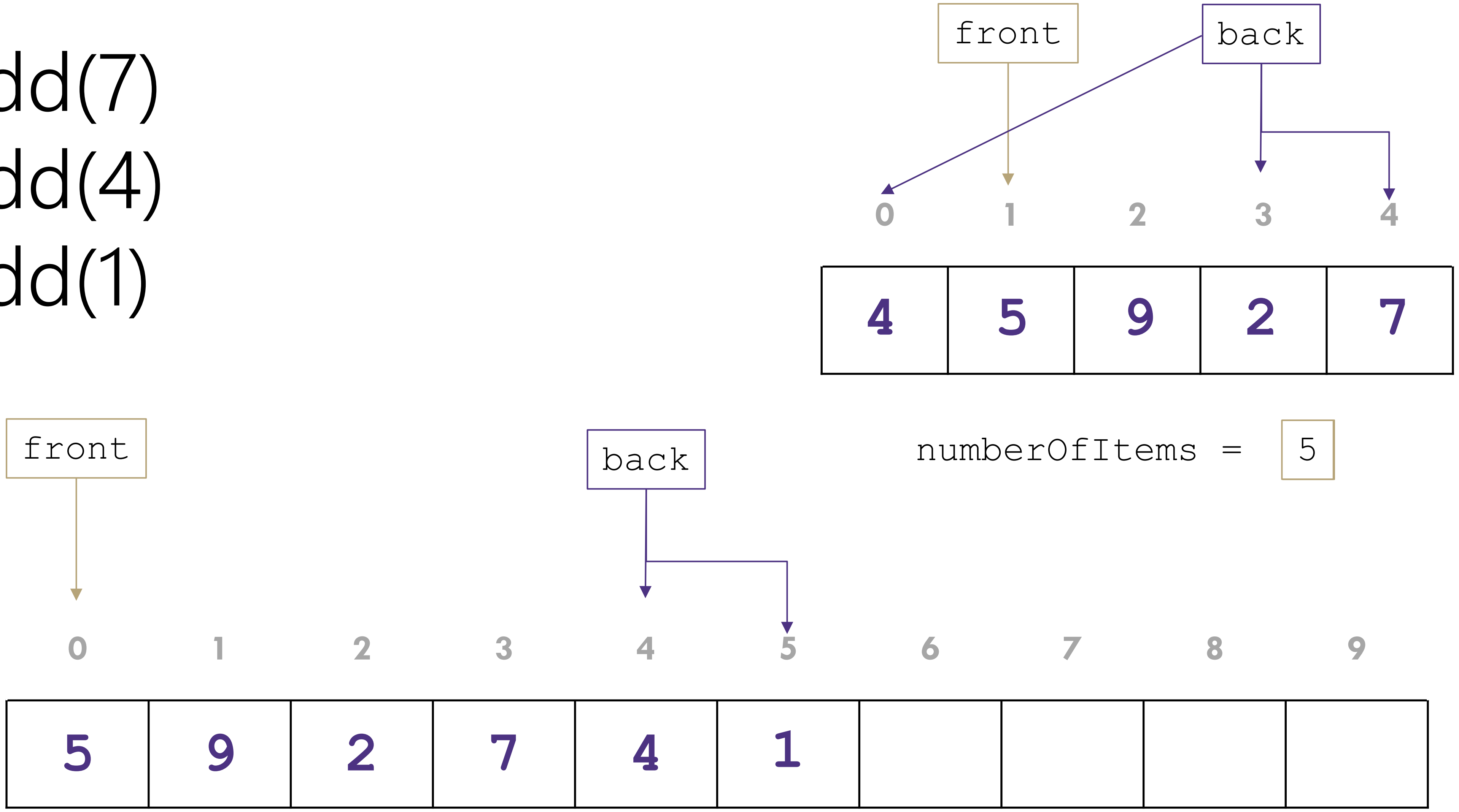
```
numberOfItems = 3
front = 1
back = 2
```

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the "add()" operation will be?

