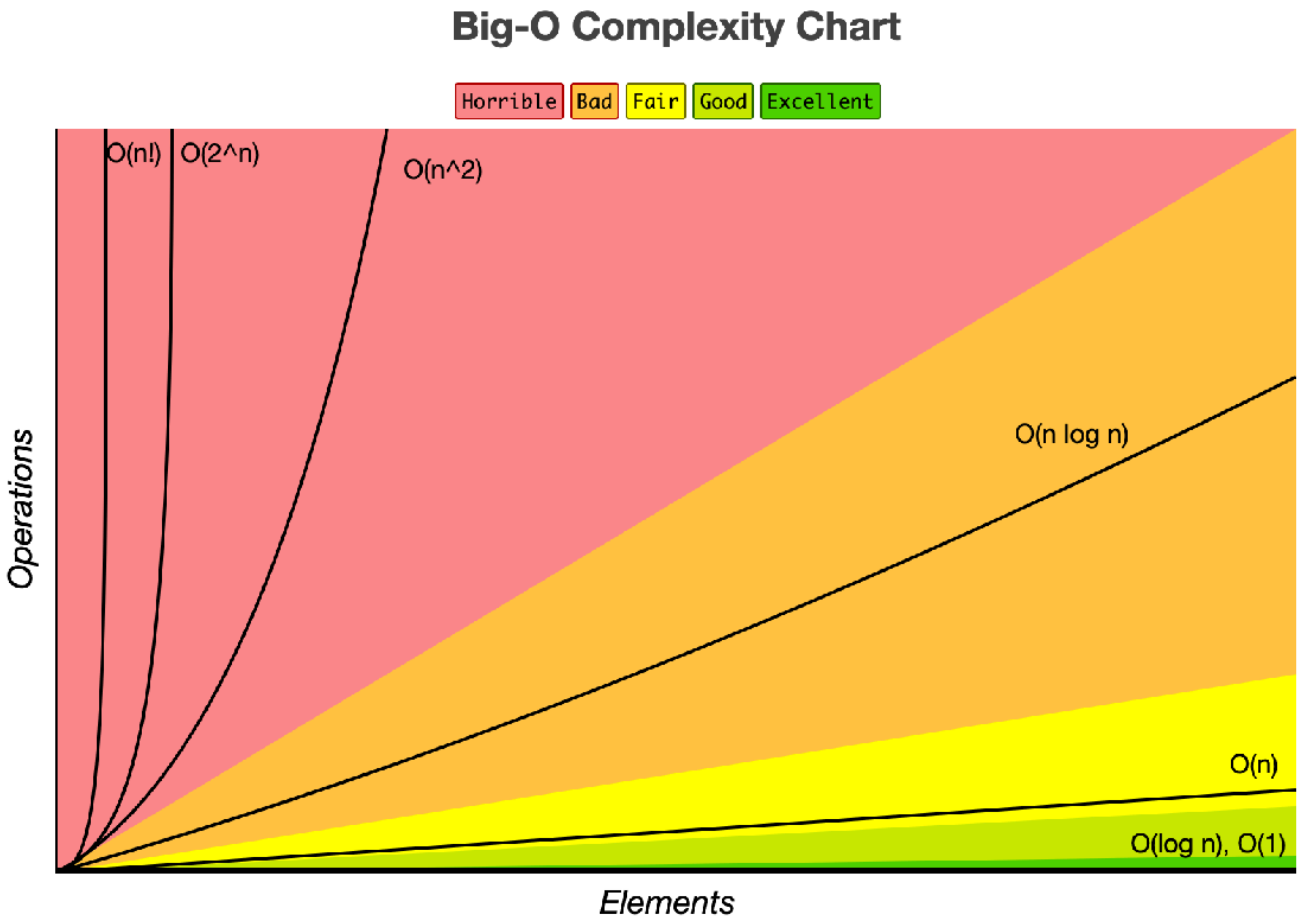
CS62 Class 7: Algorithmic Analysis

Basic Data Structures



https://www.bigocheatsheet.com/

Last week 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)

