CYDEO

# Data Structures and Algorithms Course

## Algorithm Analysis

# Data Structures & Algorithms Curriculum

- Algorithm Analysis
- Arrays
- Linked Lists
- Stacks
- Recursion
- Queues
- Hash Tables
- Sets
- Trees
- AVL Trees
- Heap
- Sorting Algorithms
- Searching Algorithms

# Today's Agenda

- What is a Data Structure / Algorithm?
- Algorithm Analysis & Big O notation
- Arrays

# What is a Data Structure

- Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.
- Depending on the organization of the elements, data structures are classified into two types:

  1) Linear data structures: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Linked Lists, Stacks and Queues.
  2) Non – linear data structures: Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.

# What is an Algorithm

- An algorithm is the step-by-step instructions to solve a given problem.

    1) Get the frying pan.
    2) Get the oil.
        a. Do we have oil?
            i. If yes, put it in the pan.
            ii. If no, do we want to buy oil?
                1. If yes, then go out and buy.
                2. If no, we can terminate.
    3) Turn on the stove, etc...

- For a given problem (preparing an omelette), we are providing a step-by- step procedure for solving it.

# How do we compare algorithms?

- We need performance measures.

-**Execution time?**

(Not a good measure, execution times are specific to a particular Computer)

-**Number of statements executed?**

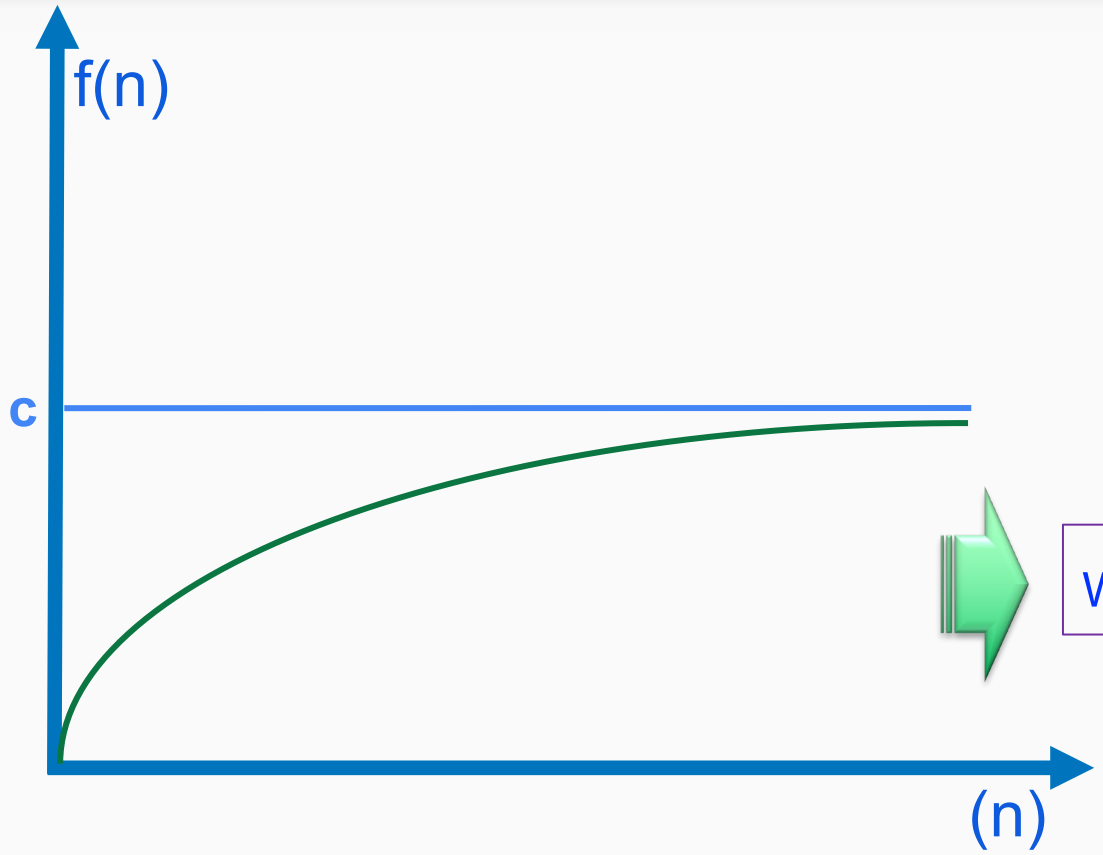
(Num of Statements varies with programming language)

-**Ideal solution :** Expressing running time of an algorithm as a function of input size.

***f(n)*** where the input size is ***n***.

-This type of comparison is independent of machine time, programming style, etc.

# Asymptotic Behavior



When “n” goes to ∞

- Always assume that we have a very large size data (n) when talking about the performance of algorithms.

# Asymptotic Behavior

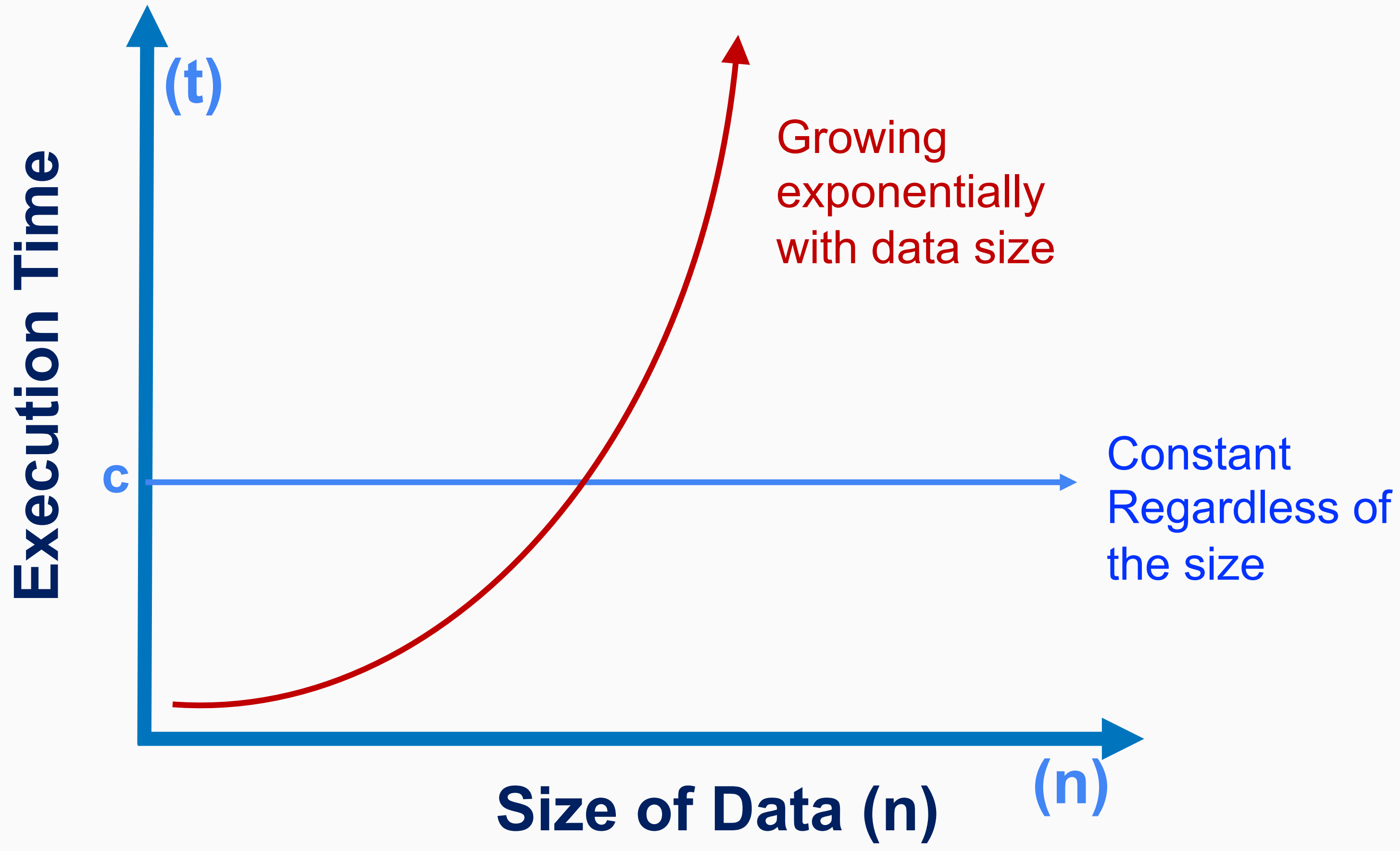
- In mathematical analysis, asymptotic analysis is a method of describing limiting behavior.

- If $f(n) = n^2 + 3n$, then as n becomes very large, the term 3n becomes insignificant compared to $n^2$. The function f(n) is said to be "asymptotically equivalent to $n^2$, as $n \to \infty$".

- If my input size is large, I can use asymptotic analysis.