Implementing a Queue with an Array
> Wrapping Around

add(7)
add(4)
add(1)



Implementing a Queue with Nodes

Queue ADT

state
Set of ordered items
Number of items

behavior
add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

LinkedQueue<E>

state
```
Node front
Node back
size
```

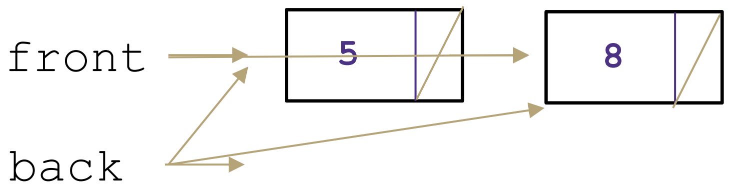
behavior
```
add - add node to back
remove - return and remove node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

```
add(5)
add(8)
remove()
```

`numberOfItems` = 2



Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst case runtime of the “add()” operation will be?

Questions?

Dictionaries

Dictionaries (aka Maps)

Every Programmer's Best Friend

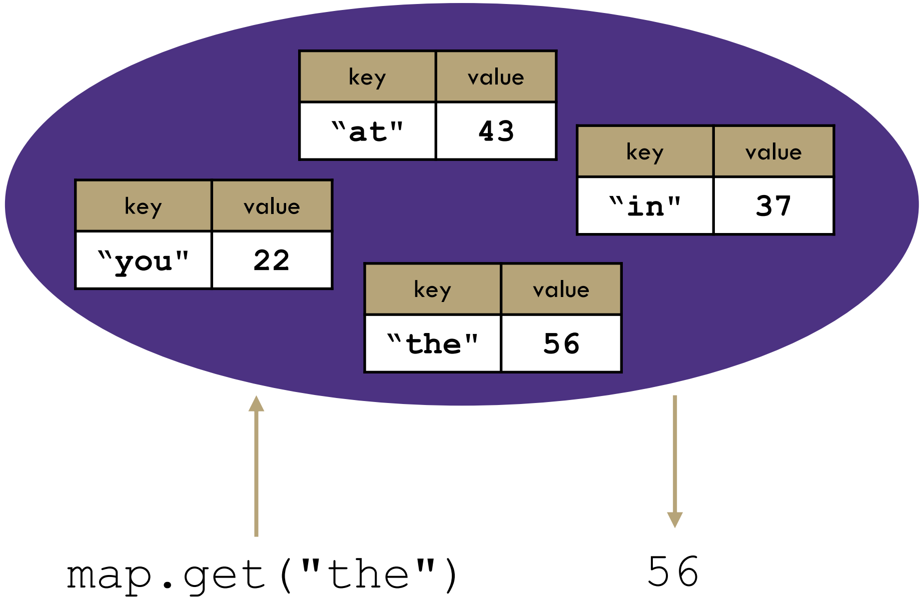
You'll probably use one in almost every programming project.

- Because it's hard to make a big project without needing one sooner or later.

```
// two types of Map implementations supposedly covered in CSE 143
Map<String, Integer> map1 = new HashMap<>();
Map<String, String> map2 =  new TreeMap<>();
```

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.
- a.k.a. "dictionary"



Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug → 37.3

2 Minutes

Implementing a Dictionary with an Array

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayDictionary<K, V>

state

Pair<K, V>[] data

behavior

put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, replace pair to be removed with last pair in collection
size return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b' 97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis – (if key is the last one looked at / not in the dictionary)

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – (if the key is the first one looked at)

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

2 Minutes

Implementing a Dictionary with Nodes

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

LinkedDictionary<K, V>

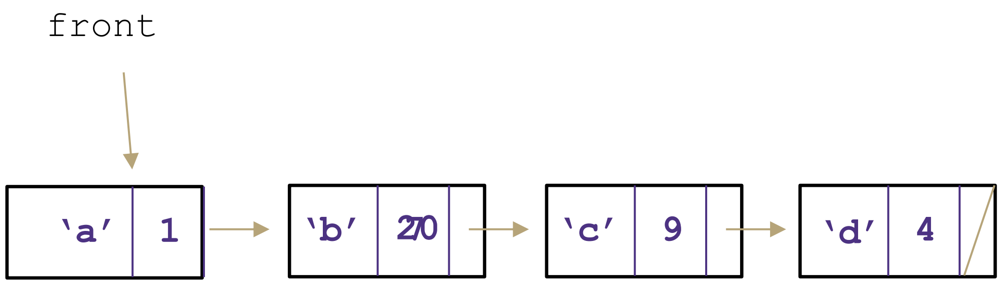
state

```
front
size
```

behavior

```
put if key is unused, create new with
pair, add to front of list, else
replace with new value
get scan all pairs looking for given
key, return associated item if found
containsKey scan all pairs, return if
key is found
remove scan all pairs, skip pair to be
removed
size return count of items in
dictionary
```

