CECS 274: Data Structures Introduction

California State University Long Beach

What is a Data Structure?

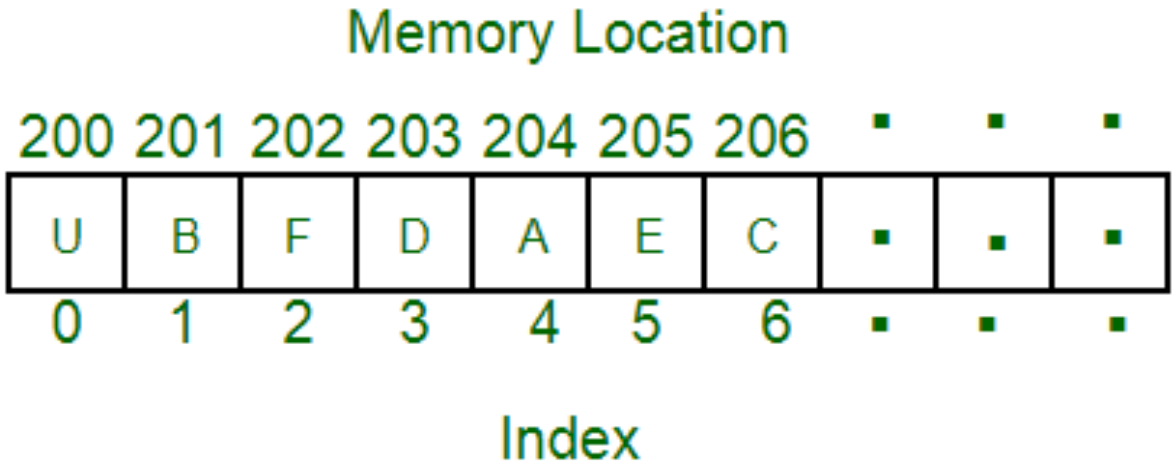
A data structure is a specialized design for storing, querying, removing, and manipulating data. Two or more data structures may support the same operations (e.g., adding a new element, removing an existing element, etc.); however, they will differ in the algorithms used to carry out the operations. Hence, not all data structures perform the same operations with the same efficiency. This course will explore several data structures, their efficiencies, and their applications.

Why are data structures important to learn?

Data structures are used everywhere in services and products that improve our quality of life. The success of products made by multi-million/billion dollar companies depends on correctly employing the appropriate data structures. Some examples include:

- Performing a web-search. (e.g., Google, Apple, Microsoft, etc.)
- Store, edit, search a contact on your phone (e.g., Google, Samsung, Apple, etc.)
- Open, store, move a file. File system data structures are used to locate the parts of that file on disk so they can be retrieved.
- Disaster preparedness services: storing/retrieving geospatial data (e.g., FEMA Geospatial Resource Center)

The Need for Efficiency: An Example (1/3)



Data can be stored in an array (i.e. an indexed collection of elements). Think about it: how do we search for or remove a particular item?

- One possibility: inspect each item in the array until we find a match. This implementation is straightforward, but not very efficient. Does this really matter?

The Need for Efficiency: An Example (2/3)

- Imagine an application with a moderately-sized data set, say of one million (10^6) items.
- Suppose you wish to locate all 10^6 items. To find the first item of interest, you would need to make, at worst, 10^6 inspections (e.g. assume you start looking at the beginning of the array and you search in order, but the item of interest is actually at the end of the array). Assuming the worst possible scenario for each item of interest we search for (i.e. every item we search for an item, we find it at the end of our search), how many inspections would it take to locate all 10^6 items?

The Need for Efficiency: An Example (3/3)

- Ans: $(10^6)(10^6) = 10^{12}$ (1 trillion) inspections!
- Processor speed: $\leq 10^9$ operations per second
- It will take at least $10^{12}/10^9 = 1000$ seconds ≈ 17 minutes.

Google indexes over 8.5 billion web pages. Imagine how long it would take to find a specific web page using our approach!

