# JAVA DSA

QUEUE, ARRAY, HASH MAP

QUICK SORT, MERGE SORT

# What is Data?

In computing, data is information that has been translated into a form that is efficient for movement or processing. Relative to today's computers and transmission media, data is information converted into binary digital form.

Ex:- One person information is also data

Name - Venkat

Profession - Trainer

Experience - 18 Years

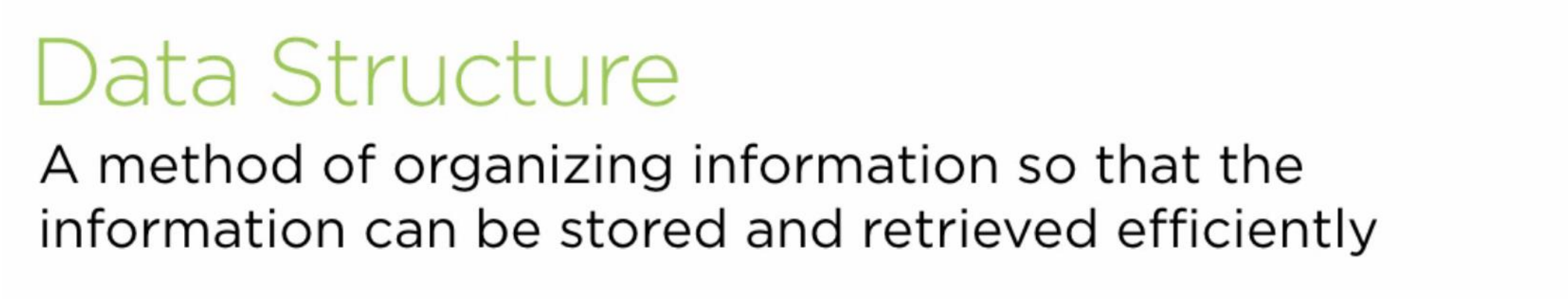
**Can you define the data for the given Diagram?**

