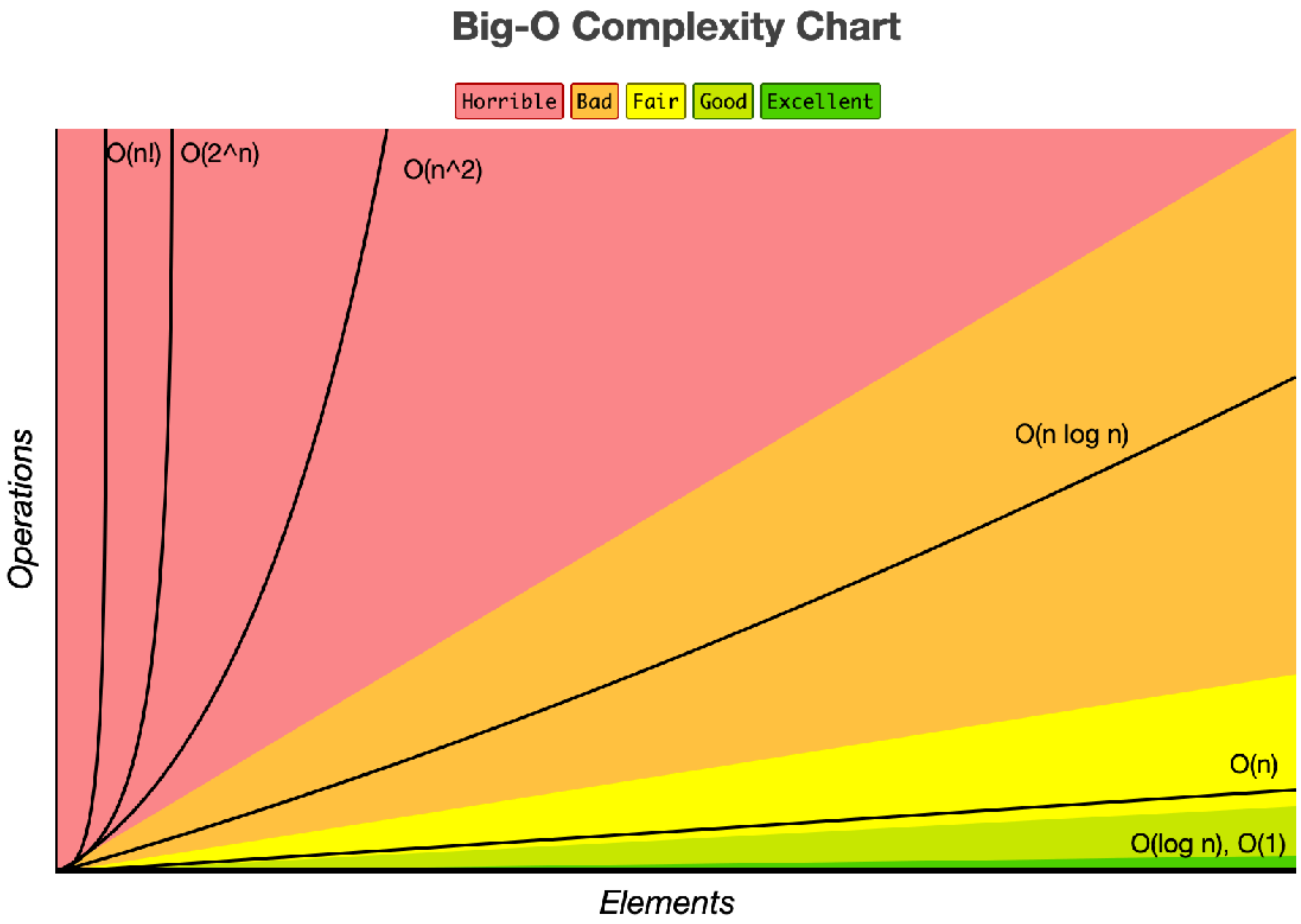
CS62 Class 5: Algorithmic Analysis

Basic Data Structures



https://www.bigocheatsheet.com/

Lecture 5 agenda

- (from last time) Finishing up ArrayLists
- Mathematical models of running time
- Order of growth classification
  - Big O (worst case), theta (average case), omega (best case)
- Amortized Analysis (via ArrayLists)

Removing (and returning) the last element

```java
/**
 * Removes and returns the element from the elementnd of the ArrayList.
 *
 * @return the removed E
 * @pre size>0
 */
public E remove() {
    if (isEmpty()){
        throw new NoSuchElementException("The list is empty");
    }
    size--;
    E element = data[size];
    data[size] = null;

    // Shrink to save space if possible
    if (size > 0 && size == data.length / 4){
        resize(data.length / 2);
    }

    return element;
}
```

Checking pre-condition

Remember our invariant that the last element is going to be at size - 1

Q: Why size == data.length / 4? Why not size <= data.length / 4?

A: Because we can only remove one element at a time, so it's guaranteed to eventually be equal

Clearing the ArrayList

```
/**
 * Clears the ArrayList of all elements.
 */
public void clear() {

    // Help garbage collector.
    for (int i = 0; i < size; i++){
        data[i] = null;
    }

    size = 0;
}
```