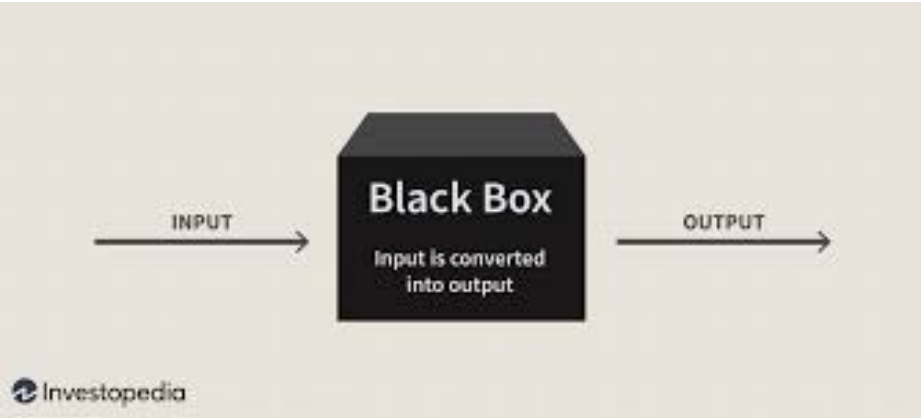
The solution is to organize data within a carefully designed data structure so that not every operation (search/add/modify/delete) requires every data item to be inspected.

Are data structures important to learn? **YES.**

Components of a Data Structure

A data structure consists of two components:

1. An **interface**: a list of named behaviors that describe what the data structure does.
2. **Implementation**: definitions and algorithms that execute the behaviors of the interface, i.e. the inner workings of the data structure.



Data Structure Analogy

Think of a data structure as a control remote...

- Interface: buttons (volume up/down, mute, channel up/down, etc.)
- Implementation: hardware + electrical mechanics (resistors, antennas, batteries, etc)

The Interface

- An interface is also referred to as an **Abstract Data Type** or **ADT**.
- An interface defines the set of operations supported by a data structure and the semantics, or meaning, of those operations.
- An interface only provides a list of supported operations along with specifications about what types of arguments each operation accepts and the value returned by each operation.

Implementation

- A data structure implementation includes the internal representation of the data structure as well as the definitions of the algorithms for each operations supported.
- There can be many implementations for a single interface.

Interface vs. Implementation: Java Example

```java
/**
 * This interface defines the basic functionalities of
 * a universal remote control
 * */
public interface Control {

    /**
     * abstract method; powers the controlled device ON or OFF
     * @return true if the device was powered ON, false otherwise
     * */
    public boolean power();


    /**
     * abstract method; increases volume in the controlled device
     * by 1 unit (unit to be determined by implementation)
     * @return true if the device was ON and volume was increased,
     * false otherwise
     * */
    public boolean volumeUP();

    /**
     * abstract method; decreases volume in the controlled device
     * by 1 unit (unit to be determined by implementation)
     * @return true if the device was ON and volume was decreased,
     * false otherwise
     * */
    public boolean volumeDOWN();


}
```

Interface vs. Implementation: Java Example

Interface Control

`public interface` **`Control`**

This interface defines the basic functionalities of a universal remote control

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
`boolean`	**`power`**`()`	abstract method; powers the controlled device ON or OFF
`boolean`	**`volumeDOWN`**`()`	abstract method; decreases volume in the controlled device by 1 unit (unit to be determined by implementation)
`boolean`	**`volumeUP`**`()`	abstract method; increases volume in the controlled device by 1 unit (unit to be determined by implementation)

Interface vs. Implementation: Java Example

```java
public class TVControl implements Control{
    //instance and class attributes
    private boolean isON;
    private int volume;
    public final int defaultVolume = 15;

    /**
     * default constructor initializes a TVControl object
     * with attributes to determine if device is ON or OFF,
     * and to determine volume.
     * **/
    public TVControl() {
        isON = false;
        volume = defaultVolume;
    }

    @Override
    public boolean power() {
        this.isON = !isON;
        return isON;
    }

    @Override
    public boolean volumeUP() {
        if (isON) {
            this.volume += 1;
        }
        return isON;
    }

    @Override
    public boolean volumeDOWN() {
        if (isON) {
            this.volume -= 1;
        }
        return isON;
    }
```