```
containsKey('c')
get('d')
put('b', 20)
```



Big O Analysis – (if key is the last one looked at / not in the dictionary)

`put()`	O(N) linear
`get()`	O(N) linear
`containsKey()`	O(N) linear
`remove()`	O(N) linear
`size()`	O(1) constant

Big O Analysis – (if the key is the first one looked at)

`put()`	O(1) constant
`get()`	O(1) constant
`containsKey()`	O(1) constant
`remove()`	O(1) constant
`size()`	O(1) constant

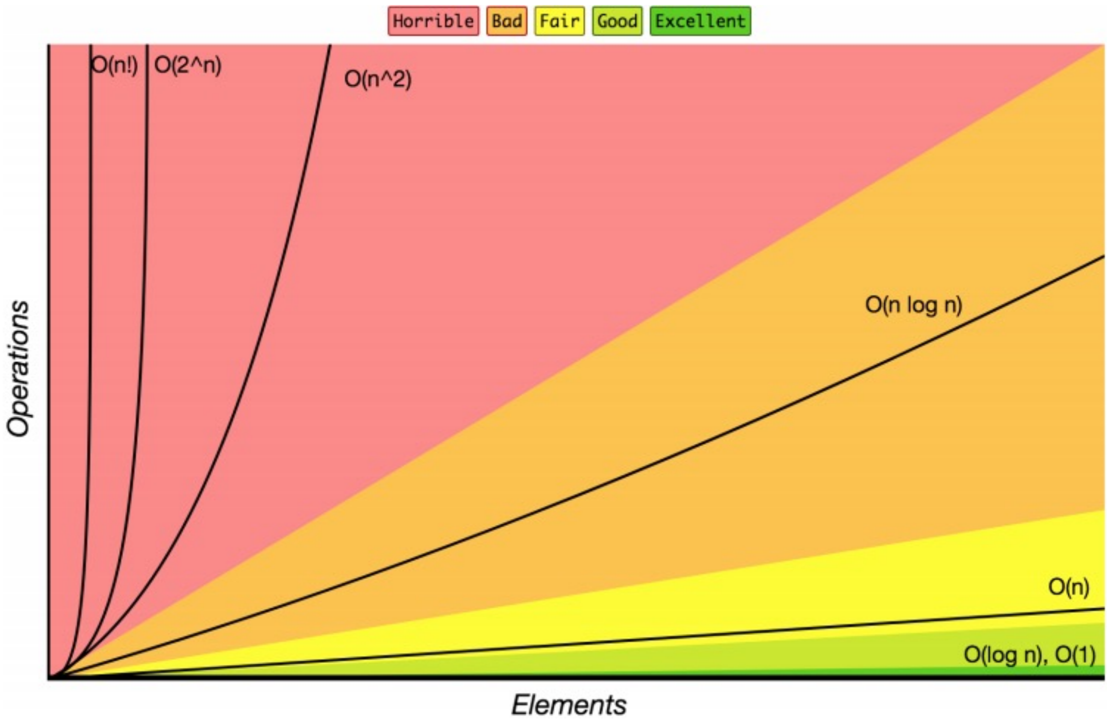
Big O

Review: Complexity Class

Note: You don't have to understand all of this right now – we'll dive into it soon.

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

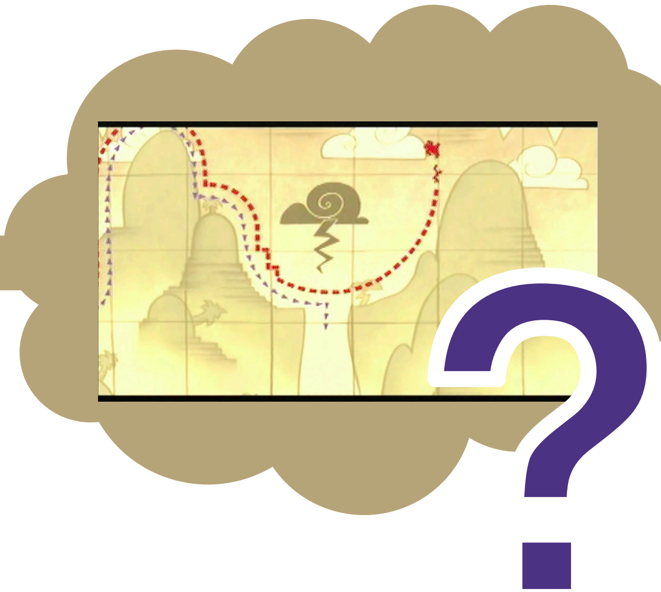
Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



Code to Big-Oh

CODE

```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```



BIG-OH

O(n)

143 general patterns: "O(1) constant is no loops, O(n) is one loop, O(n^2) is nested loops"

- This is still useful!
- But in 373 we'll go much more in depth: we can explain more about *why*, and how to handle more complex cases when they arise (which they will!)