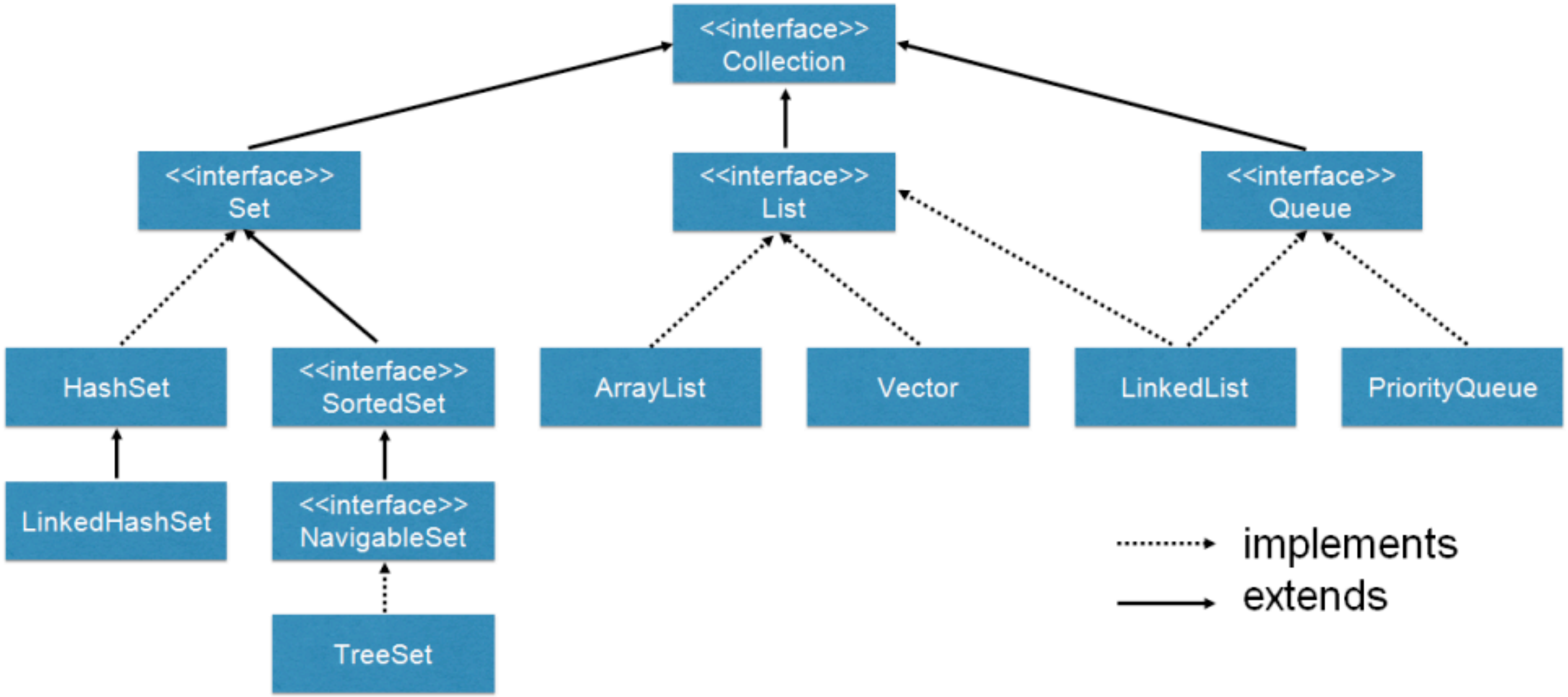
Note that we don’t need to call remove() many times - let’s avoid unnecessary computation.

Iterate through the underlying Array and set everything to null - prevent “loitering”

Update size

ArrayLists vs Vectors

Collection Interface



- Honestly, in the real world, not many people use ArrayLists. They prefer Vectors (e.g., most Leetcode problems in Java will use Vectors as “lists”)
- Vectors are slower, but *synchronized*, so they are memory safe.
- .push(), .pop() methods…we won’t learn them in this class, but telling you so you’re familiar in case they show up!

`ArrayList` in Java Collections

- Resizable list that increases by 50% when full and does NOT shrink.
- Not thread-safe (more in CS105).

```
java.util.ArrayList;

public class ArrayList<E> extends AbstractList<E> implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

`Vector` in Java Collections

- Java has one more class for resizable arrays.
- Doubles when full.
- Is synchronized (more in CS105).

```
java.util.Vector;

public class Vector<E> extends AbstractList<E> implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Mathematical models of running time