Interface vs. Implementation: Python Example

```python
class Controller:
    """
        This interface defines the basic functionalities of a universal remote control
    """

    @abstractmethod
    def power(self):
        """
            powers the controlled device ON or OFF
            @:return True if the device was powered ON, False otherwise
        """
        raise NotImplementedError("Subclasses should implement this!")

    @abstractmethod
    def volume_up(self):
        """
        increases volume in the controlled device by 1 unit
        (unit to be determined by implementation)
        :return: True if the device was ON and volume was increased, False otherwise.
        """
        raise NotImplementedError("Subclasses should implement this!")

    @abstractmethod
    def volume_down(self):
        """
        decreases volume in the controlled device by 1 unit
        (unit to be determined by implementation)
        :return: True if the device was ON and volume was decreased, False otherwise.
        """
        raise NotImplementedError("Subclasses should implement this!")
```

Interface vs. Implementation: Python Example

```python
class TVController(Controller):
    """
    This class instantiates TVController object with the basic
    functionalities to turn ON/OFF controlled device, and
    increase/decrease volume.  TVController implements the
    Controller interface.
    """
    default_volume = 15

    def __init__(self):
        """
        default constructor initializes TVControl object with attributes
        to determine if device is ON or OFF, and to determine volume
        """
        self.is_on = False
        self.volume = TVController.default_volume

    def power(self):
        self.is_on = not self.is_on
        return self.is_on

    def volume_up(self):
        if self.is_on:
            self.volume += 1
        return self.is_on

    def volume_down(self):
        if self.is_on:
            self.volume -= 1
        return self.is_on
```

Interface vs. Implementation

The main difference between an interface and its implementation is...

- The interface describes **WHAT** a data structure does.
- The implementation describes **HOW** the data structure does it.

Efficiency Analysis

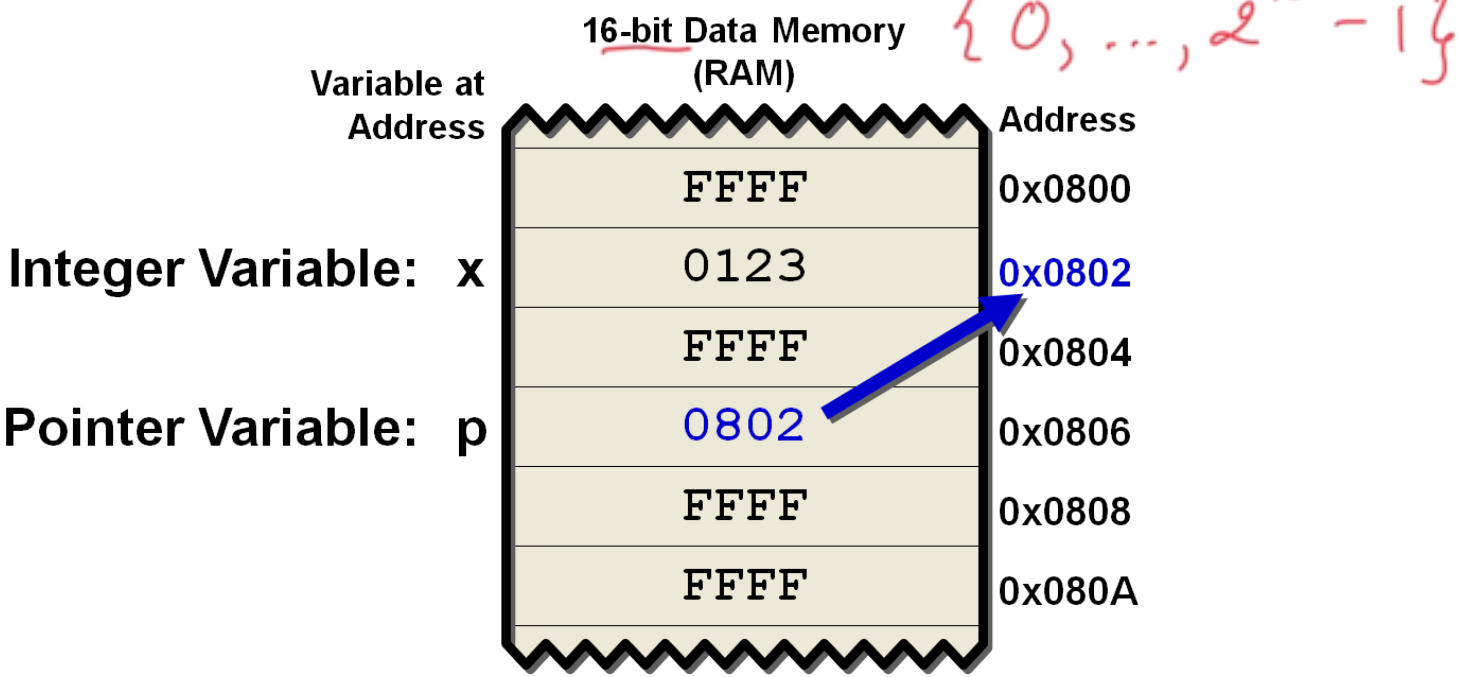
When analyzing the efficiency of a data structure, two criteria are considered:

- **Time complexity**: The running times of operations on the data structure should be as small as possible.
- **Space complexity**: The data structure should use as little memory as possible.

In our course, we will focus mainly on time complexity analysis, also referred to as run-time analysis.

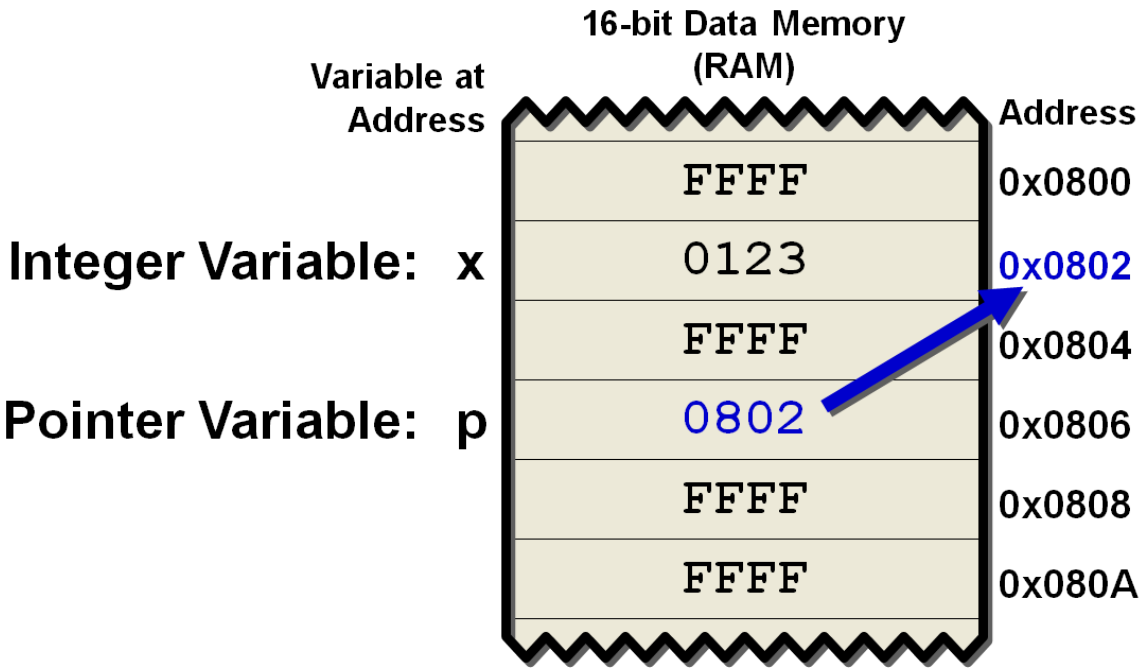
Mode of Computation (1/2)

- We use the ω-bit word-RAM model (Random Access Machine)
- In this model, we have access to a random access memory consisting of cells, each of which stores a ω-bit word.
- This implies that a memory cell can represent, for example, any integer in the set $\{0, ..., 2^{\omega-1}\}$



Mode of Computation (2/2)

- In the word-RAM model, basic operations on words take constant time (arithmetic, comparison, bitwise operations).
- Any cell can be read or written in constant time.



3 Different Types of Run-time Guarantees

Run-time refers to the time necessary to execute a certain program or algorithm.