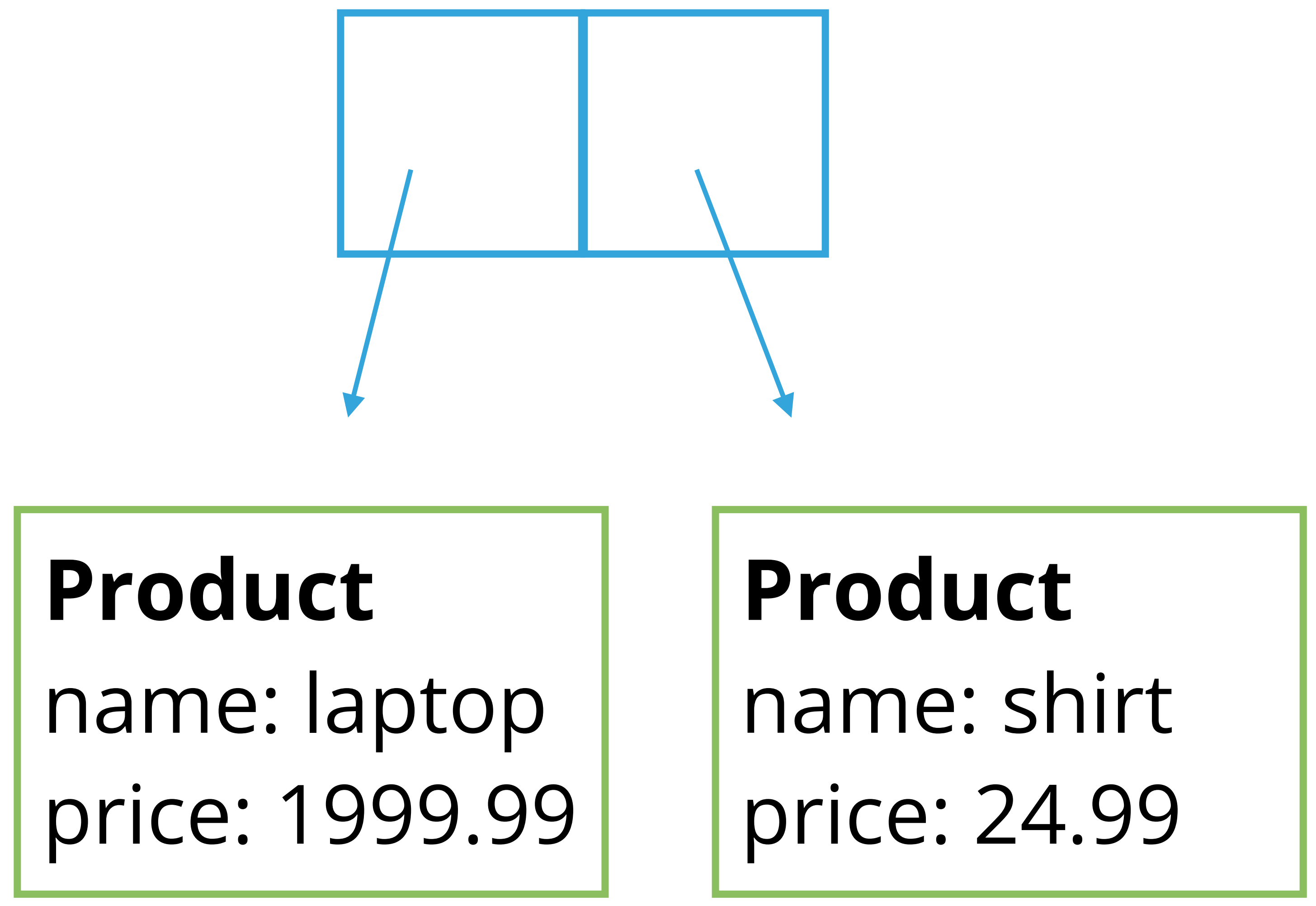
```java
public class ReviewProblem {
        Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Inventory<Product> warehouse = new Inventory<>();
        Product laptop = new Product("laptop", 1999.99);
        warehouse.addItem(laptop);