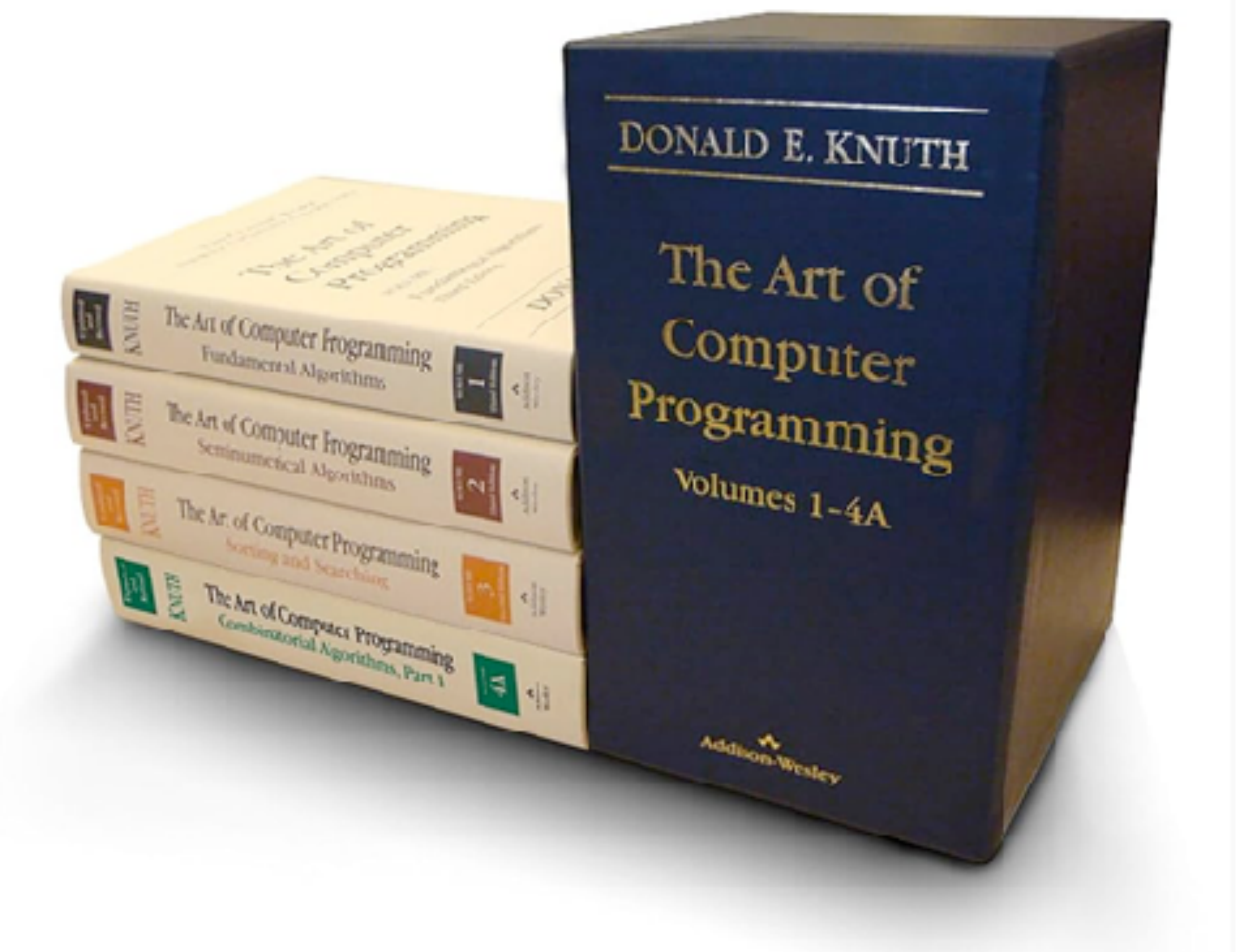
Code efficiency

- Efficiency comes in two flavors:
- Programming cost.
  - How long does it take to develop your programs?
  - How easy is it to read, modify, and maintain your code?
  - More important than you might think!
  - Majority of cost is in **maintenance**, not development!
- Execution cost (today).
  - How much time does your program take to execute?
  - How much memory does your program require?

What affects execution cost?

- System independent effects: Algorithm + input data
- System dependent effects: Hardware (e.g., CPU, memory, cache) + Software (e.g., compiler, garbage collector) + System (E.g., operating system, network, etc).

Total Running Time



- Popularized by Donald Knuth in the 60s in the four volumes of "The Art of Computer Programming".
  - Knuth won the Turing Award (The "Nobel" in CS) in 1974. (Read more in this week's textbook chapter! https://cs.pomona.edu/classes/cs62/history/bigO)

- In principle, accurate mathematical models for performance of algorithms are available.

- Total running time = **sum of cost x frequency** for all operations.
- Need to analyze program to determine the basic set of operations.
- Exact cost depends on the machine & compiler.
- Frequency depends on the algorithm & input data.

Cost of Basic Operations

- Add < integer multiply < integer divide < floating-point add < floating-point multiply < floating-point divide.

Operation	Example	Nanoseconds
Variable declaration	`int a`	c_1
Assignment statement	`a = b`	c_2
Integer comparison	`a < b`	c_3
Array element access	`a[i]`	c_4
Array length	`a.length`	c_5
Array allocation	`new int[n]`	$c_6 n$
string concatenation	`s+t`	$c_7 n$

Constant time (c_1 – c_5)

Linear time ($c_6 n$, $c_7 n$)

Example: 1-SUM (# of 0s in array)

- How many operations as a function of n?

```
int count = 0;
 for (int i = 0; i < n; i++) {
     if (a[i] == 0) {
         count++;
     }
 }
```

Operation	Frequency	
Variable declaration	2	count & i
Assignment	2	count & i
Less than	$n+1$	+1 is for loop exit
Equal to	n	each element
Array access	n	a[i]
Increment	n to $2n$	i++ and count++

Example: 2-SUM

- How many operations as a function of n?

```
int count = 0;
 for (int i = 0; i < n; i++) {
     for (int j = i+1; j < n; j++) {
         if (a[i] + a[j] == 0) {
             count++;
         }
     }
 }
```

Inner loop operations

when i=0, we do n comparisons with j
when i=1, we do n-1 comparisons with j
when i=2, we do n-2 comparisons with j
...
when i=n-1, we do 1 comparison with j

$$1 + 2 + 3 + \ldots + n = n(n+1)/2$$

Becoming too tedious to calculate!
(equal to, array access, increment:
exercise to the reader (answers next slide))

outer: n+1 (from i<n)
inner: n(n+1)/2 (from j<n)
adding these and doing factoring, we get (n+1)(n+2)/2 →

Operation	Frequency	
Variable declaration	$n + 2$	2 -> count & i; n -> j
Assignment	$n + 2$	2 -> count & i; n -> j
Less than	$(n+1)(n+2)/2$	
Equal to	$n(n-1)/2$	
Array access	$n(n-1)$	
Increment	$n(n+1)/2$ to n^2	

Example: 2-SUM

- How many operations as a function of n?

```
int count = 0;
 for (int i = 0; i < n; i++) {
     for (int j = i+1; j < n; j++) {
         if (a[i] + a[j] == 0) {
             count++;
         }
     }
 }
```

Operation	Frequency
Variable declaration	$n+2$
Assignment	$n+2$
Less than	$(n+1)(n+2)/2$
Equal to	$n(n-1)/2$
Array access	$n(n-1)$
Increment	$n(n+1)/2$ to n^2

Equals operations

when i=0, we do n-1 operations

when i=1, we do n-2

when i=2, we do n-3

…

when i=n-1, we do 0

$$0+1+\ldots+(n-1)=(n-1)(n-1+1)/2=n(n-1)/2$$

Array access operations

when i=0, we do 2(n-1) operations

when i=1, we do 2(n-2)

when i=2, we do 2(n-3)

…

when i=n-1, we do 0

$$2(0+1+\ldots+(n-1))=n(n-1)$$

Increment operations

outer loop -> n increments for i

inner loop -> n(n-1)/2 for j

count -> min: 0, max: n(n-1)/2

for the max count case,

$$n(n+1)/2+n(n-1)/2=n^2.$$

Tilde Notation

Recall: you learned this in 51P

- Estimate running time (or memory) as a function of input size n.
- Ignore lower order terms.
  - When n is large, lower order terms become negligible.