- **Best-case:** The run-time of an algorithm when data circumstances are ideal and algorithm takes minimal time to operate.
- **Expected-case**: The run-time of an algorithm when data circumstances are typical.
- **Worst-case**: The run-time of an algorithm when data circumstances give rise the largest possible number of computations that the algorithm can make.

Measuring Run-time (1/2)

Major caveat of measuring run-time: we do not measure run-time using units of time.

The reason? Different machines have different processor speeds and can take significantly different times to run the same algorithm.

Measuring Run-time (2/2)

Instead, run-time is discussed in terms of the number of floating-point operations (i.e., arithmetic or assignment operations) that are necessary to execute the algorithm. Since the number of operations will depend on the number of visits that are made to each element of the data set, and since counting visits is easier than counting operations, we do the following:

- use a function $g(n)$ to represent the actual number of operations.
- use a function $f(n)$ to represent the number of visits made to complete the algorithm.
- use $f(n)$ times some constant as an estimate for the real run-time $g(n)$.

Example: Counting Visits

For example, suppose we wish to estimate the run-time of the following function which takes in an array of n lists and finds the minimum value of each list. The length of each list is at most n too:

```
def find_minima(arr):
    minima = []
    for lst in arr:
        minimum = lst[0]
        for i in range(1, len(lst)):
            if lst[i] < minimum:
                minimum = lst[i]
        minima.append(minimum)
    return minima
```

n lists

n iterations

1 visit

(n-1) iterations

1 visit

1 visit

2 visits

n * (1 + (n-1))

Total visits = $n * (1 \text{ visit} + (\text{inner for-loop visits}))$

$= n * (1 + (n-1) * 2) = n(2n-1) = 2n^2 - n$

Asymptotic Behavior (1/2)

Since counting the number of visits to elements in a data set is just an approximation for the run-time, it is not even necessary to consider *lower-order terms*. Lower-order terms refers to the terms in a polynomial or other function $f(n)$ that do not have a significant effect on the value of f for large n. This is referred to as analyzing the as the ***asymptotic behavior*** of the function. For example, if you consider the value of each term in

$$f(n) = 2n^2 - n$$

for increasing values of n, what do you notice?