        warehouse.addItem(new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

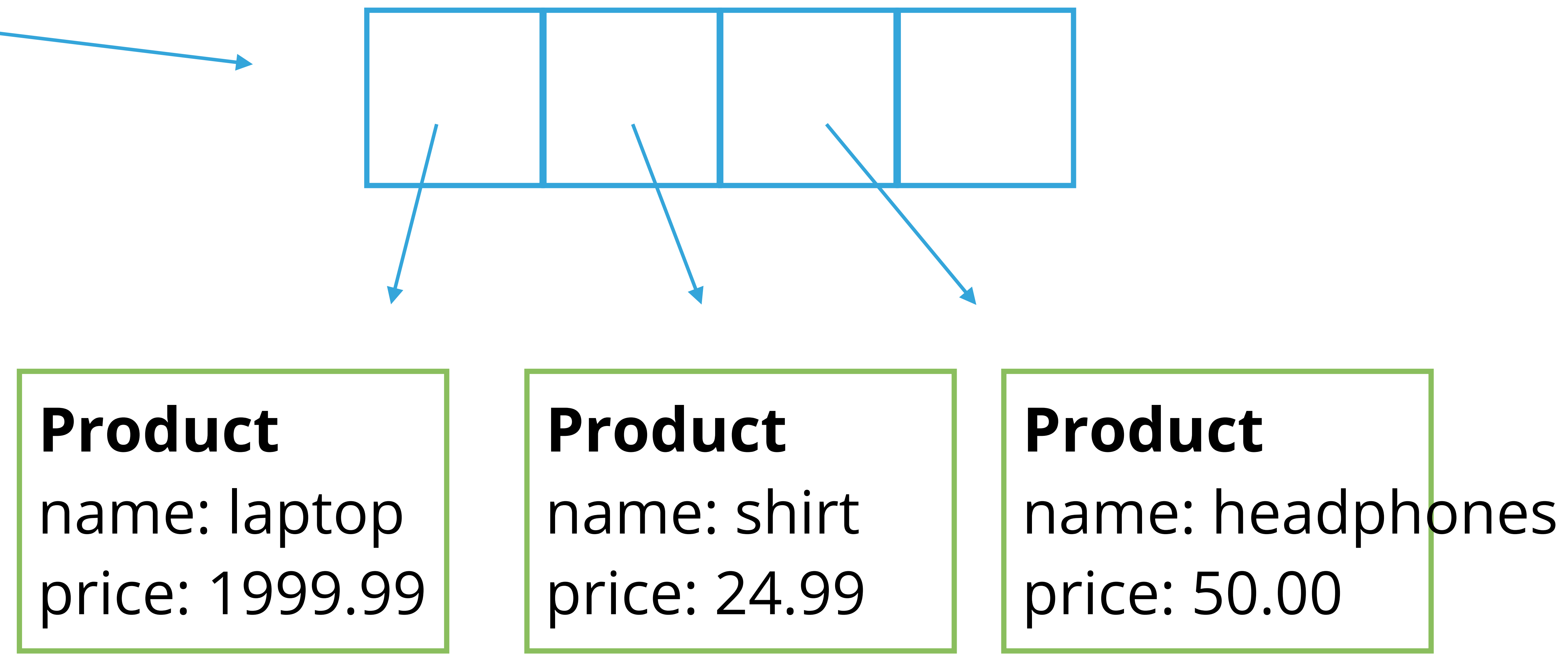
Step 0: Do you understand the code?

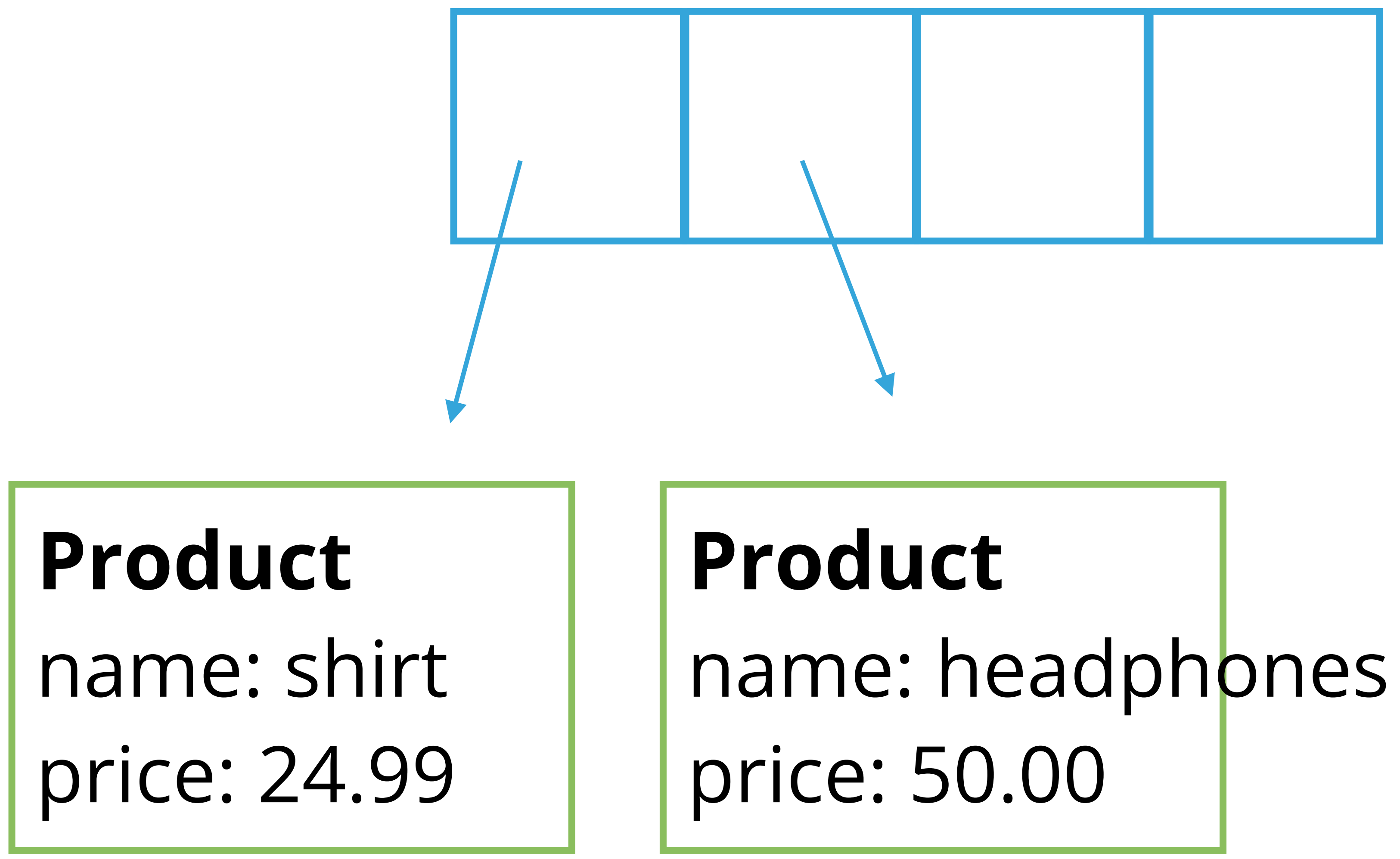
Step 1: Please draw the underlying ArrayList every time showInventory() is called.

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();
    }
}
```