- Example 1: $\frac{1}{6}n^3 + 10n + 100 \quad \sim n^3$
- Example 2: $\frac{1}{6}n^3 + 100n^2 + 47 \quad \sim n^3$
- Example 3: $\frac{1}{6}n^3 + 100n^{\frac{2}{3}} + \frac{1/2}{n} \quad \sim n^3$

Simplification

- Cost model: Use some basic operation as proxy for running time. E.g., array accesses, which is the most expensive operation
- Combine it with tilde notation.
- $\sim n^2$ is the dominant (largest) term for the 2-SUM problem

Operation	Frequency	Tilde notation
Variable declaration	$n + 2$	$\sim n$
Assignment	$n + 2$	$\sim n$
Less than	$(n+1)(n+2)/2$	$\sim n^2$
Equal to	$n(n-1)/2$	$\sim n^2$
Array access	$n(n-1)$	$\sim n^2$
Increment	$n(n+1)/2$ to n^2	$\sim n^2$

Simplification Summary

- Ignore lower order terms.
- Ignore any coefficients.
- Convert dominant term in tilde notation table to worst case run-time.

Average-case Runtime ∊ Θ(N^2)

Operation	Frequency	Tilde notation
Variable declaration	$n+2$	$\sim n$
Assignment	$n+2$	$\sim n$
Less than	$(n+1)(n+2)/2$	$\sim n^2$
Equal to	$n(n-1)/2$	$\sim n^2$
Array access	$n(n-1)$	$\sim n^2$
Increment	$n(n+1)/2$ to n^2	$\sim n^2$

Order of growth classification

Types of analysis

- Best case: lower bound on cost (Omega, Ω)
  - What the goal of all inputs should be.
  - Often not realistic, only applies to "easiest" input.
- Worst case: upper bound on cost (Big O, O)
  - Guarantee on all inputs.
  - Calculated based on the "hardest" input.
- Average case: expected cost for random input (Theta, Θ)
  - A way to predict performance.
  - The "tightest" bound.
  - Not straightforward how we model random input.

Worst case analysis

- Definition: If $f(n) \sim cg(n)$ for some constant $c > 0$, then the order of growth of $f(n)$ is $g(n)$.
  - Ignore leading coefficients.
  - Ignore lower-order terms.

- We will be using the big-O (O) notation. For example:
  - $3n^3 + 2n + 7 = O(n^3)$
  - $2^n + n^2 = O(2^n)$
  - $1000 = O(1)$
- Yes, $3n^3 + 2n + 7 = O(n^6)$, but that's a rather useless bound.

Worksheet time!

- Use the Big O notation to simplify the following quantities:
- a. $n + 1$
- b. $1 + \frac{1}{n}$
- c. $(1 + \frac{1}{n})(1 + \frac{2}{n})$
- d. $2n^3 - 15n^2 + n$
- e. $\frac{\log(2n)}{\log(n)}$
- f. $\frac{\log(n^2 + 1)}{\log(n)}$

Need an algebra refresher? Check out the cheat sheet on the class homepage

From 1.4.5 of our recommended textbook

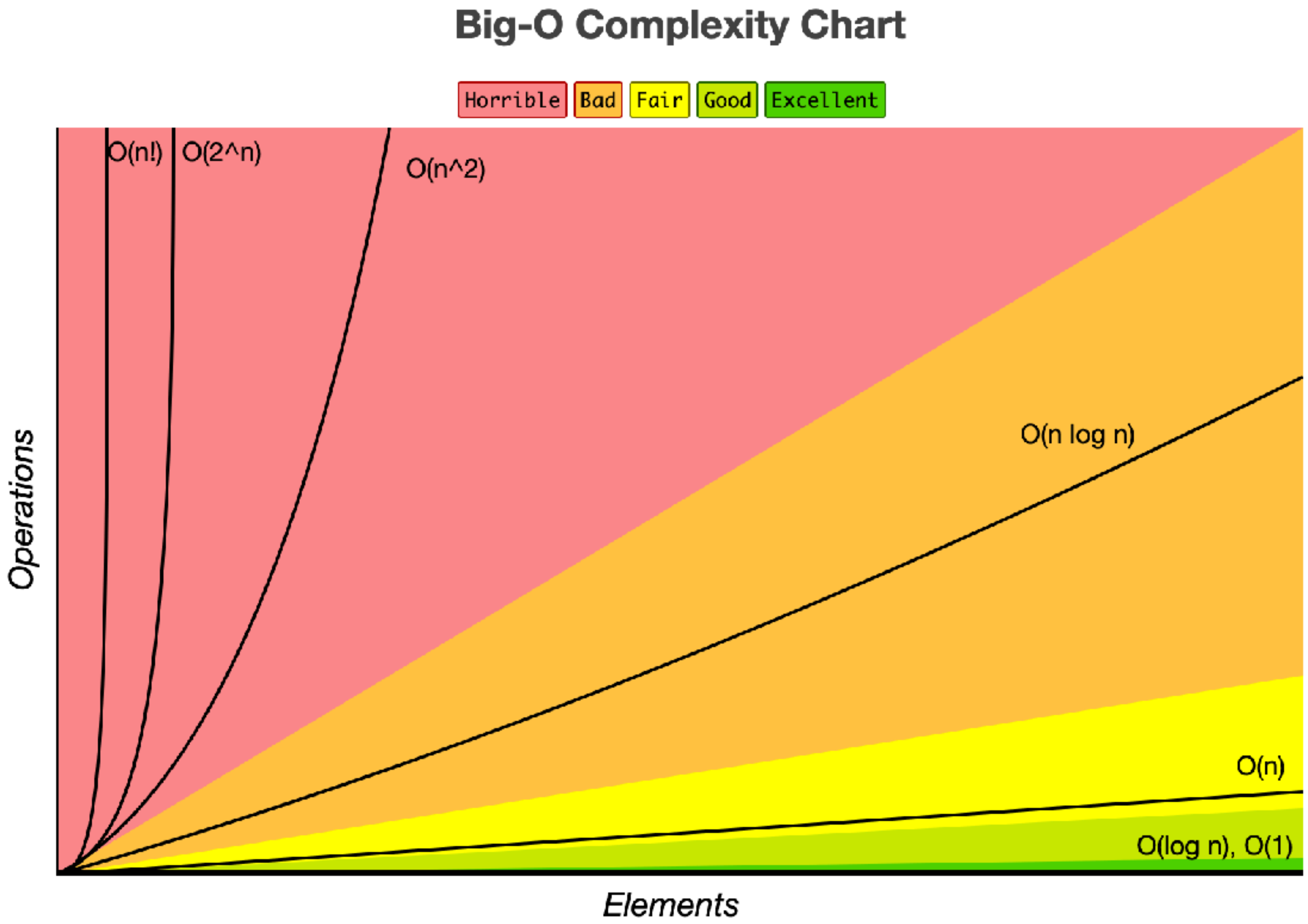
https://algs4.cs.princeton.edu/14analysis/