Meet Algorithmic Analysis


```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

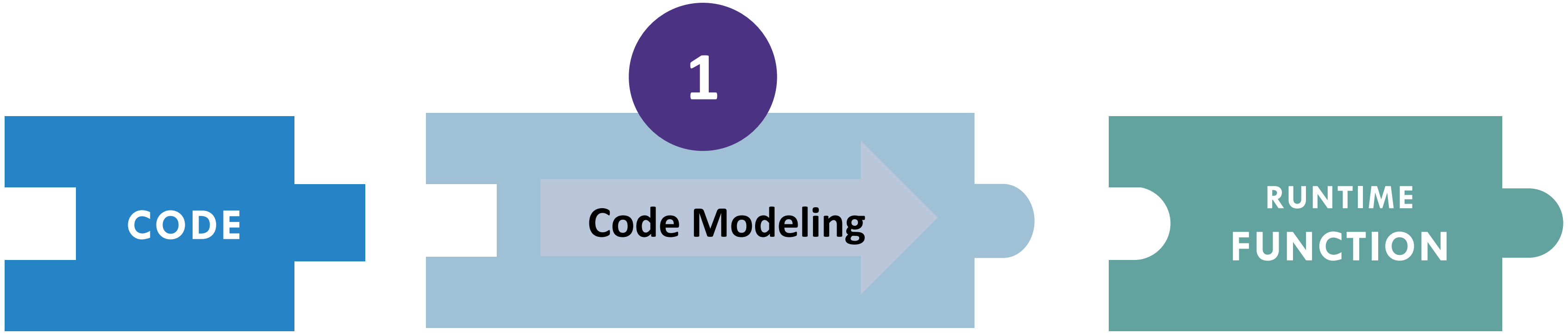
$f(n) = 2n$

$O(n)$

Algorithmic Analysis: The overall process of characterizing code with a complexity class, consisting of:

- Code Modeling: Code → Function describing code's runtime
- Asymptotic Analysis: Function → Complexity class describing asymptotic behavior

Code Modeling



Code Modeling – the process of mathematically representing how many operations a piece of code will run in relation to the input size n.

- Convert from code to a function representing its runtime

What Counts?

We don't know exact runtime of every operation, but for now let's try simplifying assumption: all basic operations take the same time

- Basics:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access
- Function Calls
  - Total runtime in body
  - Remember: `new` calls a function (constructor)
- Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch later
- Loops
  - Number of iterations * total runtime in condition and body

Code Modeling Example 1

```
public void method1(int n) {
    int sum = 0;   +1
    int i = 0;   +1
    while (i < n) {   +1
        sum = sum + (i * 3);   +3
        i = i + 1;   +2
    }
    return sum;   +1
}
```

Loop runs n times

+6 *n

f(n) = 6n + 3

Code Modeling Example 2

```
public void method2(int n) {
    int sum = 0;    +1
    int i = 0;    +1
    while (i < n) {    +1
        int j = 0;    +1
        while (j < n) {    +1
            if (j % 2 == 0) {    +2
                // do nothing
            }
            sum = sum + (i * 3) + j;    +4
            j = j + 1;    +2
        }
        i = i + 1;    +2
    } return sum;    +1
}
```

This inner loop runs n times

+9 *n

This outer loop runs n times

9n + 4 *n

$f(n) = (9n+4)n + 3$

Where are we?