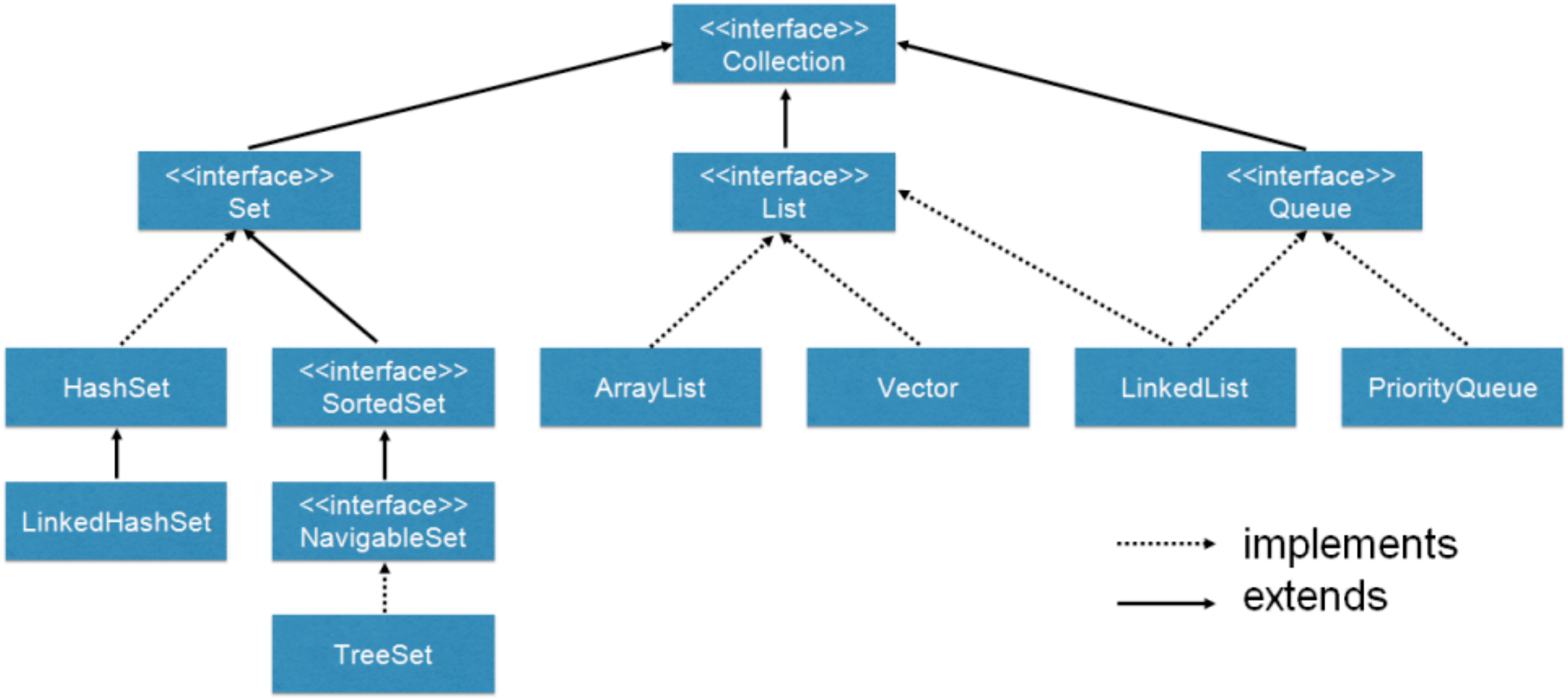


Lecture 7 agenda

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

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>
```

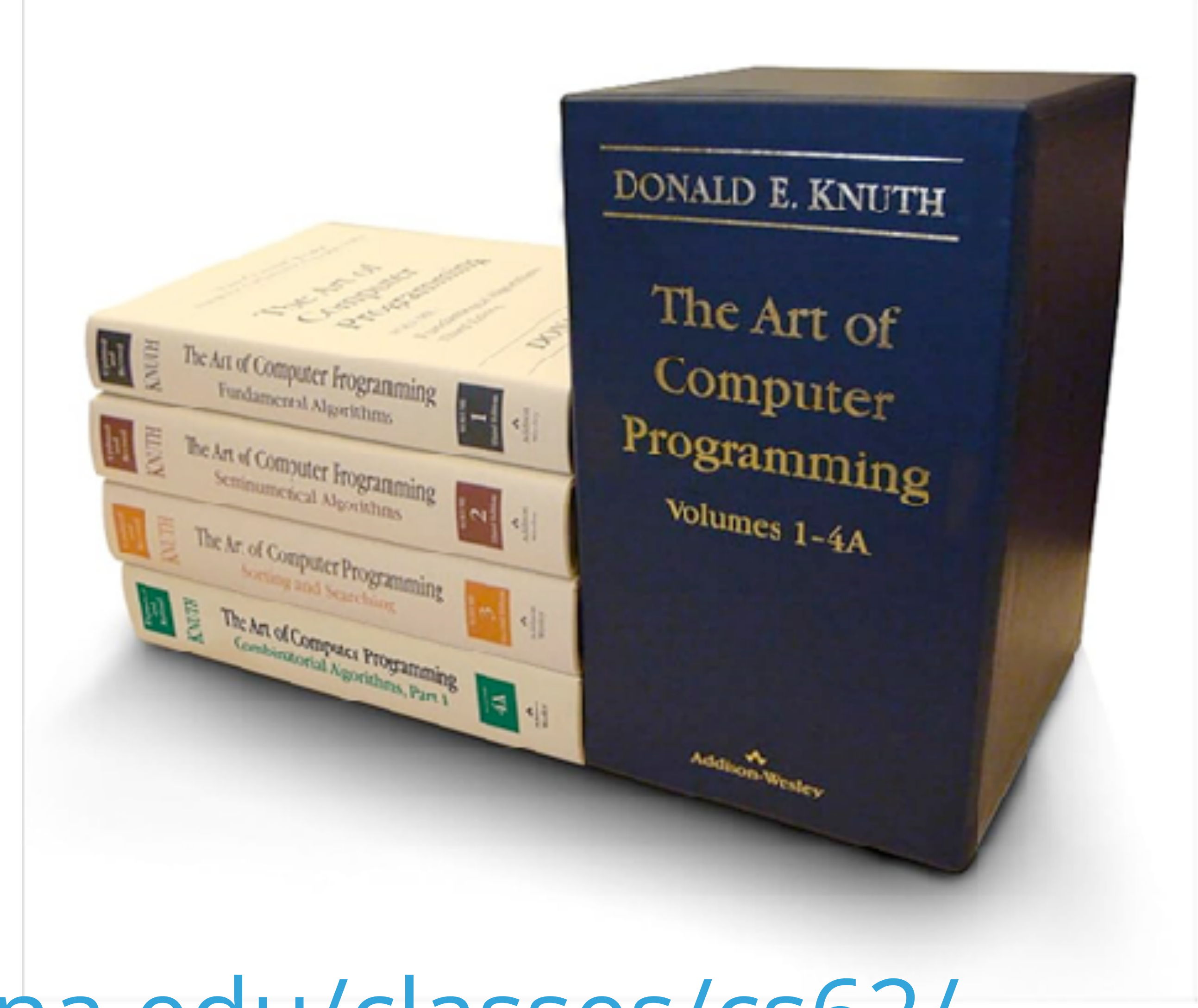
https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Mathematical models of running time

What affects performance?

- 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

Linear time

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 polynomial factoring, we get (n+1)(n+2)/2

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

2 -> count & i; n -> j
2 -> count & i; n -> j

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

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	$\sim n^2$

3-SUM problem, simplified

- Approximately how many array accesses as a function of input size 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++;
            }
        }
    }
}
```