Worksheet answers

- Use the Big O notation to simplify the following quantities:
- a. $n + 1$ ~ $O(n)$
- b. $1 + \frac{1}{n}$ ~ $O(1)$
- c. $(1 + \frac{1}{n})(1 + \frac{2}{n})$ ~ O(1)
- d. $2n^3 - 15n^2 + n$ ~ $O(n^3)$
- e. $\frac{\log(2n)}{\log(n)} \sim \frac{\log(n)}{\log(n)}$ ~ O(1)
- f. $\frac{\log(n^2 + 1)}{\log(n)} \sim \frac{\log(n^2)}{\log(n)} \sim \frac{2\log(n)}{\log(n)}$ ~ 2 ~ O(1)

From slowest growing to fastest growing

$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$



https://www.bigocheatsheet.com/

Common order of growth classifications

- Good news: only a small number of function suffice to describe the order-of-growth of typical algorithms.
- 1: constant
  - Doubling the input size won't affect the running time. Holy-grail.
- $\log n$: logarithmic
  - Doubling the input size will increase the running time by a constant.
- n : linear
  - Doubling the input size will result to double the running time.
- $n \log n$: linearithmic
  - Doubling the input size will result to a bit longer than double the running time.
- n^2: quadratic
  - Doubling the input size will result to four times as much running time.
- n^3: cubic
  - Doubling the input size will result to eight times as much running time.
- 2^n: exponential
  - When you increase the input by some constant amount, the running time doubles.
- $n!$: factorial
  - When you increase the input, the running time grows proportional to the factorial of the input size.

Common order of growth classifications

This column is the doubling hypothesis: we'll explore more in a future lab

Order-of-growth	Name	Example code	$T(n)/T(n/2)$
1	Constant	`a[i]=b+c`	1
$\log n$	Logarithmic	`while(n>1){n=n/2;…}`	~ 1
n	Linear	`for(int i=0; i<n; i++)`	2
$n \log n$	Linearithmic	`for (i = 1; i <= n; i++){` `int x = n;` `while (x > 0)` `x -= i;` `}`	~ 2
n^2	Quadratic	`for(int i=0; i<n; i++) {` `for(int j=0; j<n; j++){`	4
n^3	Cubic	`for(int i=0; i<n; i++) {` `for(int j=0; j<n; j++){` `for(int k=0; k<n; k++){`	8

Useful approximations

- Harmonic sum: $1 + 1/2 + 1/3 + \ldots + 1/n$ $\sim$ $\ln n$
- Infinite geometric series: $n + n/2 + n/4 + \ldots + 1 = 2n - 1$ $\sim$ n
- Geometric sum: $1 + 2 + 4 + 8 + \ldots + n = 2n - 1$ $\sim$ n (n needs to be a power of 2)
- Triangular sum: $1 + 2 + 3 + \ldots + n$ $\sim$ n^2
- Binomial coefficients: $\binom{n}{k}$ $\sim$ $\frac{n^k}{k!}$ when k is a small constant.

- You don't need to memorize approximations; it's fine to Google them or use a tool like Wolfram alpha.
- Look at our math review handout!

Big-Theta: Formal Definition (Visualization)

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N0.

i.e. very large N

Example: $4N^2+N \in \Theta(N^2)$

- $R(N) = 4N^2+N$
- $f(N) = N^2$
- k1 = 3
- k2 = 5

Big O and Big Omega and Big Theta

Whereas Big Theta can informally be thought of as something like “equals”, Big O can be thought of as “less than or equal” and Big Omega can be thought of as "greater than or equal"

The following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in O(N^4)$
- $N^3 + 3N^4 \in O(N^6)$
- $N^3 + 3N^4 \in O(N^{N!})$
- $N^3 + 3N^4 \in \Omega(N^4)$
- $N^3 + 3N^4 \in \Omega(N^2)$
- $N^3 + 3N^4 \in \Omega(1)$

Big-Theta: Formal Definition

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big-O: Formal Definition

$$R(N) \in O(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big-Omega: Formal Definition

$$R(N) \in \Omega(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N)$$

for all values of N greater than some N_0.

i.e. very large N

Summary

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N).	$\Omega(N^2)$	$N^2/2$ $2N^2$ $N^{N!}$

Worksheet time!