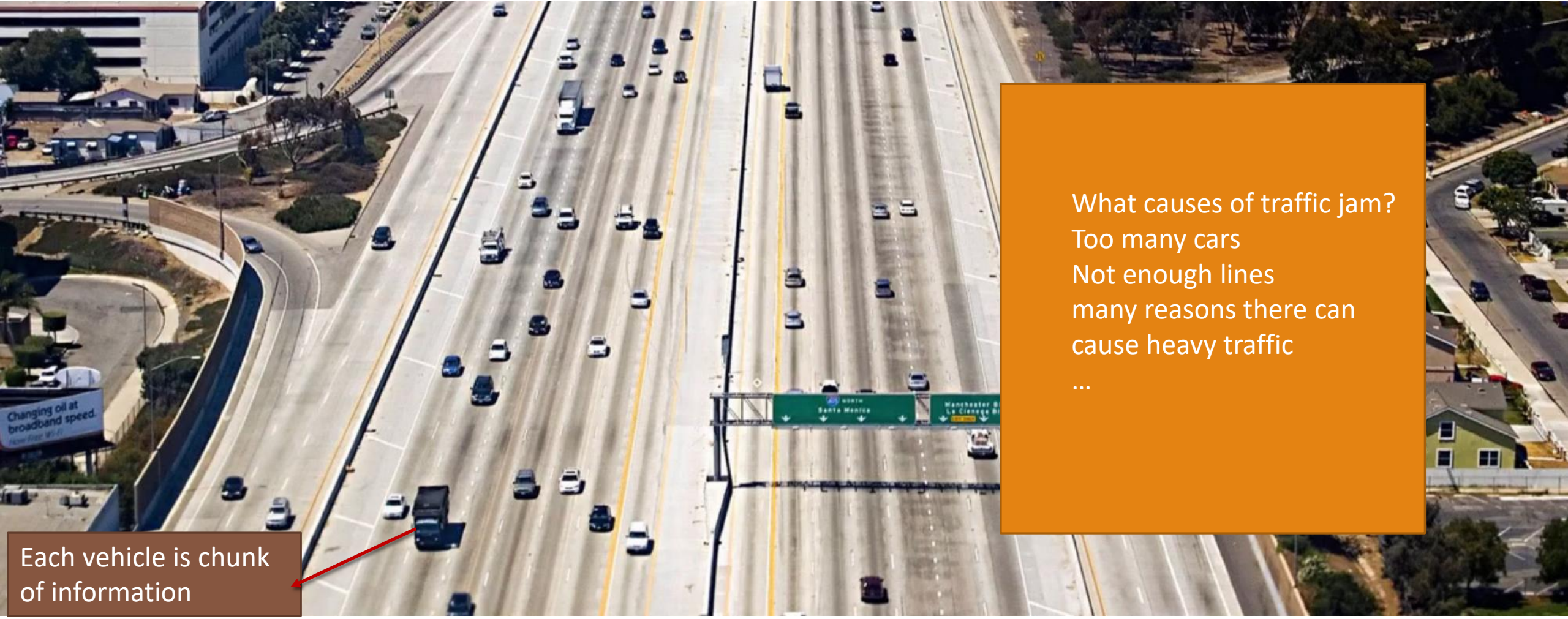

SAMSUNG Galaxy Z Fold3 5G (Phantom Black, 512 GB)

4.1★ 89 Ratings & 4 Reviews

- 12 GB RAM | 512 GB ROM
- 19.3 cm (7.6 inch) QXGA+ Display
- 12MP + 12MP + 12MP | 10MP Front Camera
- 4400 mAh Lithium-ion Battery
- Qualcomm Snapdragon 888 Octa-Core Processor
- 1 Year Manufacturer Warranty for Device and 6 months Manufacturer Warranty for In-Box Accessories