- $\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n-1} 3 = 1/2n(n^2 - 3n + 2) \sim n^3$ array accesses.

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.
  - 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? Look at the Canvas resource: https://pomona.instructure.com/courses/4112/files?preview=261591

From 1.4.5 of our recommended textbook

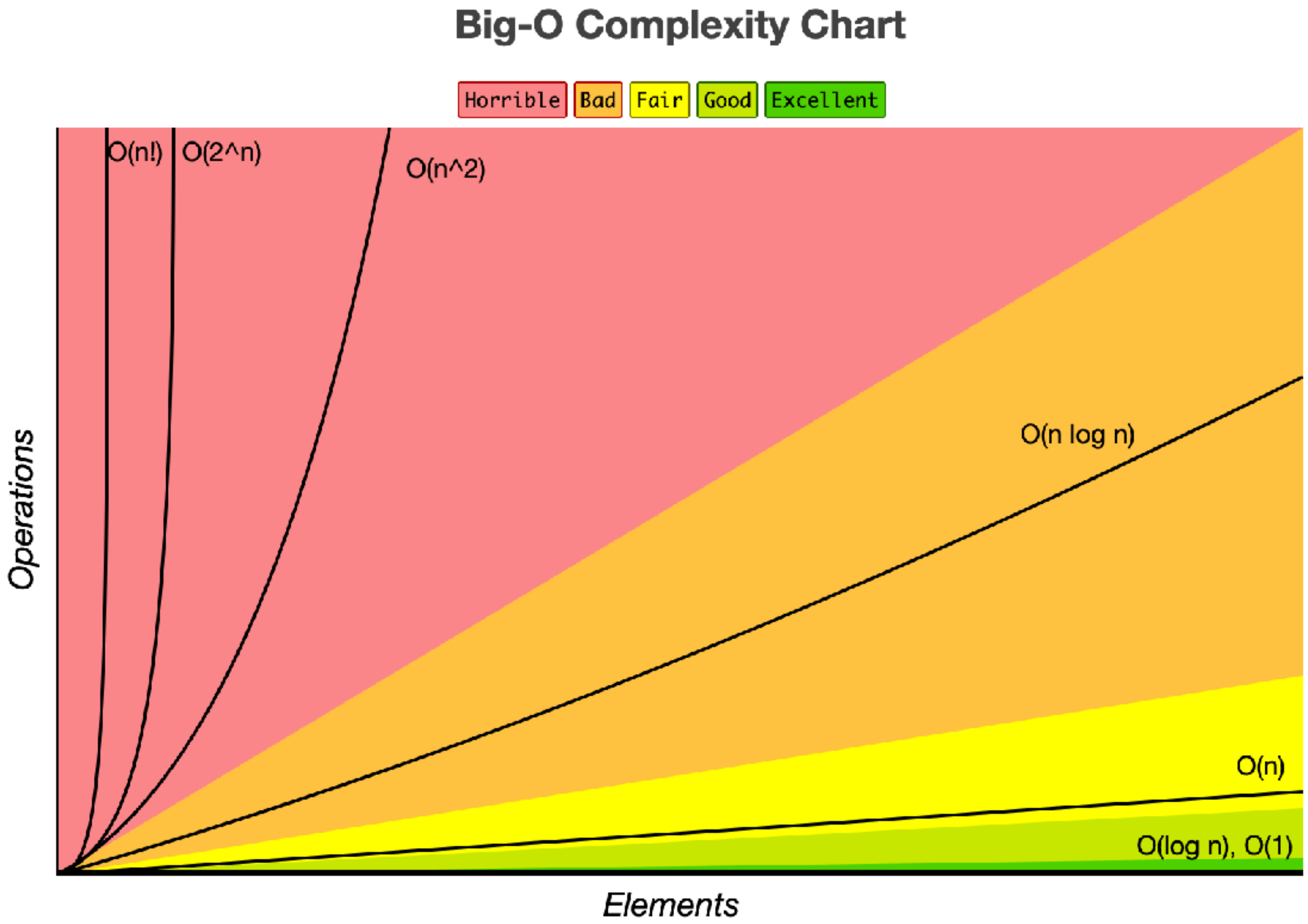
https://algs4.cs.princeton.edu/14analysis/