# Rate of Growth

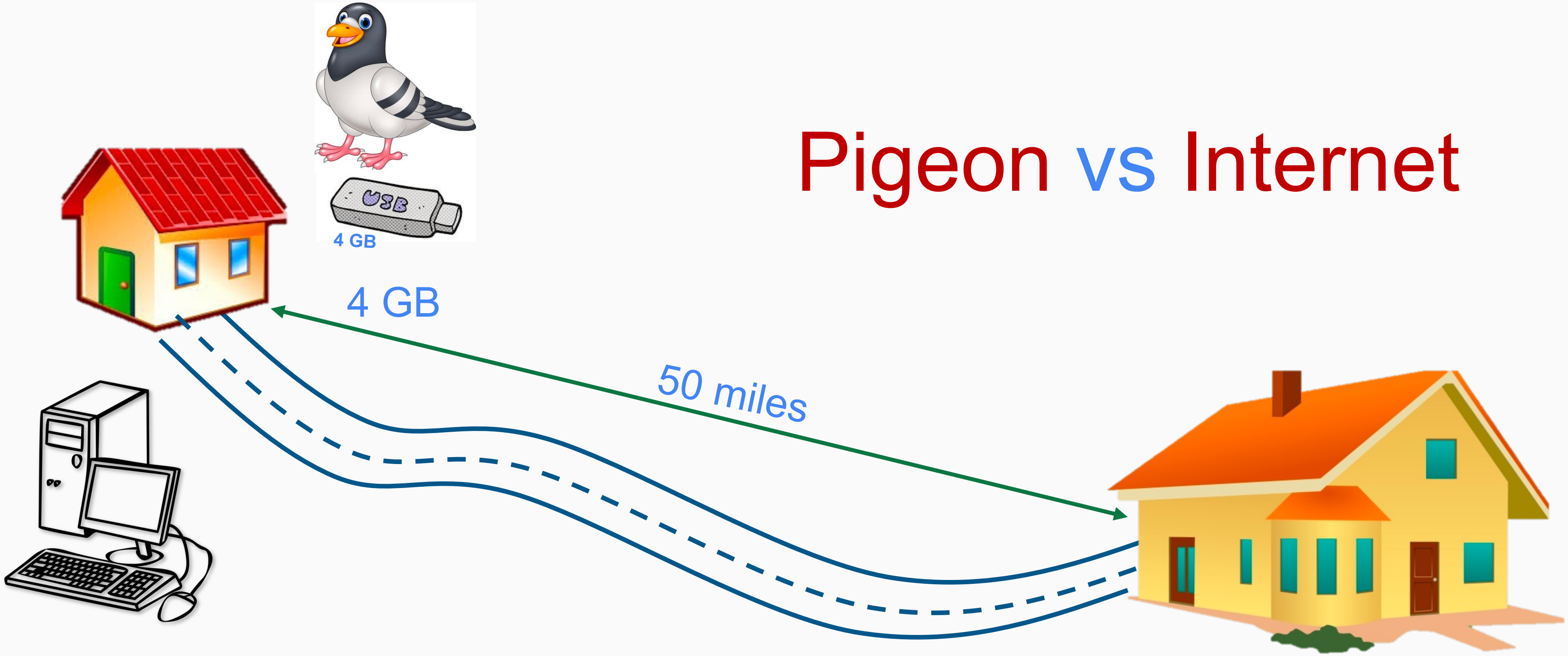
- The rate at which the **execution time** increases as a function of input_is called **rate of growth**.



CYDEO

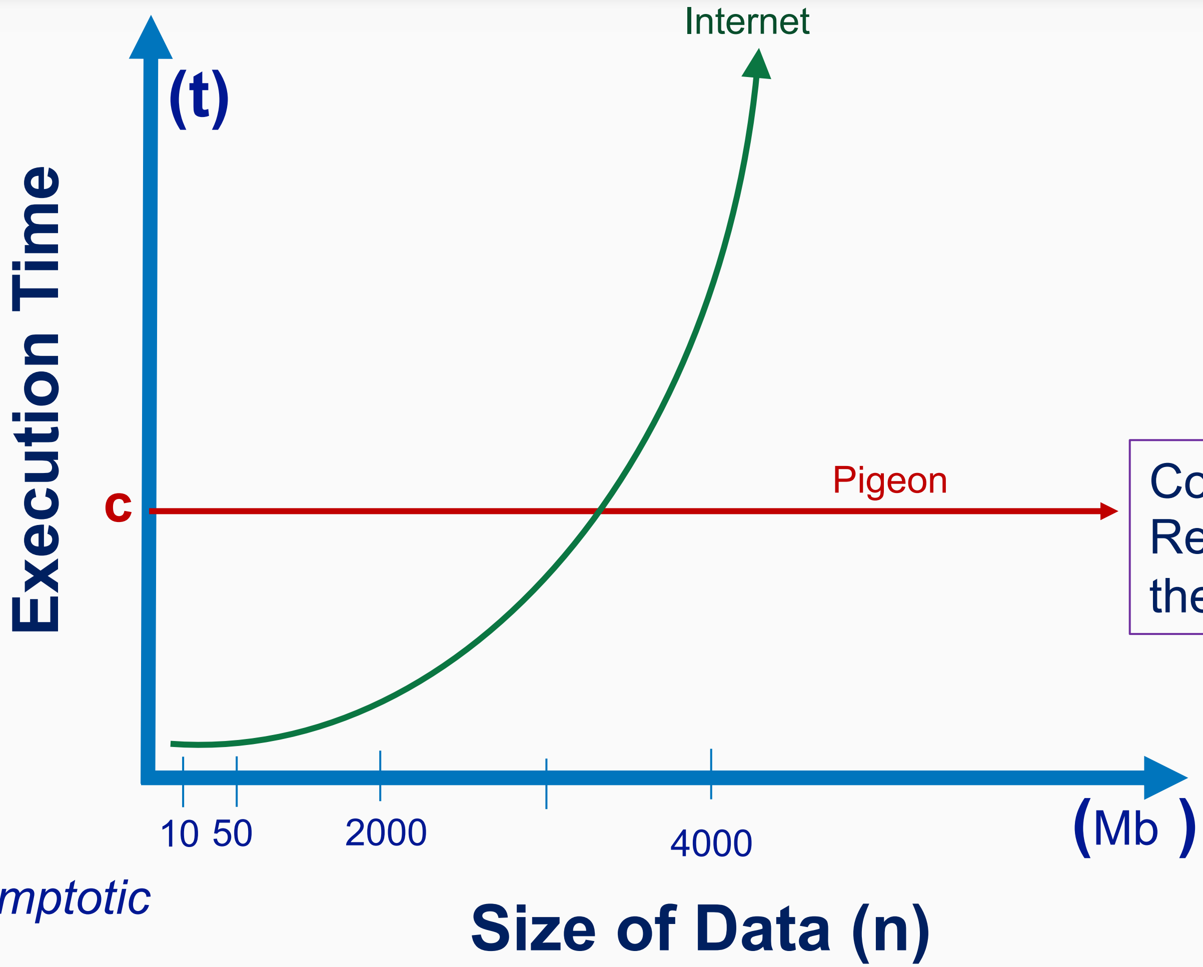
# Rate of Growth -Experiment

- In 2009, a company in South Africa had an issue: "really slow internet speed".
- The company had two offices located about *50 miles away from each other* and they decided to set up a fun test to see *if it would be faster to transfer data over their very much slow internet or via carrier pigeon*.



# Rate of Growth - Experiment

- Guess who’s winner



Constant Regardless of the size

*- If the data set is large, the asymptotic behavior becomes important.*
*- So we need a notation that deals with the asymptotic behavior.*

# Time and Space Complexity

In any piece of code; you deal with two types of complexities:

**1. Time complexity:** Number of steps taken by the algorithm, measured with respect to ***n*** *(input data to be processed)*, the size of the input.

**2. Space complexity:** The amount of space required by the algorithm to execute, measured with respect to *n (input data to be processed)*.

# Time Complexity- O(1)

***Constant | O(1) :***

Notice that in the previous scenario, the pigeon would take the same amount of time to carry 5KB, 10MB or 2TB of data stored in the USB drive. The pigeon will always take the same amount of time to move any amount of data from office A to office B.

***If the result of the operation requires constant time regardless of the data size that complexity is O(1)***

# What is Big O notation

- Big O notation, also called Landau's symbol, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions.
- Basically, it tells you how fast a function grows or declines.
  In other words:**Rate of Growth.**
- The ’Big-O’ is the language and metric we use to describe the efficiency of algorithms.
- Big-O is a Big Picture approach.

# What is Big O notation

- The ’Big-O’ notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

Rule 1: Always worst Case

Rule 2: Remove Constants

# What is Big O notation

- Rule 3 : If *f* (*n*) is a polynomial of degree *d*, that is,

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d \qquad a_d > 0, \text{ then } f(n) \text{ is } O(n^d).$$

$For\ example :$ $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is O($n^4$)

CYDEO

# Rate of Growth

$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle \qquad Total\ Cost \approx cost\_of\_car$$

$$\$35000 \quad + \$500$$

- From above example assume n is a large value of input size:

$$f(n) = n^4 + n^2 + 100n + 500 \approx n^4$$

Highest rate of growth

# Examples with Big O Notation

$5n^2 + 3n\log n + 2n + 5\ is$ $\boxed{O(n^2).}$

$20n^3 + 10n\log n + 5\ is$ $\boxed{O(n^3).}$

$3\log n + 2\ is$ $\boxed{O(\log n)}$

# How Do I Calculate the Complexity of Code Blocks

## What Can Cause Time in a Code Piece?

- Assigning a value to a variable
- Following an object reference
- Performing an arithmetic operation (+, -, *, / )
- Comparing two numbers
- Accessing a single element of an array by index
- Calling a method
- Returning from a method

Take a constant time “c”

# How Do I Calculate the Complexity of Code Blocks

**What Causes Space Complexity?**

Variables

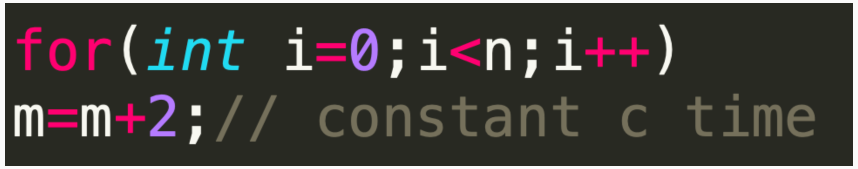
Data Structures (additional)

Method Call

Allocations

# Calculation of Big O

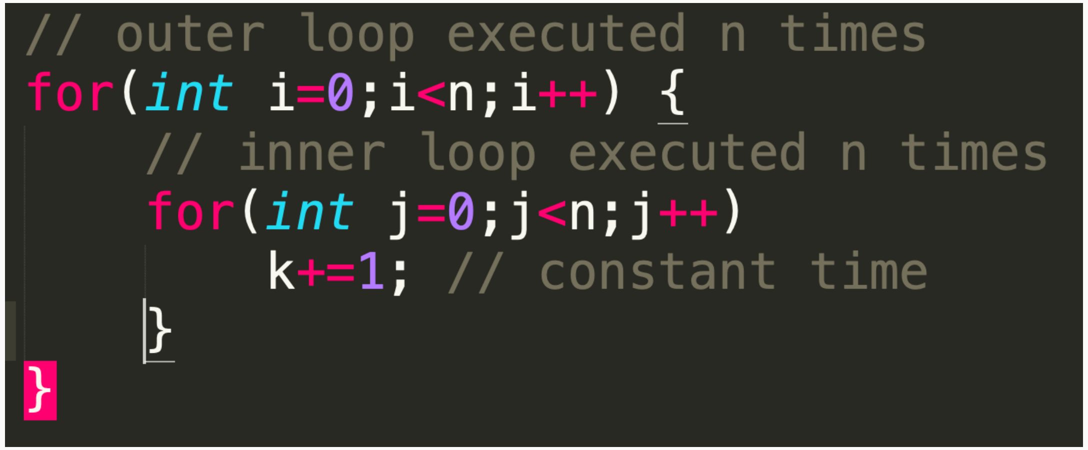
## Loops:



```
for(int i=0;i<n;i++)
m=m+2;// constant c time
```