```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

$f(n) = 2n$

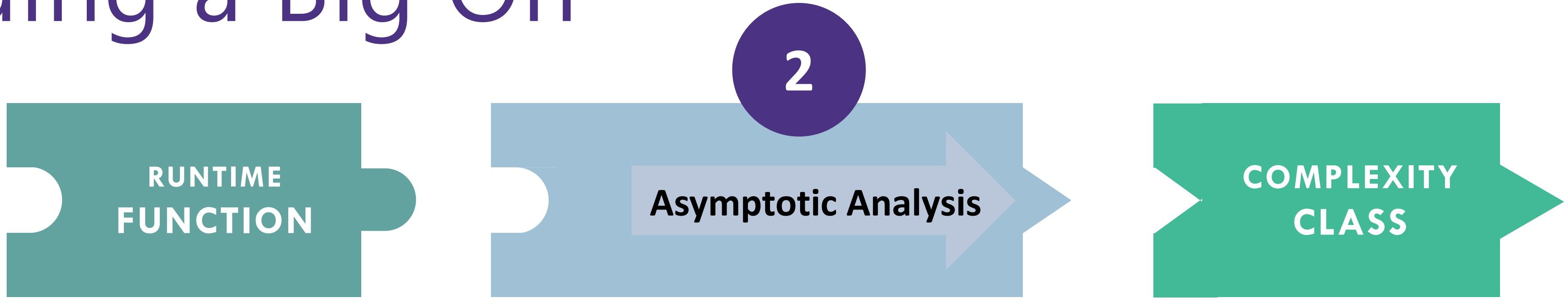
$O(n)$

We just turned a piece of code into a function!

- We'll look at better alternatives for code modeling later

Now to focus on step 2, asymptotic analysis

Finding a Big Oh



We have an expression for $f(n)$. How do we get the $O()$ that we've been talking about?

1. Find the "dominating term" and delete all others.
   - The "dominating" term is the one that is largest as n gets bigger. In this class, often the largest power of n.

2. Remove any constant factors.

f(n) = (9n+3)n + 3

$= 9n^2 + 3n + 3$

$\approx 9n^2$

$\approx n^2$

f(n) is $O(n^2)$

Can we really throw away all that info?

Big-Oh is like the “significant digits” of computer science

Asymptotic Analysis: Analysis of function behavior as its input approaches infinity

- We only care about what happens when n approaches infinity
- For small inputs, doesn’t really matter: all code is “fast enough”
- Since we’re dealing with infinity, constants and lower-order terms don’t meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

Simple

We don’t care about tiny differences in implementation, want the big picture result

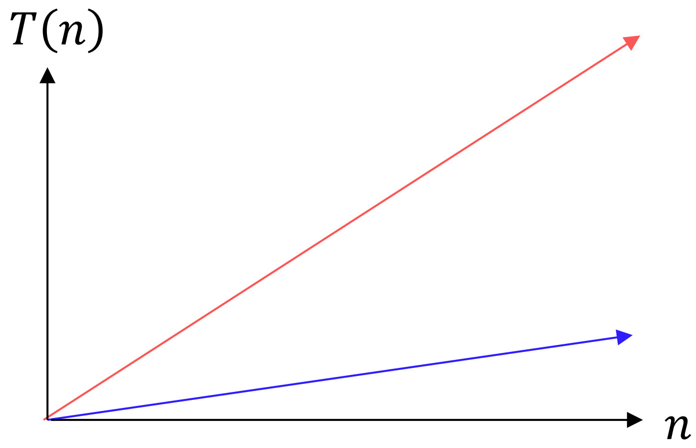
Decisive

Produce a clear comparison indicating which code takes “longer”

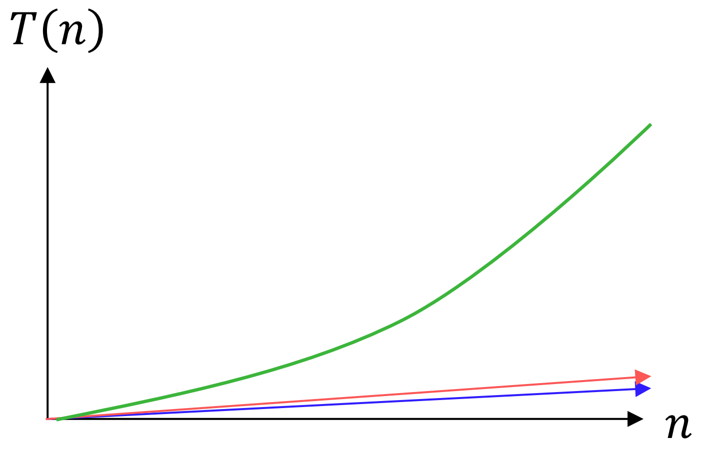
Function growth

Imagine you have three possible algorithms to choose between.
Each has already been reduced to its mathematical model