Give the order of growth of the running time for the following code fragments as a function of *n*:

```
int count = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = j+1; k < n; k++) {
            if (a[i] + a[j] + a[k] == 0) {
                count++;
            }
        }
    }
}
```

```
int sum = 0;
for (int k=n; k>0; k/=2){
    for (int i=0; i<k; i++){
        sum++;
    }
}
```

Worksheet answers

Give the order of growth of the running time for the following code fragments as a function of *n*:

```
int count = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = j+1; k < n; k++) {
            if (a[i] + a[j] + a[k] == 0) {
                count++;
            }
        }
    }
}
```

- **Θ(n^3)**
  - three nested loops, constant time work in inner most loop
  - outer loop: n times
  - 2nd loop: n-i times
  - 3rd loop: n-j times

```
int sum = 0;
for (int k=n; k>0; k/=2){
    for (int i=0; i<k; i++){
        sum++;
    }
}
```

- **Θ(n)**
  - inner loop runs for n+n/2+n/4+…+1~ 2n ~ Θ(n) (geometric series)

Amortized Analysis (via ArrayLists)

Recall: add()

```
/**
 * Appends the element to the end of the ArrayList. Doubles its capacity if
 * necessary.
 *
 * @param element the element to be inserted
 */
public void add(E element) {
    if (size == data.length){
        resize(2 * data.length);
    }

    data[size] = element;
    size++;
}
```

Constant time operation (checking equality of 2 variables)

????

Constant time operations (variable assignment, accessing array, incrementing)

Recall: resize()

```java
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array of Es with the provided
    capacity
    E[] temp = (E[]) new Object[capacity];

    //copy all elements from old array (data) to temp array
    for (int i = 0; i < size; i++){
        temp[i] = data[i];
    }

    //point data to the new temp array
    data = temp;
}
```

O(n) run time to create a new empty Array

O(n) iterating through the array

O(1) assigning a pointer

Worst-case performance of $\mathtt{add}()$ is $O(n)$

- Cost model: 1 for insertion, n for copying n items to a new array.
- Worst-case: If ArrayList is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- Add is usually a constant time operation, unless we call resize, which takes O(n).
- Total cost: $n + 1 = O(n)$.

  resize() insertion

- Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

Amortized analysis

- Amortized cost per operation: for a sequence of n operations, it is the total cost of operations divided by n.
- Think of withdrawing money from your bank account, but then slowly spending the money bit by bit…even though you took out $100 at once, maybe you on average only spent $1 a day
  - Same thing with add(): We do a very expensive operation one time (resize), which opens up more space in the Array so we may subsequently do a bunch of cheap constant time operations

Amortized analysis for n `add()` operations

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Insertion Cost	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Copying Cost	0	1	2	0	4	0	0	0	8	0	0	0	0	0	0	0	16
Total Cost	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17

- As the ArrayList increases, doubling happens *half as often* but *costs twice as much.*
- O(total cost)= $\sum$("cost of insertions") + $\sum$("cost of copying")
- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") = $1 + 2 + 2^2 + \ldots + 2^{\log_2 n - 1} \leq 2n$.
- O(total cost) $\leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O^+(1)$, but "lumpy".

We'll see this more in a future lab

Lecture 5 wrap-up

- Today is the last day to make up Quiz 1. Come to OH right after class!! (You have 1 dropped quiz)
- Part I of Darwin (Species & World) released; more in lab
- Please read lab before coming to lab (Git)

Resources

- Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/
- History of Algorithmic Analysis: https://cs.pomona.edu/classes/cs62/history/bigO/
- More practice problems on class website
- More practice problems behind this slide (do the first one to prepare for the quiz tonight :))
- Exercise to the reader: what is the run time of other methods in ArrayList?

Last time review

- Interfaces are *blueprints* that say what methods a class that *implements* the interface should specify.
- Generics are "type placeholders" for when we want to ensure all the objects are of the same type, but we don't know what that type is until run time.
- ArrayLists are a special data structure that are resizable arrays. We implement them using arrays, but doubling their size when full, or halving their size when 1/4 full.

Last week review problem

```java
import java.util.ArrayList;

interface Storable {
    String getName();
    double getPrice();
}
class Product implements Storable {
    private String name;
    private double price;
    public Product(String name, double price){
        this.name = name;
        this.price = price;
    }
    public String getName(){return name;}
    public double getPrice(){return price;}
}
class Inventory<E extends Storable> {
    private ArrayList<E> items = new ArrayList<>();

    public void addItem(E item) {
        items.add(item);
    }

    public void removeItem(E item) {
        items.remove(item);
    }

    public void showInventory() {
        System.out.println("Inventory contains:");
        for (E item : items) {
            System.out.println("- " + item.getName());
        }
    }
}
```

This new syntax <E extends Storable> means the generic <E> has to implement Storable (so we know we can call getName)