Total time= constant c x n= $cn$ = O(n)

## Nested Loops:

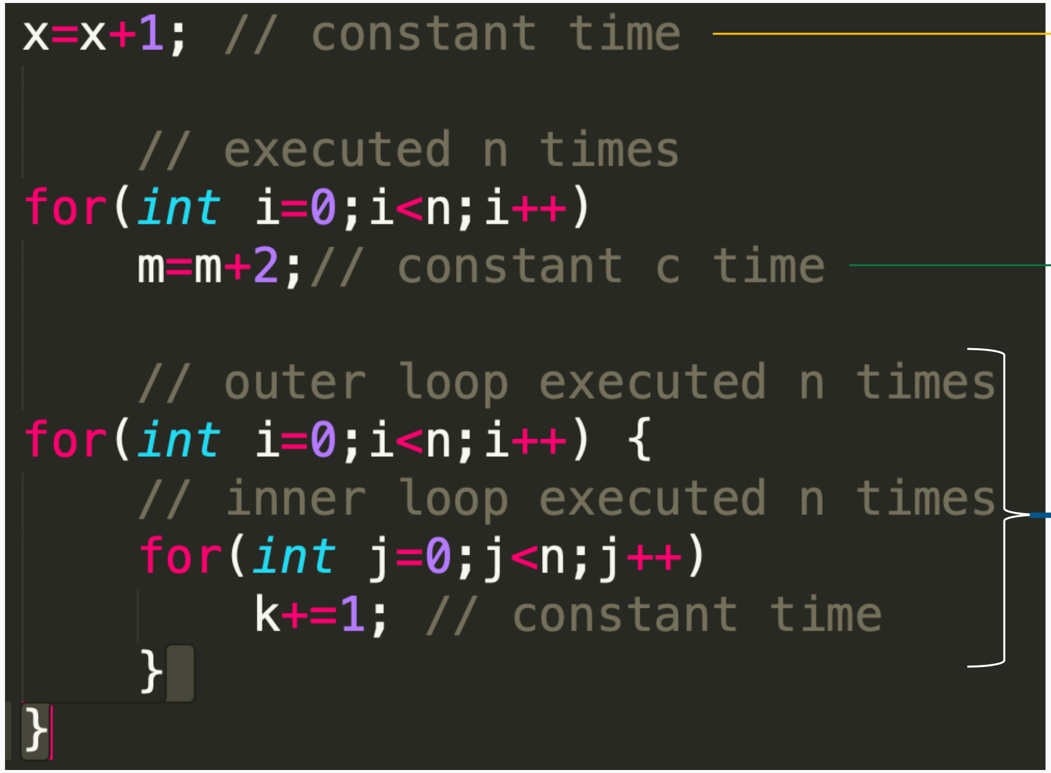


```
// outer loop executed n times
for(int i=0;i<n;i++) {
    // inner loop executed n times
    for(int j=0;j<n;j++)
        k+=1; // constant time
    }
}
```

Total time= c x n x n= $cn^2$ = O($n^2$ )

# Examples with Big O Notation (cont.)

## Consecutive Statements:



```
x=x+1; // constant time

    // executed n times
for(int i=0;i<n;i++)
    m=m+2;// constant c time

    // outer loop executed n times
for(int i=0;i<n;i++) {
    // inner loop executed n times
    for(int j=0;j<n;j++)
        k+=1; // constant time
    }
}
```

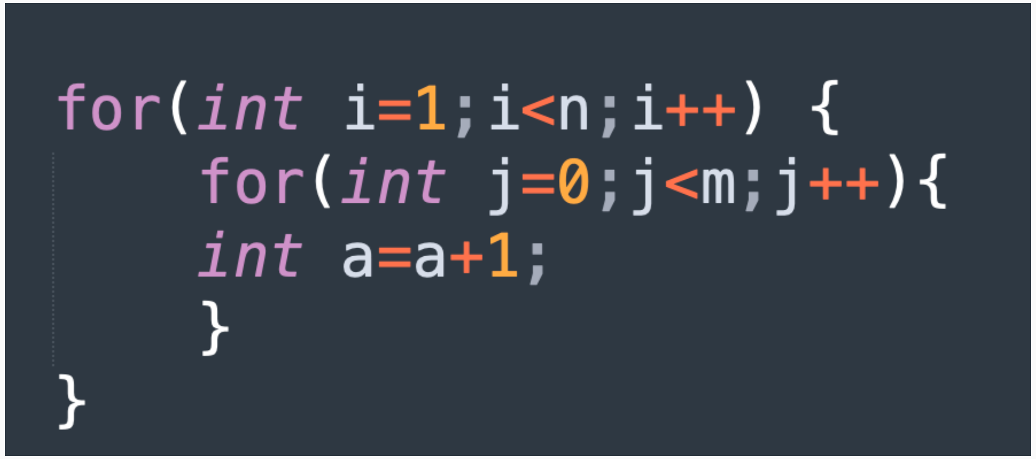
Total time=$c_0 + c_1 n + c_2 n^2$ = O($n^2$ )

# Calculation of Big O

## Rule 4 :

Different inputs should have different variables: **O(n + m) :** ’+’ for steps in order

A and B arrays nested would be: **O(n * m)** : ‘*’ for nested steps

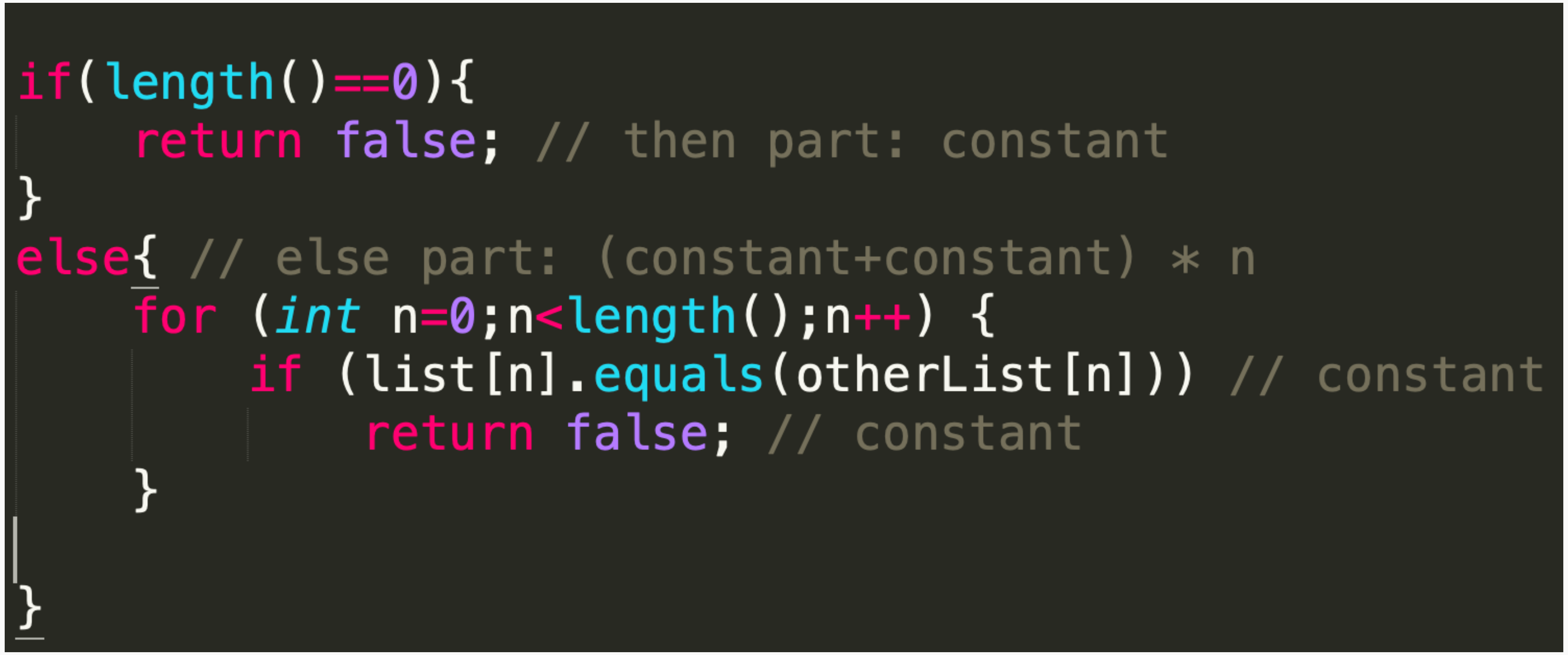


```
for(int i=1;i<n;i++) {
    for(int j=0;j<m;j++){
    int a=a+1;
    }
}
```

= c x n x m = O(n x m)

# Examples with Big O Notation (cont.)

## If then Statements:



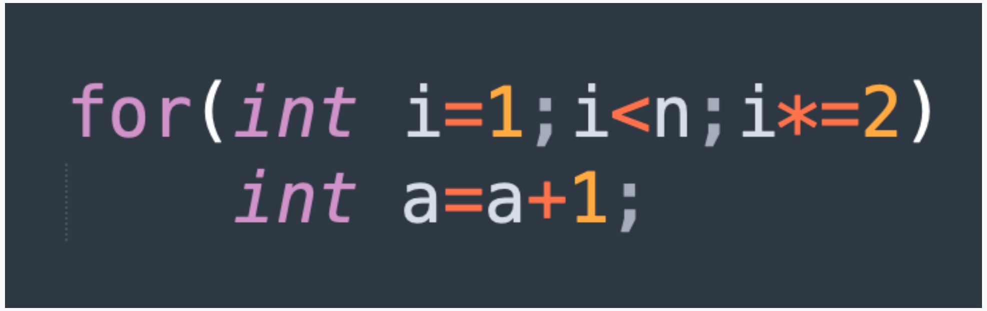
```
if(length()==0){
    return false; // then part: constant
}
else{ // else part: (constant+constant) * n
    for (int n=0;n<length();n++) {
        if (list[n].equals(otherList[n])) // constant
            return false; // constant
    }

}
```

Total $= c_0 + (c_1 + c_2) length$ = O(length)

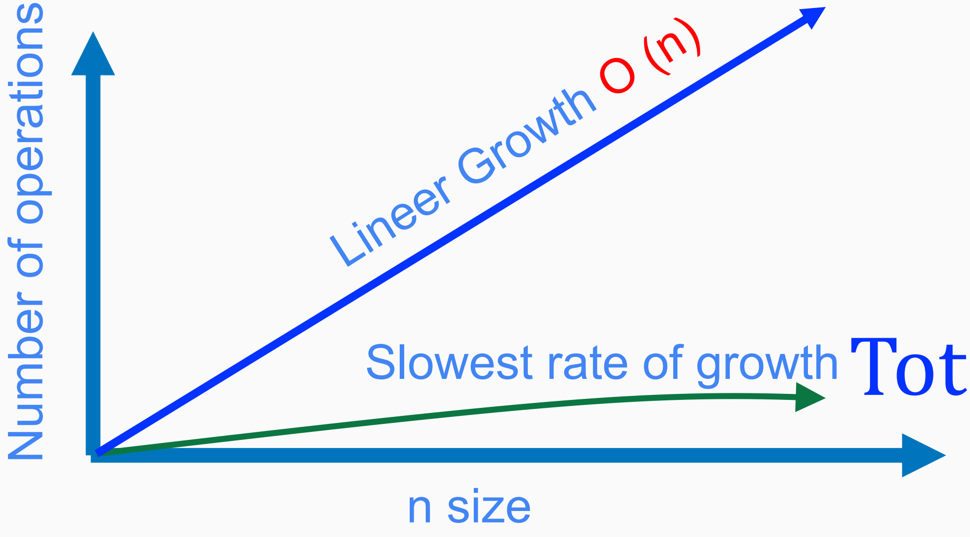
# Examples with Big O Notation (cont.)