	$2n^2$	$-n$
$n = 1$	2	-1
$n = 100$	20,000	-100
$n = 1000$	2,000,000	-1000

$f(n) = 1$

$f(n) = 19900$

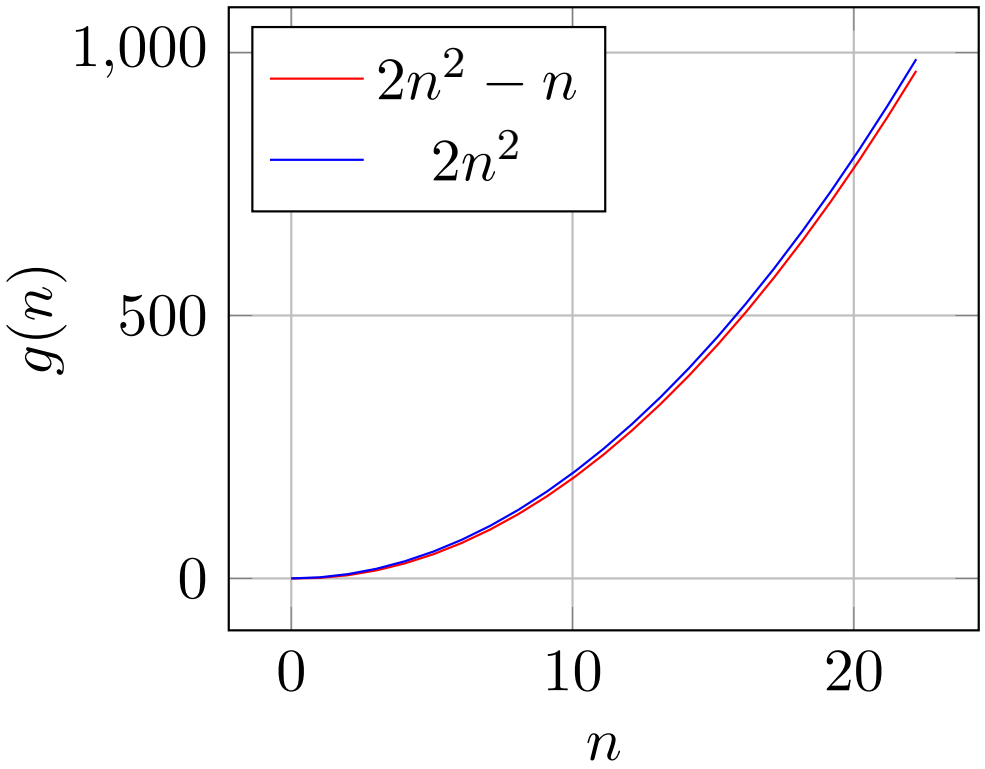
$f(n) = 1999000$

Asymptotic Behavior (1/2)

The table on the previous slide shows us that in

$$f(n) = 2n^2 - n$$

the dominant term, i.e. the term that affects the value of f the most for increasing values of n, is the $2n^2$ term.



Asymptotic Behavior & Big-O

Recall that the true run-time, $g(n)$, is approximated by the number of visits, $f(n)$, times some constant. So we say,

real run-time $\approx$ const * number of visits

$$g(n) \approx \text{const} \cdot f(n)$$

But, now, we also say that $f(n)$ does not need to be precise because lower-order terms have little effect on the value of f for growing values of n. Big-O is a concept that combines the two ideas.

Big-O

The definition of Big-O is as follows:

Let $g(n)$ and $f(n)$ be functions defined for integers $n \geq 0$.

We say $g(n)$ is $O(f(n))$, denoted $g(n) = O(f(n))$ if there are constants C and k, such that for $n \geq k$,

$$|g(n)| \leq C|f(n)|$$

i.e., the function $g(n)$ is bounded above (and below) by some constant multiple of $f(n)$, when n grows larger than k

Big-O Definition Example

For our example, notice that for $n \geq 1$

$$-2n^2 \leq \underbrace{2n^2}_{f(n)} - n \leq 2n^2$$

$$\implies |2n^2 - n| \leq Cn^2 \text{ for } n > k$$

where $C = 2$ and $k = 1$

Thus, $2n^2 - n = O(n^2)$

Since the run-time $g(n)$ is approximately $2n^2 - n$ times some constant, we can also say that $g(n) = O(n^2)$.