Worksheet answers

- Use the Big O notation to simplify the following quantities:
- a. $n + 1$ ~ $O(n)$
- b. $1 + \dfrac{1}{n}$ ~ $O(1)$
- c. $(1 + \dfrac{1}{n})(1 + \dfrac{2}{n})$ ~ O(1)
- d. $2n^3 - 15n^2 + n$ ~ $O(n^3)$
- e. $\dfrac{\log(2n)}{\log(n)} \sim \dfrac{\log(n)}{\log(n)}$ ~ O(1)
- f. $\dfrac{\log(n^2 + 1)}{\log(n)} \sim \dfrac{\log(n^2)}{\log(n)} \sim \dfrac{2\log(n)}{\log(n)}$ ~ 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 during tomorrow's 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: $1 + 1/2 + 1/3 + \ldots + 1/n \quad \sim \quad \ln n$
- Infinite geometric series: $n + n/2 + n/4 + \ldots + 1 = 2n - 1 \quad \sim \quad n$
- Geometric sum: $1 + 2 + 4 + 8 + \ldots + n = 2n - 1 \quad \sim \quad n$ (n needs to be a power of 2)
- Triangular sum: $1 + 2 + 3 + \ldots + n \quad \sim \quad n^2$
- Binomial coefficients: $\binom{n}{k} \quad \sim \quad \frac{n^k}{k!}$ 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 on Canvas!

Worksheet time!

- Give the order of growth of the running time for the following code fragment:

```
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 fragment:

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

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

Amortized Analysis (via ArrayLists)

Recall: add()

```java
/**
 * 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()

```
/**
 * 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 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.
- Total cost: $\underset{\text{resize()}}{n} + \underset{\text{insertion}}{1} = O(n)$.
- 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 $10 a day

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 $\leq \dfrac{3n}{n} = 3 = O^{+}(1)$, but “lumpy”.

We’ll see this in lab tomorrow!

Lecture 7 wrap-up

- Part I of Darwin (Species & World) due @ 11:59pm
  - Part II is even more challenging, please start early :)
  - Reminder if you use LLMs to help, you must cite it in your header file
- Lab (timing ArrayLists) released

Resources

- Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/
- History of Algorithmic Analysis: https://cs.pomona.edu/classes/cs62/history/bigO/
- On Canvas, more practice problems: https://pomona.instructure.com/courses/4112/files?preview=261592