## Logarithmic complexity:



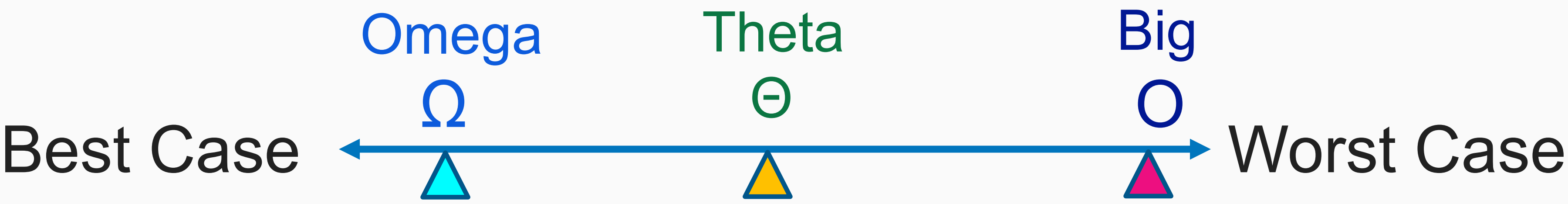
```
for(int i=1;i<n;i*=2)
    int a=a+1;
```

The value of i is doubling every time. Initially i = 1, in next step i = 2, and in subsequent steps i = 4, 8 and so on.

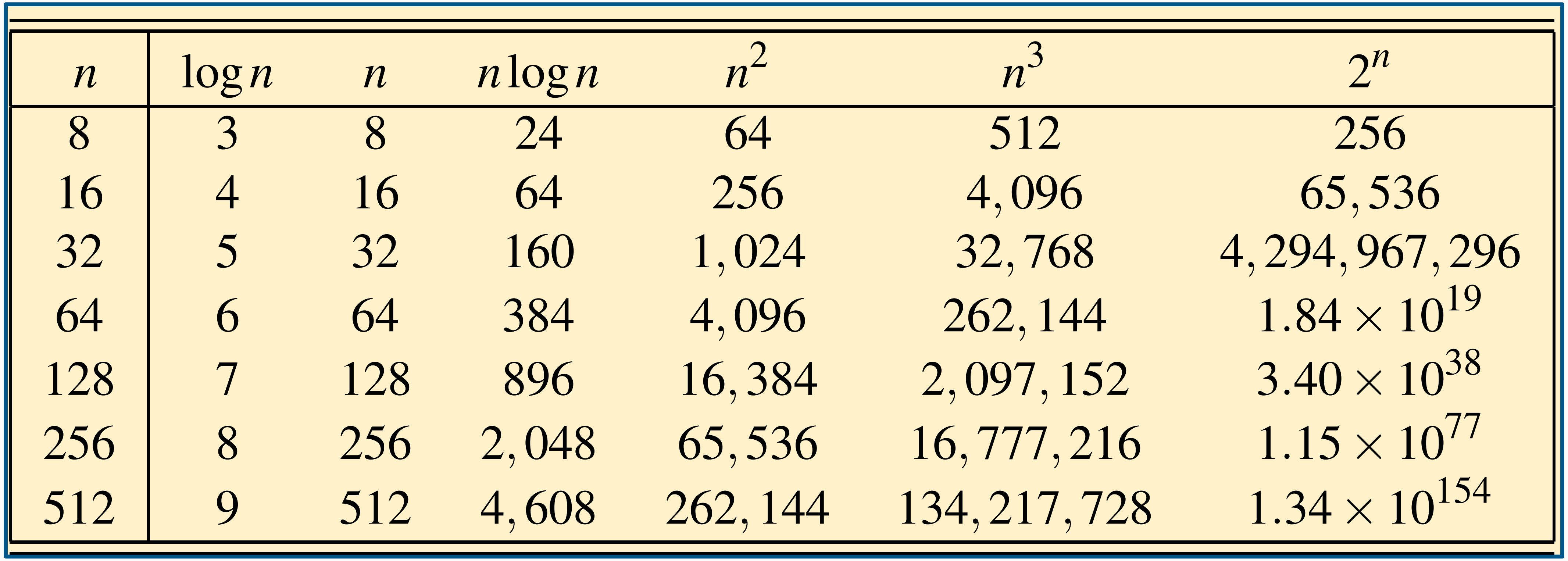


$$\text{Total time} = O(\log n)$$

# Other Notations



# Big O Notation Complexity Values



| $n$ | $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 8 | 3 | 8 | 24 | 64 | 512 | 256 |
| 16 | 4 | 16 | 64 | 256 | 4,096 | 65,536 |
| 32 | 5 | 32 | 160 | 1,024 | 32,768 | 4,294,967,296 |
| 64 | 6 | 64 | 384 | 4,096 | 262,144 | $1.84 \times 10^{19}$ |
| 128 | 7 | 128 | 896 | 16,384 | 2,097,152 | $3.40 \times 10^{38}$ |
| 256 | 8 | 256 | 2,048 | 65,536 | 16,777,216 | $1.15 \times 10^{77}$ |
| 512 | 9 | 512 | 4,608 | 262,144 | 134,217,728 | $1.34 \times 10^{154}$ |

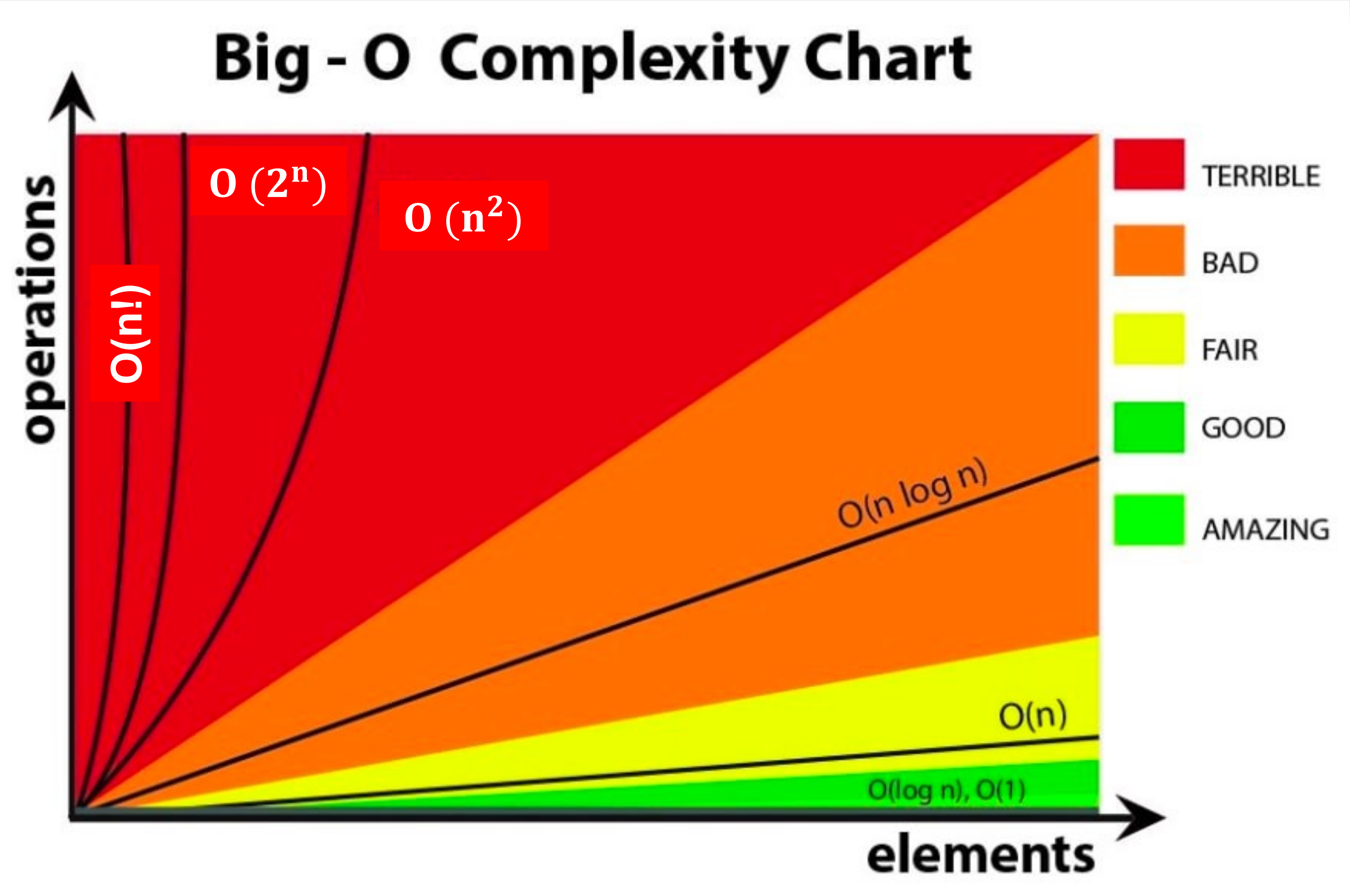
Number of operations/iterations

# Big – O Complexity Chart

Best

| Notation | Name |
|---|---|
| O(1) | Constant |
| O(log(n)) | Logarithmic |
| $O((\log(n))^c)$ | Polylogarithmic |
| O(n) | Linear |
| $O(n^2)$ | Quadratic |
| $O(n^c)$ | Polynomial |
| $O(c^n)$ | Exponential |