$f(n) = 5n + 1$

$|5n + 1| \leq 5 \cdot n$

$C = 5 > k = 1$

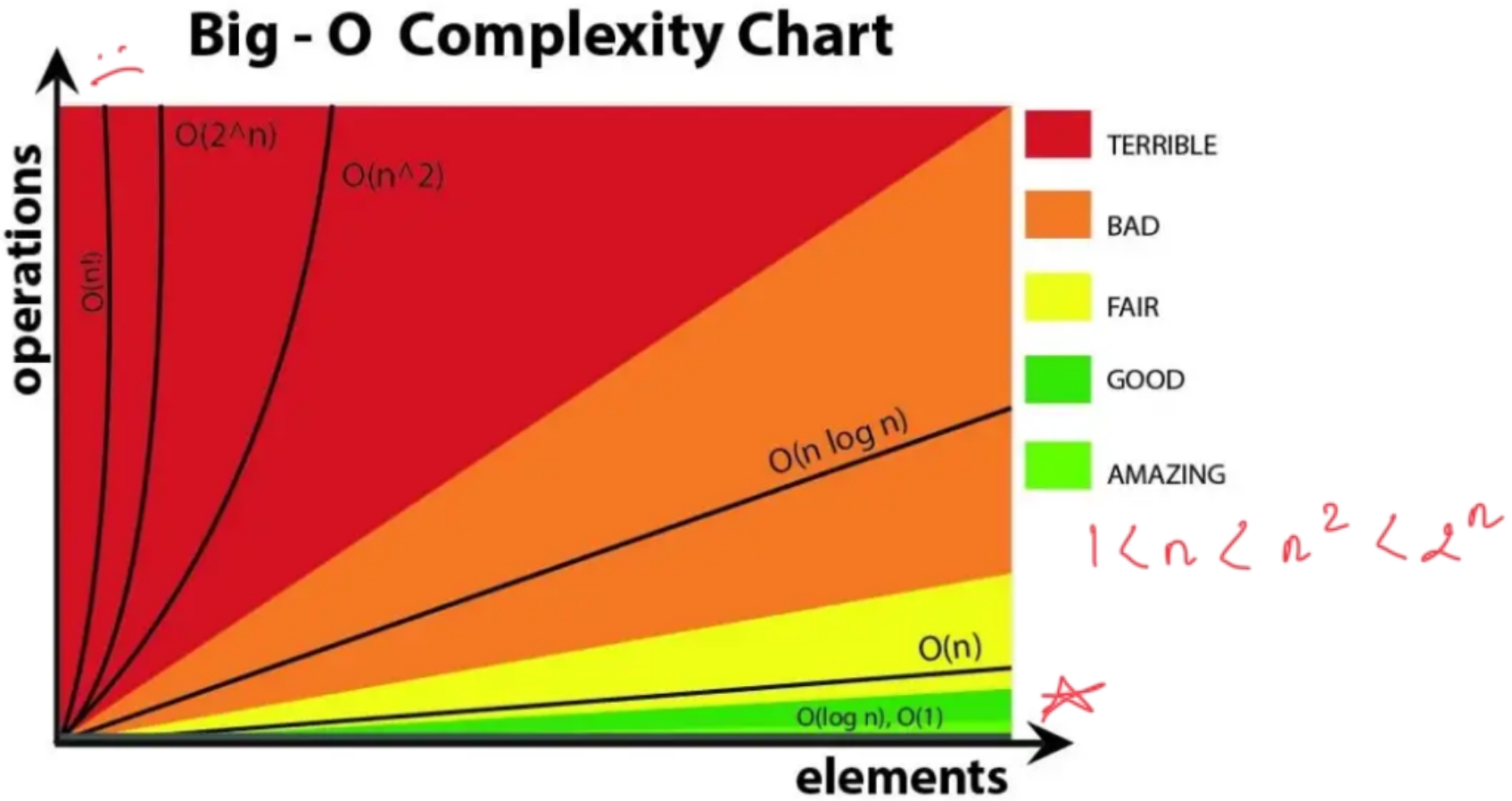
$\Rightarrow g(n) = O(n)$

Shortcut Version of Finding Big-O Run-time

In practice, we always consider the Big-O run-time approximation of an algorithm, but we do not always rigorously prove our result using the Big-O definition. Instead, we take the following shortcuts to find the Big-O run-time:

1. Count the total number of visits the algorithm takes. This should produce a function $f(n)$. $= 5n^3 - 2n^2 + n$
2. Drop the lower-order terms. $f(n) = 5n^3 - 2n^2 + n$
3. Drop the coefficient of the dominant term (if it is anything other than 1). $f(n) = 5n^3$
4. The result is the Big-O run-time. $g(n) = O(n^3)$

Most- to Least-Efficient Run-times



Credit: https://betterprogramming.pub

Most- to Least-Efficient Run-times

1. $O(1)$ - constant time
2. $O(\log n)$ - constant time
3. $O(n)$ - linear time
4. $O(n \log n)$ - log-linear time
5. $O(n^p)$ - polynomial time $p \geq 2$
6. $O(2^n)$ - exponential time
7. $O(n!)$ - factorial time

$O(n^2)$: quadratic time
$O(n^3)$: cubic time

Big-O Run-time Example 1

The function below returns a dictionary of the number of times each value appears in the given array. What is its Big-O, worst-case run-time?

```
def counts(arr):
    c = {}
    for ele in arr:
        if ele not in c:
            count = 0
            for i in range(len(arr)):
                if arr[i] == ele:
                    count += 1
            c[ele] = count
    return c
```

n

n

n

1 visit / iteration

1 visit

$f(n) = n * (1 + n * 1) = n + n^2 \Rightarrow g(n) = O(n^2)$

Big-O Run-time Example 2

Which of the following are equivalent are $O(n^2)$?

(a) $f(n) = 3n^2 + 4n - 1$

(b) $f(n) = n \log n + 2n$

(c) $f(n) = n^2 + n^3$

(d) $f(n) = n^2 + 3nm$ where $m < n$

Big-O Run-time Example 2 (solution)