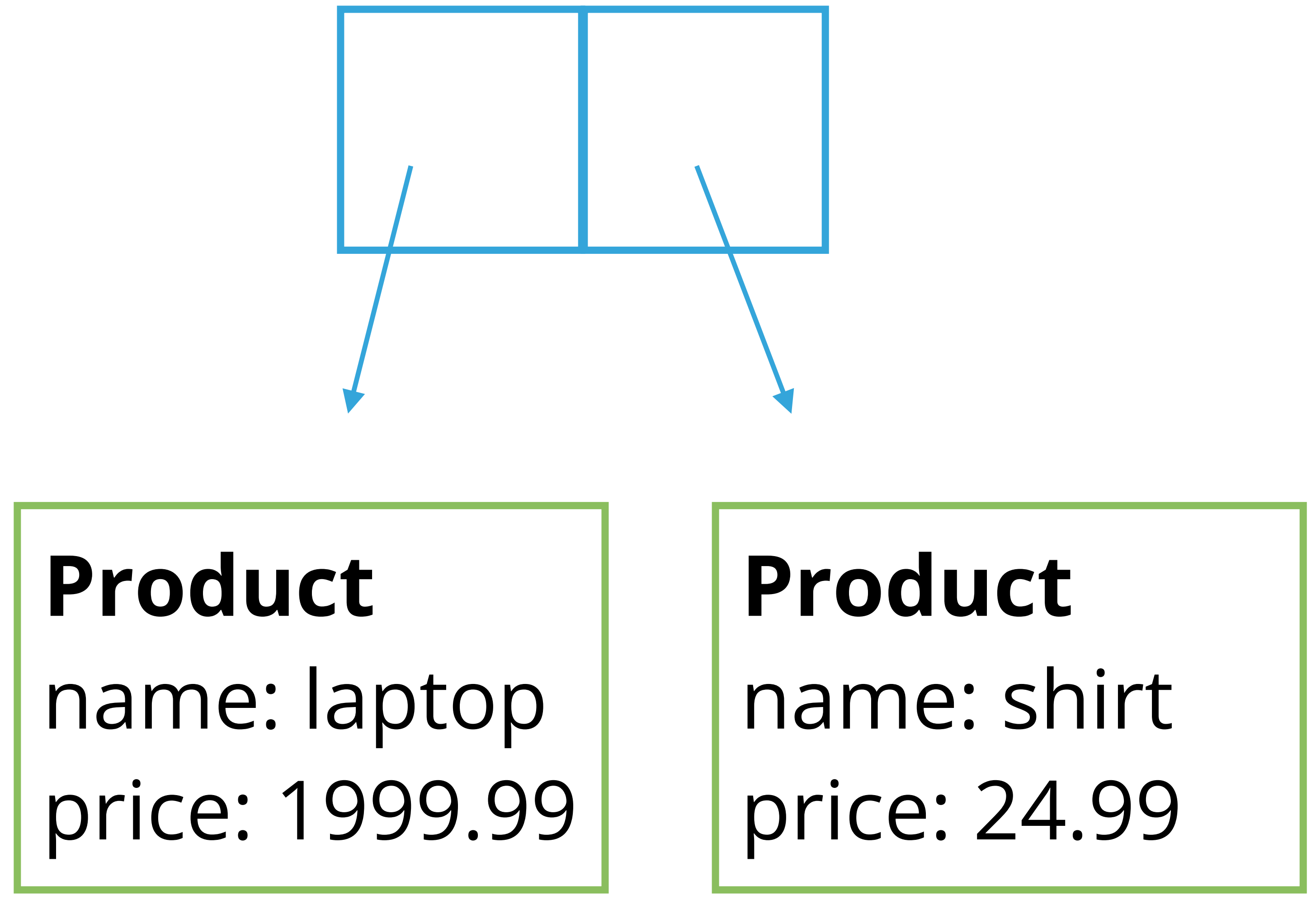
```java
public class ReviewProblem {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Inventory<Product> warehouse = new Inventory<>();
        Product laptop = new Product("laptop", 1999.99);
        warehouse.addItem(laptop);
        warehouse.addItem(new Product("shirt", 24.99));
        warehouse.showInventory();
        warehouse.addItem(new Product("headphones", 50.00));
        warehouse.showInventory();
        warehouse.removeItem(laptop);
        warehouse.showInventory();
    }
}
```

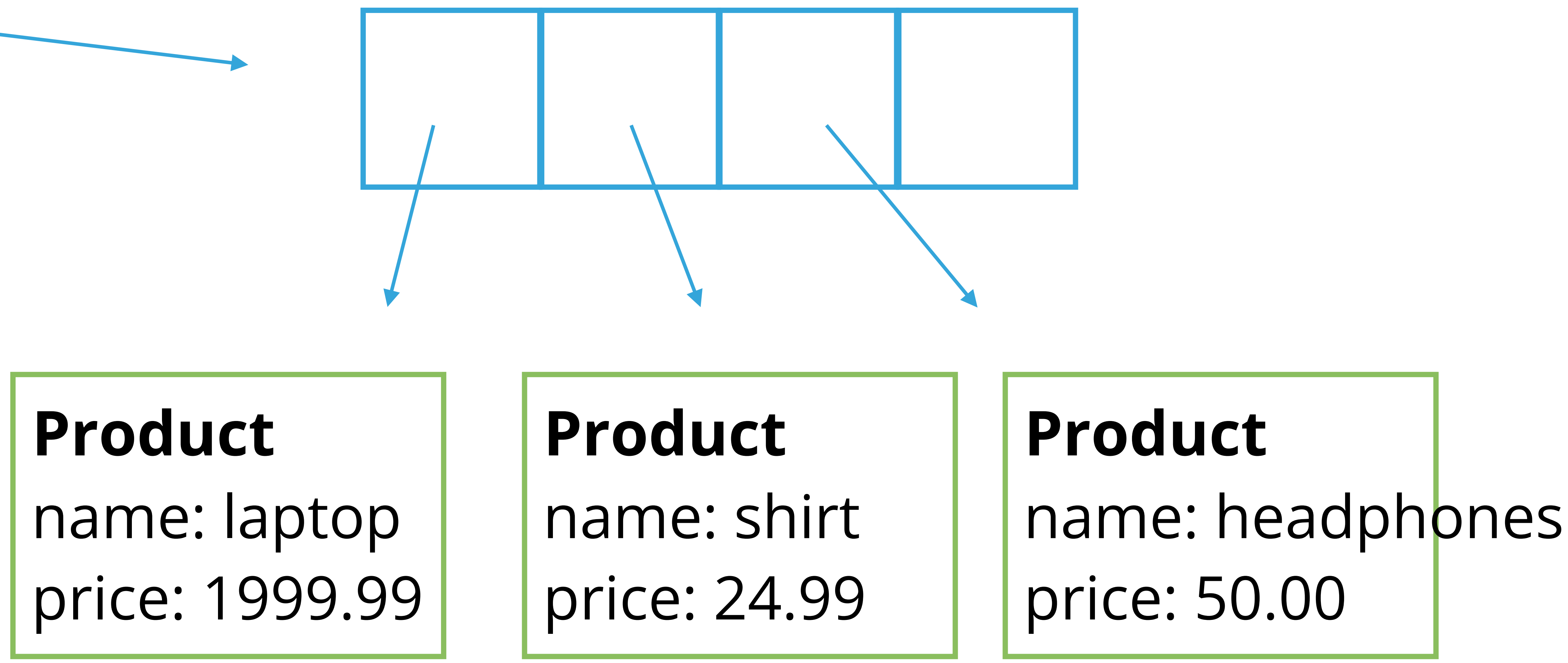
Step 0: Do you understand the code?