Worst

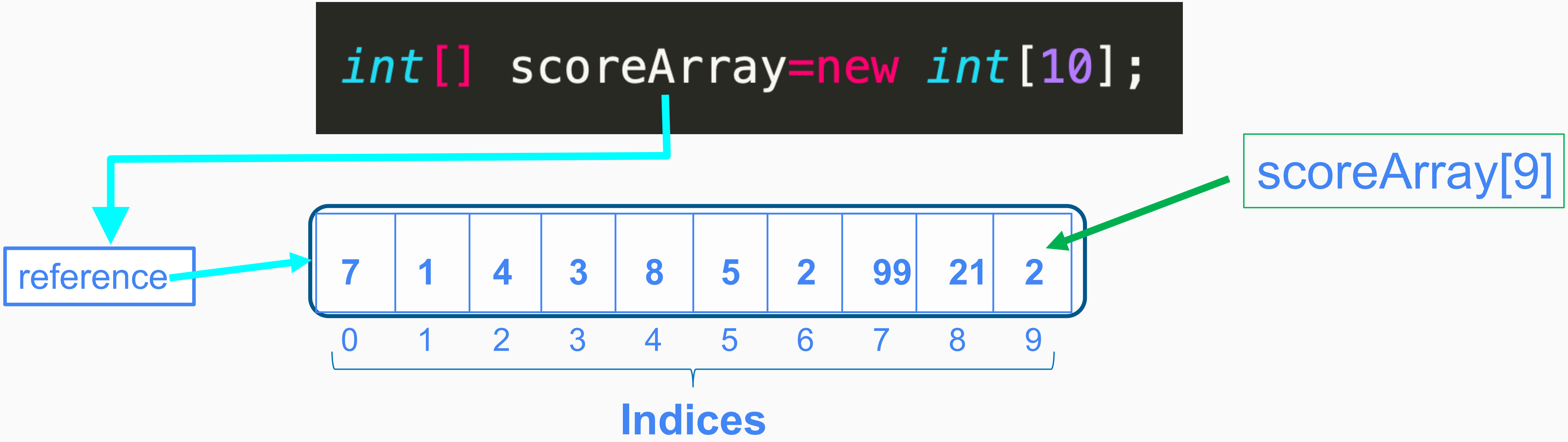


# What you learn

- What is a Data Structure / Algorithm ✓
- Big O notation ✓
- Arrays

# Arrays

- An *array* is the basic mechanism for storing a collection of identically typed entities.
- Arrays use static memory allocation.

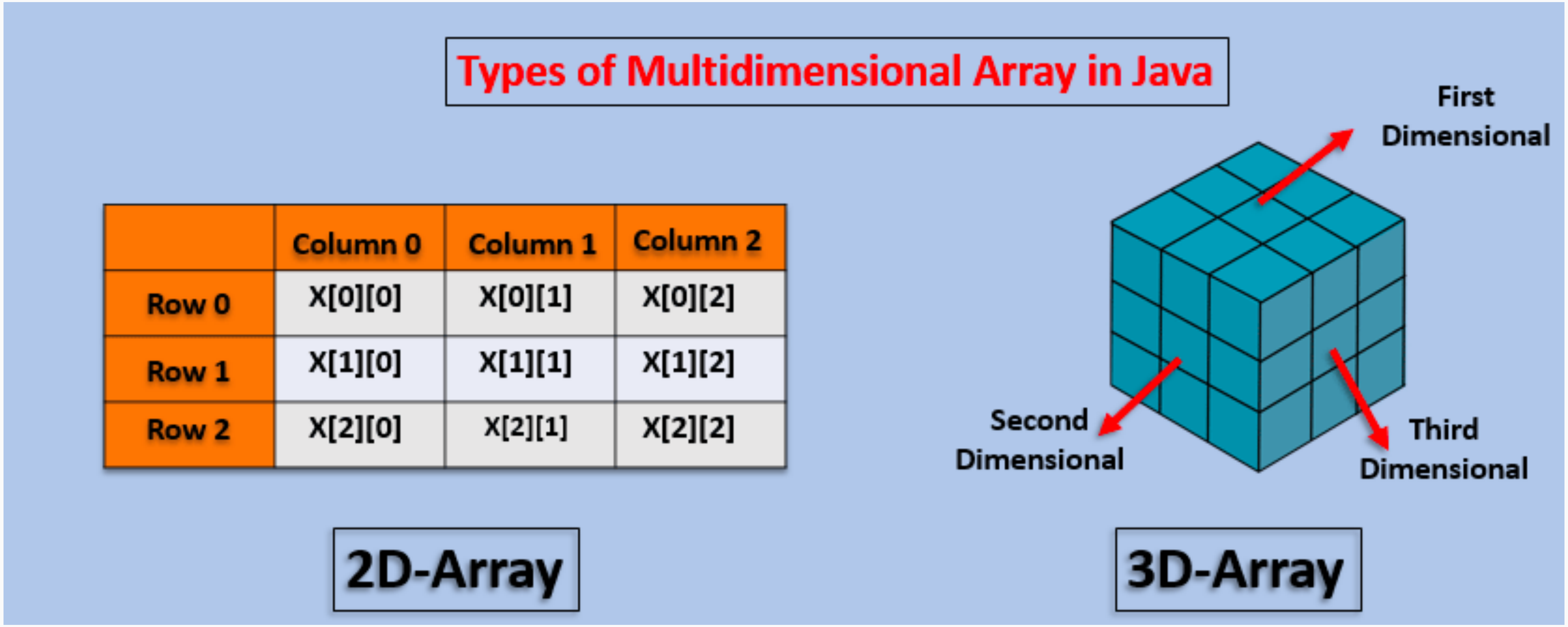


```java
int[] scoreArray=new int[10];
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | 4 | 3 | 8 | 5 | 2 | 99 | 21 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Arrays

- Arrays can also be multi-dimensional



Types of Multidimensional Array in Java

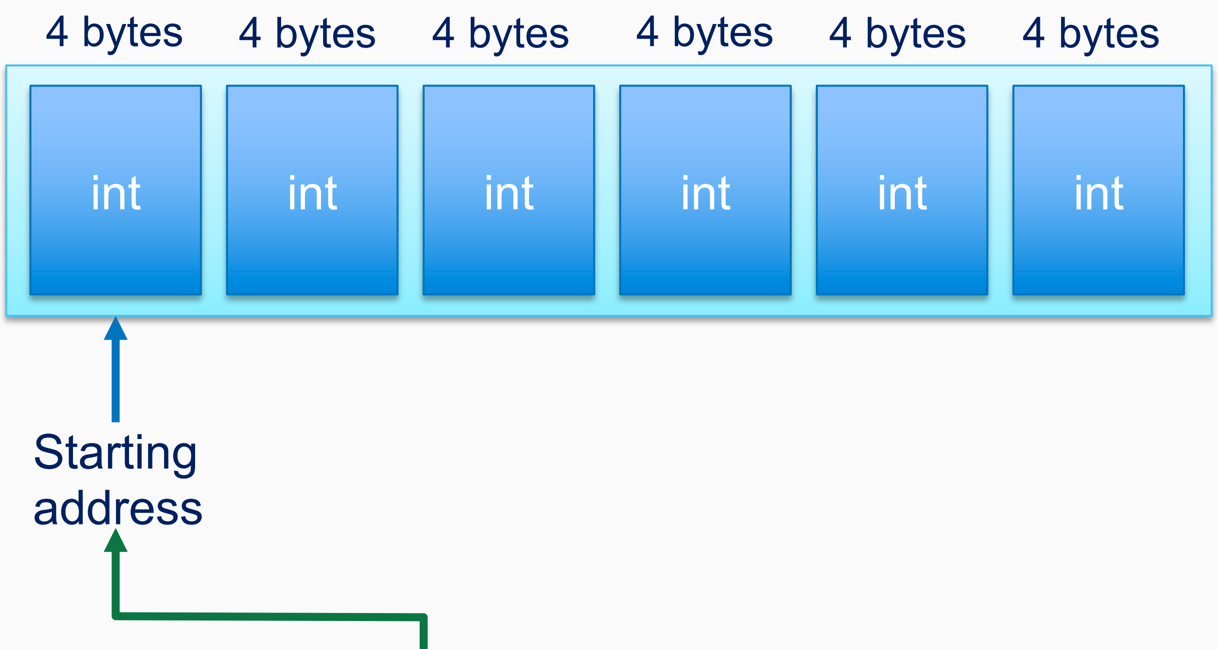
| | Column 0 | Column 1 | Column 2 |
|---|---|---|---|
| Row 0 | X[0][0] | X[0][1] | X[0][2] |
| Row 1 | X[1][0] | X[1][1] | X[1][2] |
| Row 2 | X[2][0] | X[2][1] | X[2][2] |

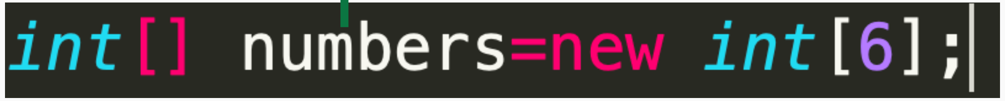
2D-Array

3D-Array

# Arrays

How are arrays kept in memory?

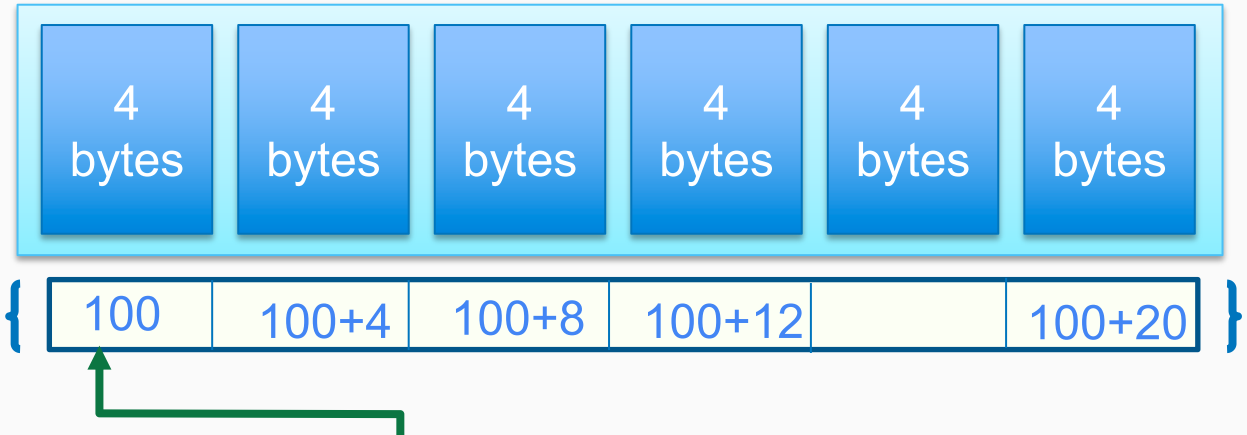