$f(n) = n$ $g(n) = 4n$ $h(n) = n^2$

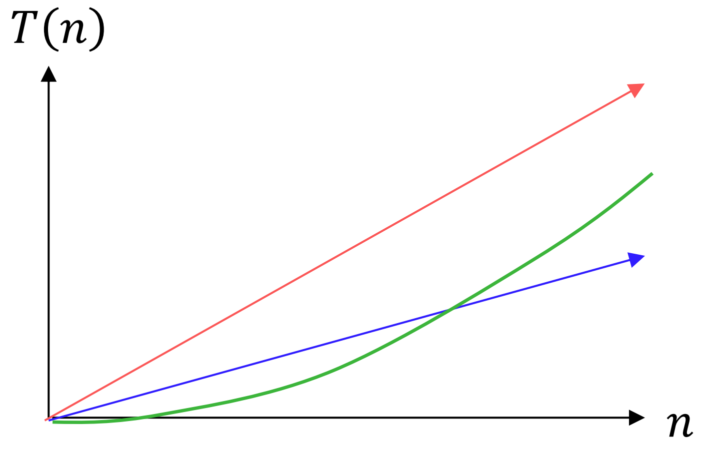


The growth rate for f(n) and g(n) looks very different for small numbers of input



...but since both are linear eventually look similar at large input sizes

whereas h(n) has a distinctly different growth rate



But for very small input values h(n) actually has a slower growth rate than either f(n) or g(n)

Definition: 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.

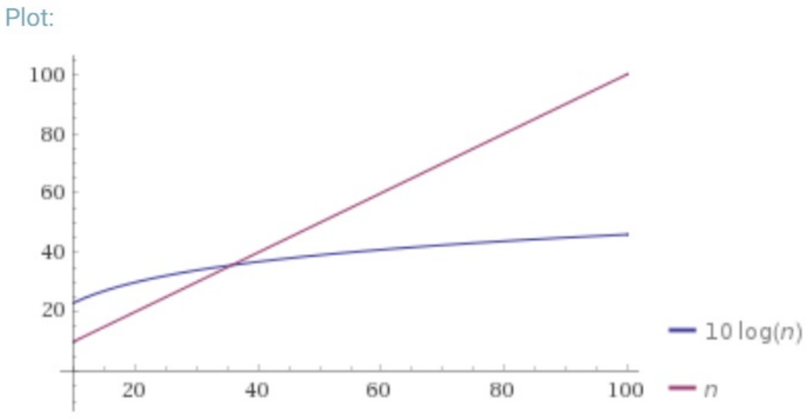
Big-O

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

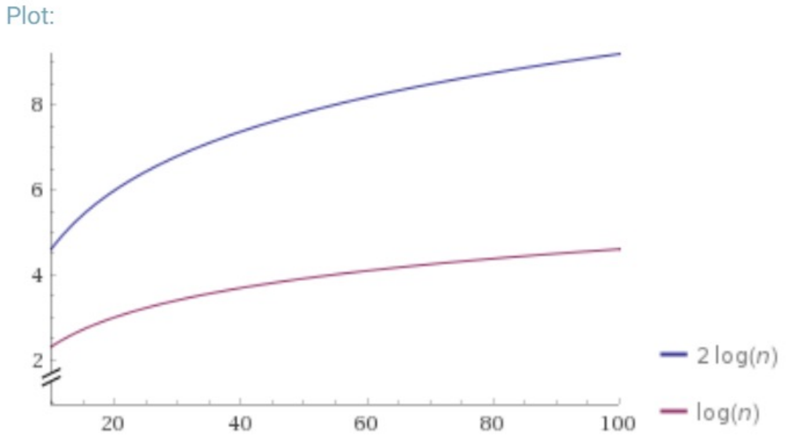
$$f(n) \leq c \cdot g(n)$$

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

Why n_0?



Why c?



Applying Big O Definition

Big-O

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

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

Show that $f(n) = 10n + 15$ is $O(n)$

Apply definition term by term

$$10n \leq c \cdot n \text{ when } c = 10 \text{ for all values of } n$$

$$15 \leq c \cdot n \text{ when } c = 15 \; for \; n \geq 1$$

Add up all your truths

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

Select values for c and n_0 and prove they fit the definition

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

$10n \leq 10n \; for \; all \; values \; of \; n$

$15 \leq 15n \; for \; n \geq 1$

So $10n + 15 \leq 25n$ for all $n \geq 1$, as required.

because a c and n_0 exist, $f(n)$ is $O(n)$