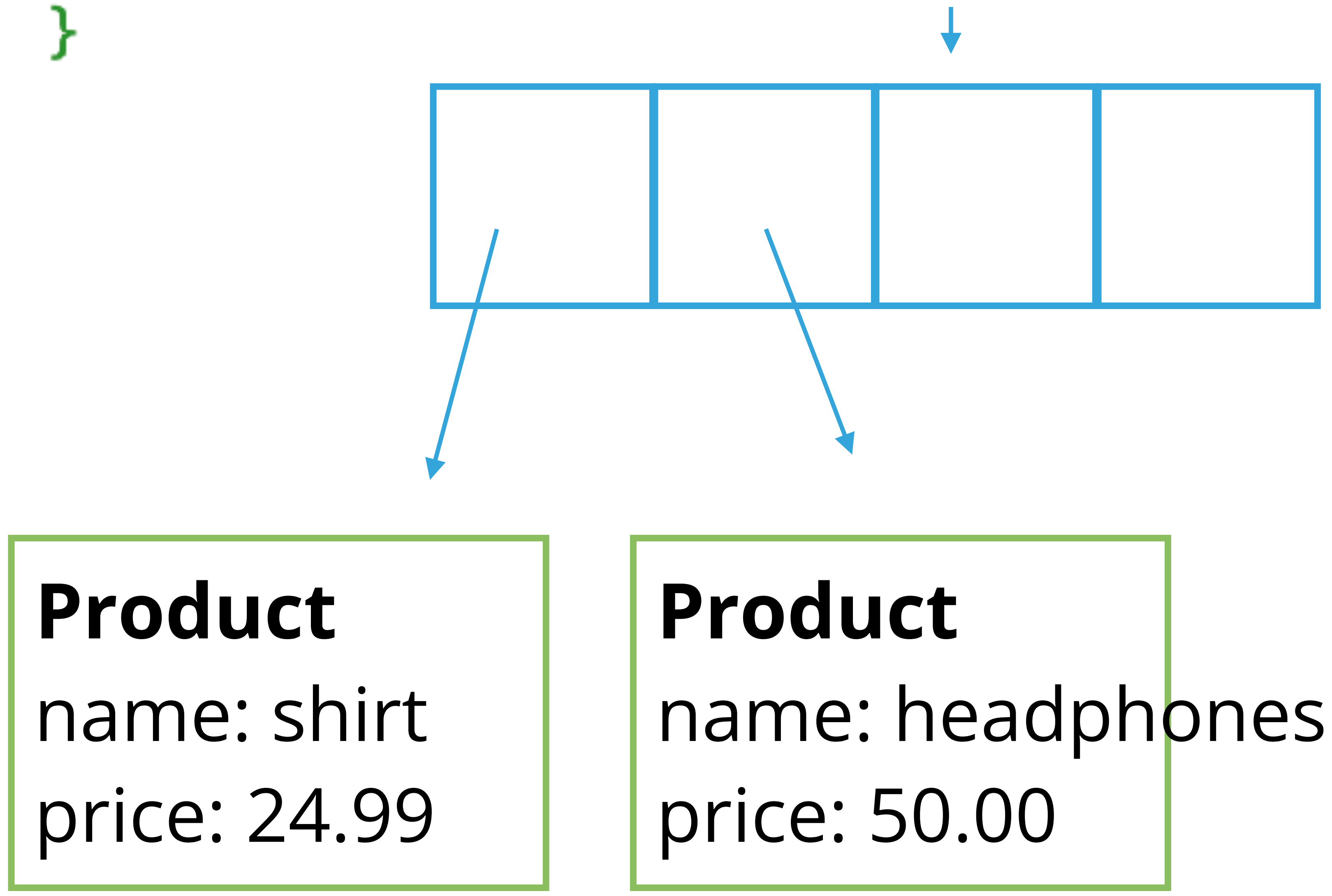
Step 1: Please draw the underlying ArrayList every time showInventory() is called.

Last week review problem answers

```java
public class ReviewProblem {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Inventory<Product> warehouse = new Inventory<>();
        Product laptop = new Product("laptop", 1999.99);
        warehouse.addItem(laptop);
        warehouse.addItem(new Product("shirt", 24.99));
        warehouse.showInventory();
        warehouse.addItem(new Product("headphones", 50.00));
        warehouse.showInventory();
        warehouse.removeItem(laptop);
        warehouse.showInventory();
    }
}
```







Order of Growth Exercise

Consider the functions below.

- Informally, what is the "shape" of each function for very large N?
- In other words, what is the order of growth of each function?

function	order of growth
$N^3 + 3N^4$	
$1/N + N^3$	
$1/N + 5$	
$Ne^N + N$	
$40 \sin(N) + 4N^2$	

Order of Growth Exercise

Consider the functions below.

- Informally, what is the "shape" of each function for very large N?
- In other words, what is the order of growth of each function?

- In "Big-Theta" notation we write this as $R(N) \in \Theta(f(N))$.
- Examples:
  - $N^3 + 3N^4 \in \Theta(N^4)$
  - $1/N + N^3 \in \Theta(N^3)$
  - $1/N + 5 \in \Theta(1)$
  - $Ne^N + N \in \Theta(Ne^N)$
  - $40 \sin(N) + 4N^2 \in \Theta(N^2)$

function	order of growth
$N^3 + 3N^4$	N^4
$1/N + N^3$	N^3
$1/N + 5$	1
$Ne^N + N$	Ne^N
$40 \sin(N) + 4N^2$	N^2