```
int[] numbers=new int[6];
```

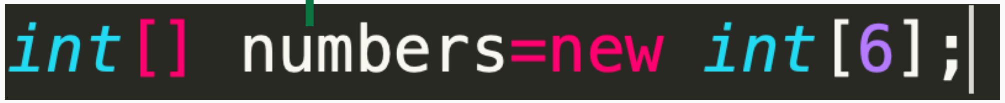
# Arrays

How are arrays kept in memory?



| | | | | | |
|---|---|---|---|---|---|
| 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes |
| 100 | 100+4 | 100+8 | 100+12 | | 100+20 |

memory addresses



```
int[] numbers=new int[6];
```

**Formula :**

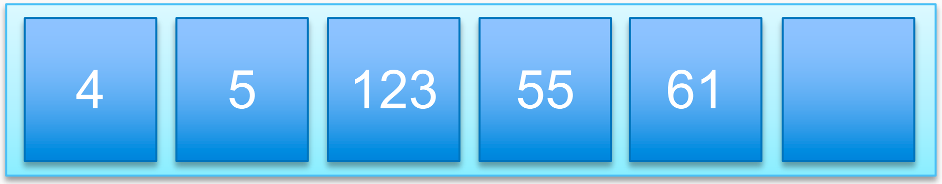
Address of Index i= numbers address + i * (memory allocated for that type)

*integer = 4 bytes*

numbers[4]. = 100+ (4 * 4) = 116

CYDEO

# Big O Analysis of Arrays



| | | | |
|---|---|---|---|
| Accessing an element of array | : | O(1) | |
| Insert an element to end | : | O(1) | |
| Insert an element to beginning | : | O(n) | Worst Case |
| Delete an element from beginning | : | O(n) | Worst Case |
| Delete an element from end | : | O(1) | |

# Arrays

- **Advantages:**
  - Simple and easy to use
  - Faster access to the elements (constant access)

- **Disadvantages:**

  -Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.

  -Fixed size: The size of the array is static (specify the array size before using it).

  -One block allocation: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).

  -Complex position-based insertion: To insert an element at a given position, we may need to shift the existing elements.

# Dynamic Arrays

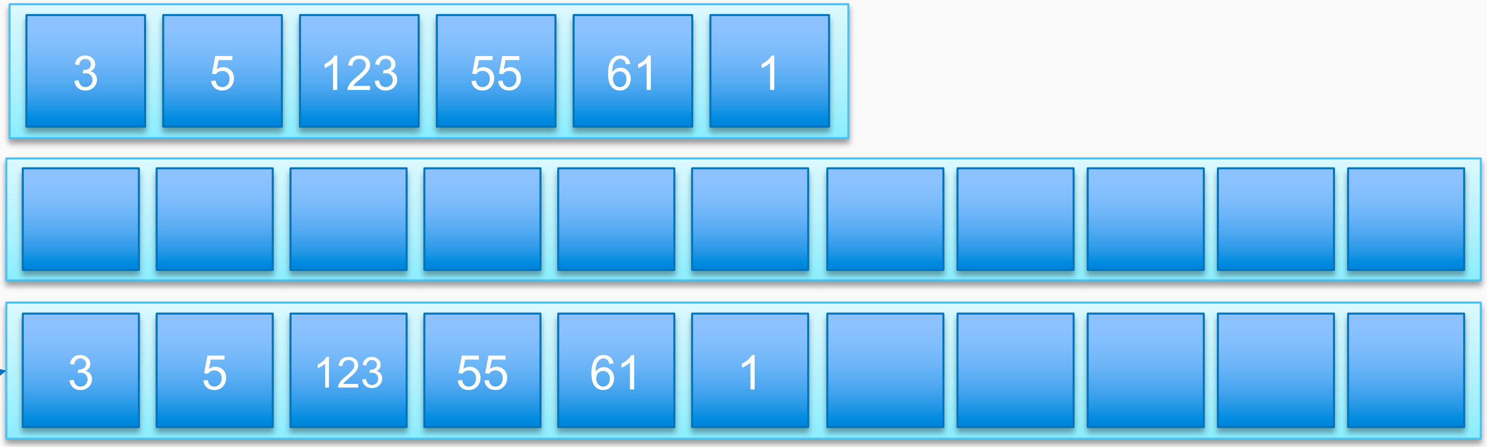
- Dynamic arrays are resizable, growable arrays.
- How can we create dynamic arrays?

1. Assume  array ‘numbers’ is full.

2. Create an empty NewArrray.

3. Copy all to NewArrray.

4. numbers= NewArrray.



5. Old memory blocks of numbers array are eligible for garbage collection by JVM.

# Dynamic Arrays

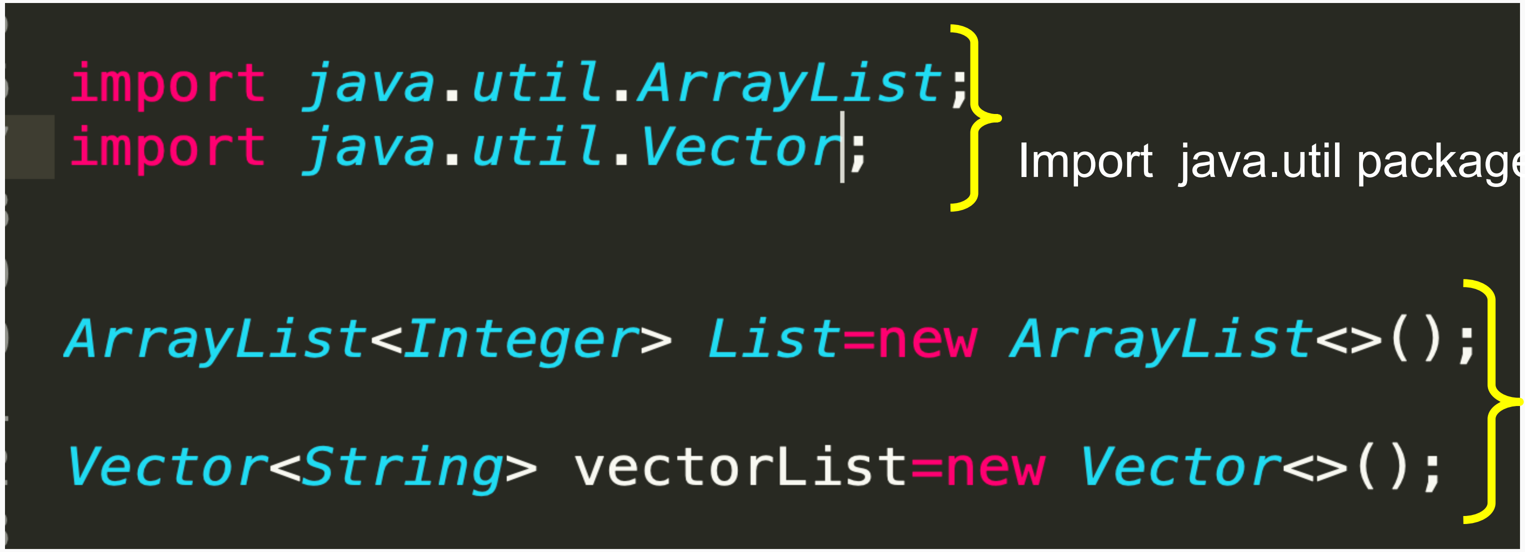
- There two implementations of Dynamic Arrays in Java:
  - Vector class : Size is increased by %100 if full. Synchronized (Only a single thread can access in multi-threaded environment )
  - ArrayList class: Size is increased by % 50 if full. If you need multithreads access to data you should prefer ArrayList class.

- **Performance :**

| Operation | Big O |
|---|---|
| Indexing | O(1) |
| Insert/delete beginning | O(n) |
| Insert/delete at end | O(1) |
| Insert/delete in middle | O(n) |

# Implementation of Dynamic Arrays in Java

- Vector and ArrayList are under java.util package
- They can hold objects so, for primitive types use wrapper class.



```
import java.util.ArrayList;
import java.util.Vector;


ArrayList<Integer> List=new ArrayList<>();

Vector<String> vectorList=new Vector<>();
```

Import  java.util package

declarations

# ArrayList and Vector Performance

- ***add()*** – *O(1)*
- ***add(index, element)*** – *O(n)*
- ***get()*** – *O(1)*
- ***remove()*** – *O(n)*
- ***indexOf()*** – *O(n)*
- ***contains()*** – implementation is based on *indexOf()*. So it will also run in *O(n)* time.

# Arrays - Key Takeaways

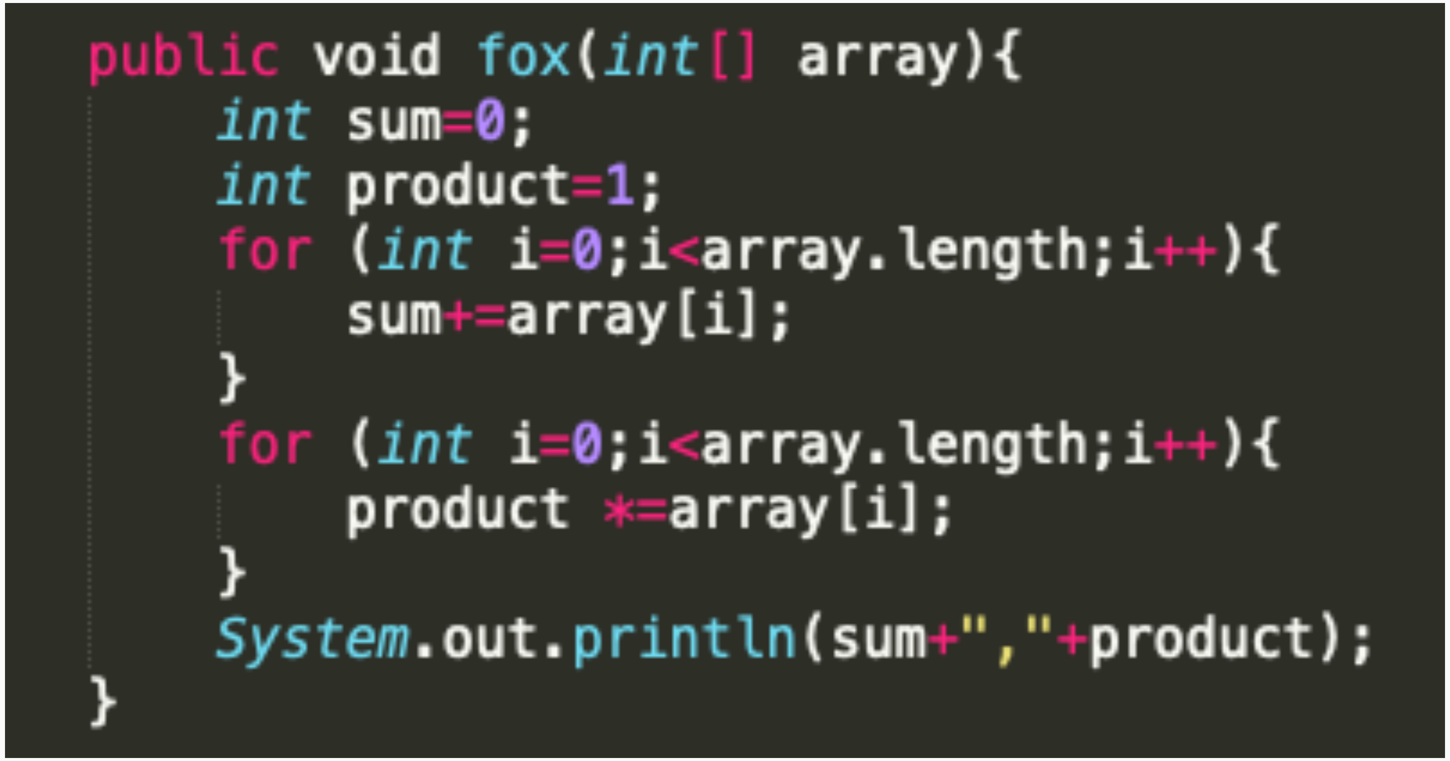
- Simplest Data Structure
- Static (Arrays) vs dynamic (ArrayList & Vector)
- Arrays are great if you know how many items you’ll store.

## Runtime Complexities

| | |
|---|---|
| Lookup by Index | O(1) |
| Lookup by Value | O(n) |
| Insert | O(n) |
| Delete | O(n) |

# Big O Exercises

1. What is the runtime complexity of the below code?

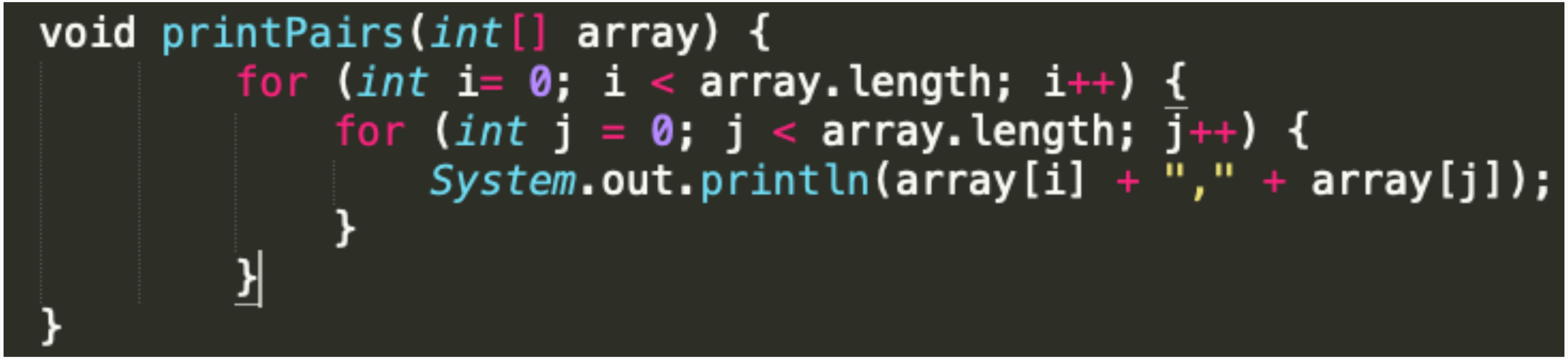


```java
public void fox(int[] array){
    int sum=0;
    int product=1;
    for (int i=0;i<array.length;i++){
        sum+=array[i];
    }
    for (int i=0;i<array.length;i++){
        product *=array[i];
    }
    System.out.println(sum+","+product);
}
```

a. O($n^2$)    b. O(log n)    c. O(nlogn)    d. O(n)    e. O(1)

# Big O Exercises

2. What is the runtime complexity of the below code?

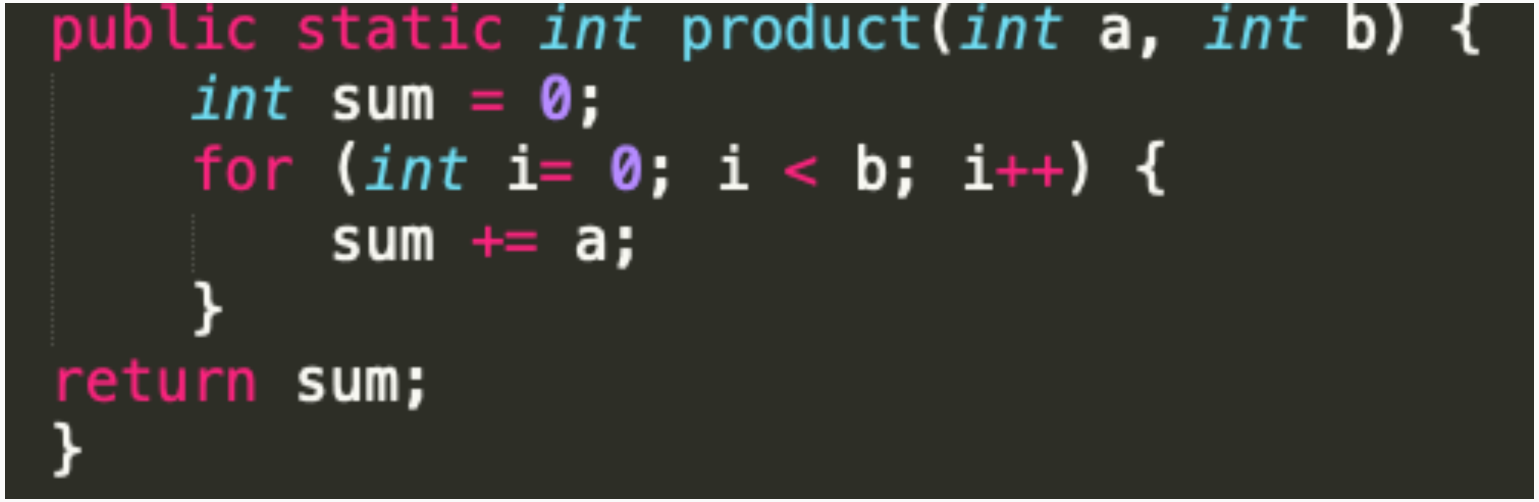


```java
void printPairs(int[] array) {
    for (int i= 0; i < array.length; i++) {
      for (int j = 0; j < array.length; j++) {
        System.out.println(array[i] + "," + array[j]);
      }
    }
}
```

a. O($n^2$) b. O(log n) c. O(nlogn) d. O(n) e. O(1)

# Big O Exercises

3. What is the runtime complexity of the following code block?

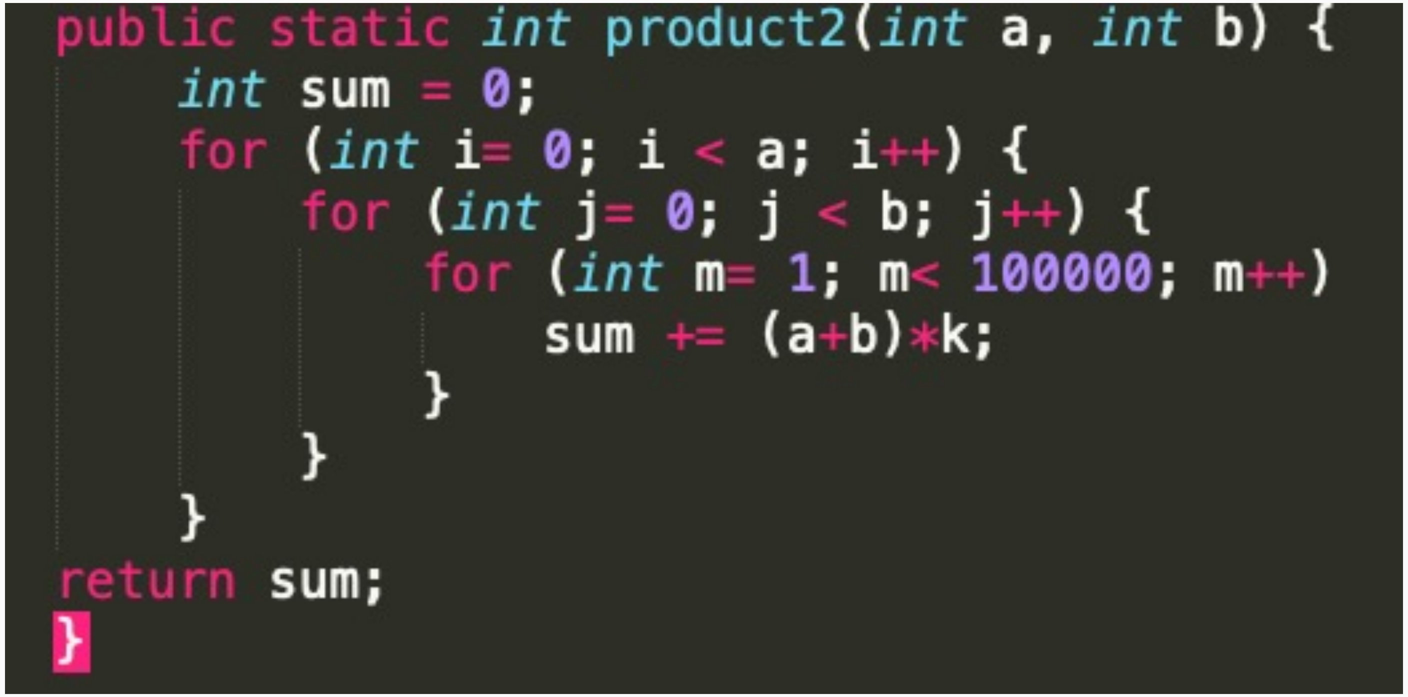


```java
public static int product(int a, int b) {
    int sum = 0;
    for (int i= 0; i < b; i++) {
        sum += a;
    }
return sum;
}
```

a. O($a^2$)     b. O(a*b)     c. O(b)     d. O(log n)     e. O(a+b)

# Big O Exercises

4. The following code computes the product of a and b. What is its runtime complexity?



```
public static int product2(int a, int b) {
    int sum = 0;
    for (int i= 0; i < a; i++) {
        for (int j= 0; j < b; j++) {
            for (int m= 1; m< 100000; m++)
                sum += (a+b)*k;
            }
        }
    }
return sum;
}
```

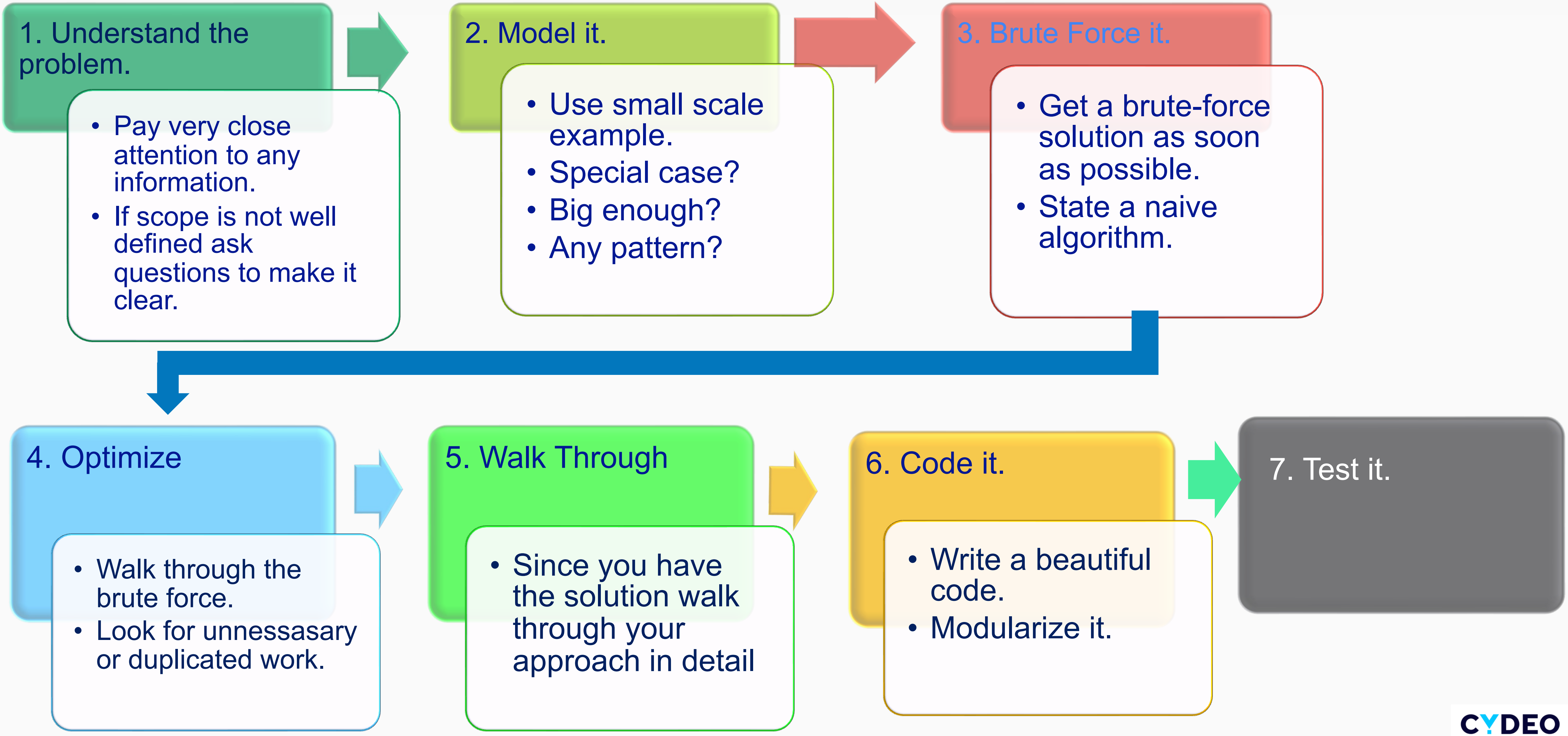
a. O($k^2$)    b. O(a*b)    c. O(k*b)    d. O(k * log n)    e. O(a+b)

CYDEO

# Arrays Algo Question

## Two Sum Problem

# Problem Solving Pattern



1. Understand the problem.

- Pay very close attention to any information.
- If scope is not well defined ask questions to make it clear.

2. Model it.

- Use small scale example.
- Special case?
- Big enough?
- Any pattern?

3. Brute Force it.

- Get a brute-force solution as soon as possible.
- State a naive algorithm.

4. Optimize

- Walk through the brute force.
- Look for unnessasary or duplicated work.

5. Walk Through

- Since you have the solution walk through your approach in detail

6. Code it.

- Write a beautiful code.
- Modularize it.

7. Test it.

# Problem Definition

**Problem: Two Sum**

- Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.
- You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.
- You can return the answer in any order.
- No solution will return empty array.

**Example:**

**Input:** nums = [2, 7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

# 1. Understand the Problem

Ask questions to interviewers !

Is the array sorted?

- No

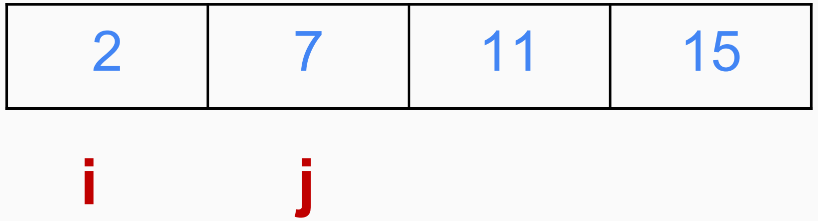
- No duplicated values accepted

# 2. Model It

Looking for:

*Such* array[i]+ array[j] = target value;

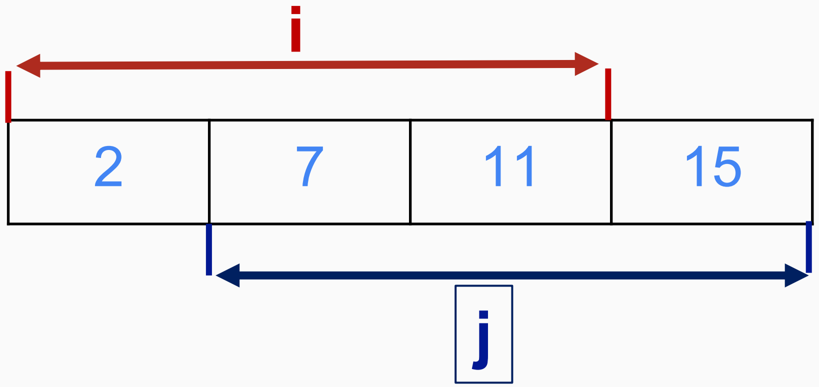
Target value= 9



|   |   |    |    |
|---|---|----|----|
| 2 | 7 | 11 | 15 |
| i | j |    |    |

# 3. Find a Brute Force Solution

*If* (array[i]+ array[j] == target value) return i & j;



# 3. Find a Brute Force Solution

**Pseudo Code:**

```
for(int i=0;i<length();i++) begin
        for(int j=0;j<length();j++) begin
                if (array[i] + array[j]==targetValue) return i & j
        end
end
```

# Assignment-1

1. For the Two Sum problem, turn the pseudo code into a brute-force solution.
2. Find the time complexity of your solution.
3. Are there any other cases we need to handle?

# Assignment-2 (Number of People in the Bus)

**Link for the problem:** https://www.codewars.com/kata/5648b12ce68d9daa6b000099/train/java

**1.** There is a bus moving in the city, and it takes and drop some people in each bus stop.
You are provided with a list (or array) of integer pairs. Elements of each pair represent number of people get into bus (The first item) and number of people get off the bus (The second item) in a bus stop.

Your task is to return number of people who are still in the bus after the last bus station (after the last array). Even though it is the last bus stop, the bus is not empty and some people are still in the bus, and they are probably sleeping there :D
Take a look on the test cases next page.

Please keep in mind that the test cases ensure that the number of people in the bus is always >= 0. So the return integer can't be negative.
The second value in the first integer array is 0, since the bus is empty in the first bus stop.

**2.** What is the time complexity of your solution?

# Assignment-2 (Number of People in the Bus)

**Test Case:**

```
public void test1() {
    ArrayList<int[]> list = new ArrayList<int[]>();
    list.add(new int[] {10,0});
    list.add(new int[] {3,5});
    list.add(new int[] {2,5});
    assertEquals(5, metro.countPassengers(list));
    }
```

**Code Template:**

```
import java.util.ArrayList;

class Metro {

  public static int countPassengers(ArrayList<int[]> stops) {
  //Code here!
  }
}
```

# Big O Cheat Sheet

**Big Os :**

**O(1)** Constant – no loops

**O(log N)** Logarithmic – usually searching algorithms have log n if they are sorted (Binary Search)

**O(n)** Linear – for loops, while loops through n items

**O(n log(n))** Log Linear – usually sorting operations

**O(n^2)** Quadratic – every element in a collection needs to be compared to ever other element. Two nested loops

**O(2^n)** Exponential – recursive algorithms that solves a problem of size N

**O(n!)** Factorial – you are adding a loop for every element

**Iterating through half a collection is still O(n)**

**Two separate collections: O(a * b)**

**Rule Book**

**Rule 1:** Always worst Case

**Rule 2:** Remove Constants ($c1+c2+ 2n^2$)

**Rule 3:** Different inputs should have different variables: **O(a + b)**

A and B arrays nested would be: **O(a * b)**

**+ for steps in order**

*** for nested steps**

**Rule 4:** Drop Non-dominant terms

**What Causes Space Complexity?**

Variables
Data Structures
Function Call
Allocations

**What Can Cause Time in a Code?**

Operations (+, -, *, /)
Comparisons (<, >, ==)
Looping (for, while)
Outside Function call (function())