# Data Structures?



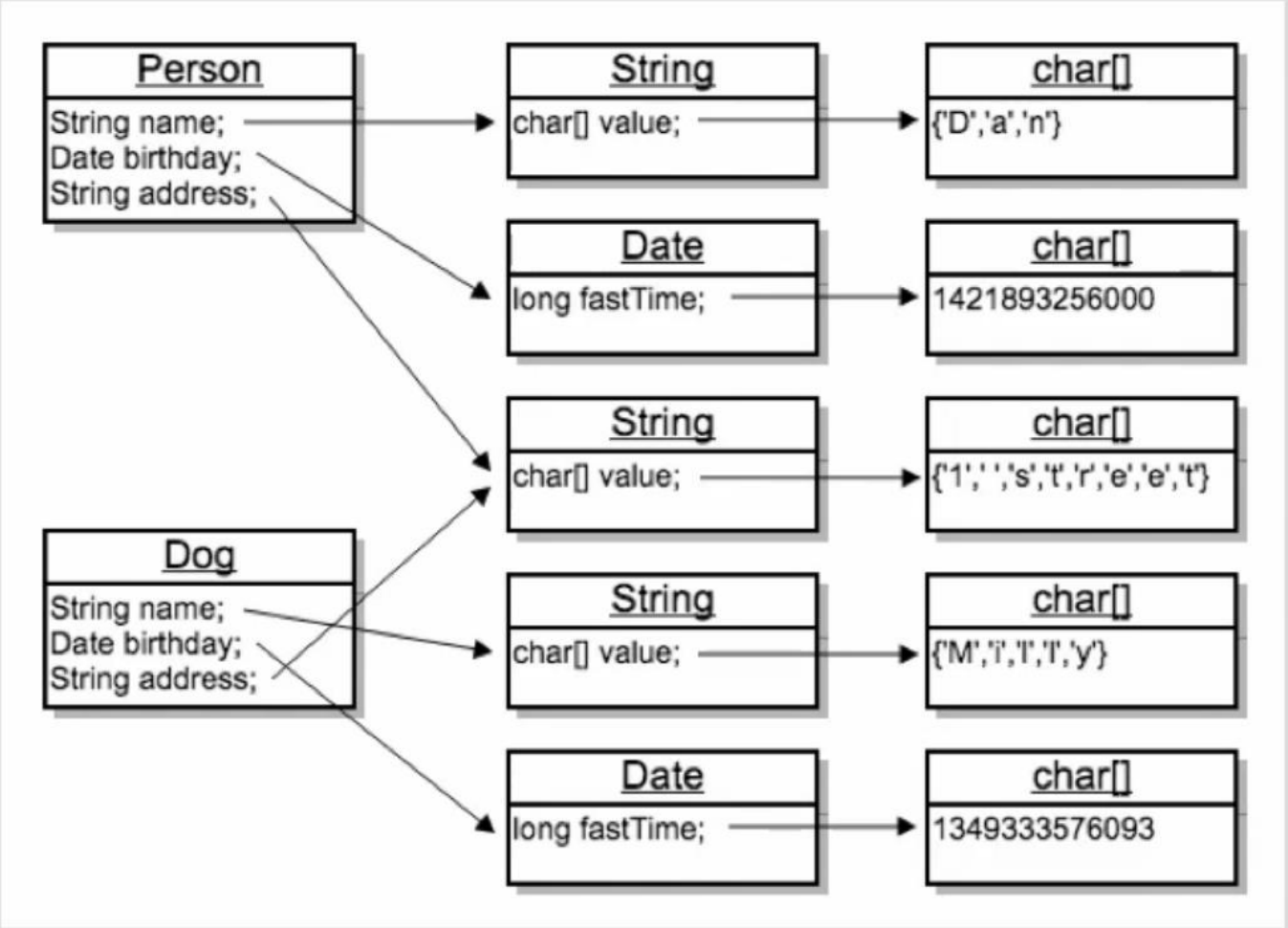



What causes of traffic jam?
Too many cars
Not enough lines
many reasons there can cause heavy traffic
…

Each vehicle is chunk of information

**“Like wise in your code well designed data structure can make you data flow nicely and keep program performance top notch”**

```
String s = "a string";
char value[] = {'a', ' ', 's', 't', 'r', 'i', 'n', 'g'};
```



# Algorithms?

An Algorithm is a set of instructions to perform a task or to solve a given problem

**For Example –**
Are recipe book is a collection of recipes in which each recipe provides an step by step instruction to prepare food

Lets say you want to prepare a tea. So, the steps would be -

1. Boil water
2. Add tea powder and milk to boiled water
3. Put tea into tea pot
4. Put hot tea into tea cups
5. Do you need sugar?
    a. If yes, put it into tea cups
    b. If no, do nothing
6. Stir, drink and enjoy !!

# Print average of 3 given numbers

Lets say you want to write algorithm for it , so the steps would be -

1. Perform sum of 3 numbers
2. Store it in a variable sum
3. Divide the sum by 3
4. Store the value in variable avg
5. Print the value stored in avg

```java
public void findAvg(int a, int b, int c) {
    int sum = a + b + c;
    int avg = sum / 3;
    System.out.println(avg);
}
```

# What is Big O Notation?

## Big O

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

— Wikipedia

- Lot of people find Big O is scared?
- But the underlying concepts are not that hard

We use Big O to describe the performance of an algorithm

# O(n)

- It just helps us to determine the given algoritm is scalable or not?
- Which basically means is this algorithm will scale well as the input grows really large
- Just because your code execute quickly in your computer it doesn't mean its is going to perform well when you give it a large data set
- That is why we use Big O Notation to describe the performance of an Algorithm

## Accessing an Array


- Accessing an array element by using its index is super fast
- but arrays have a fixed length and if you want to call / add / remove, we have to get resized
- this operation is costly as the input grows large

LINKED LIST

- so that's what we need to do / we have to use another data structure Linked List
- This data structures grow or shrink very quickly but accessing an element by its index in Linked List is slow

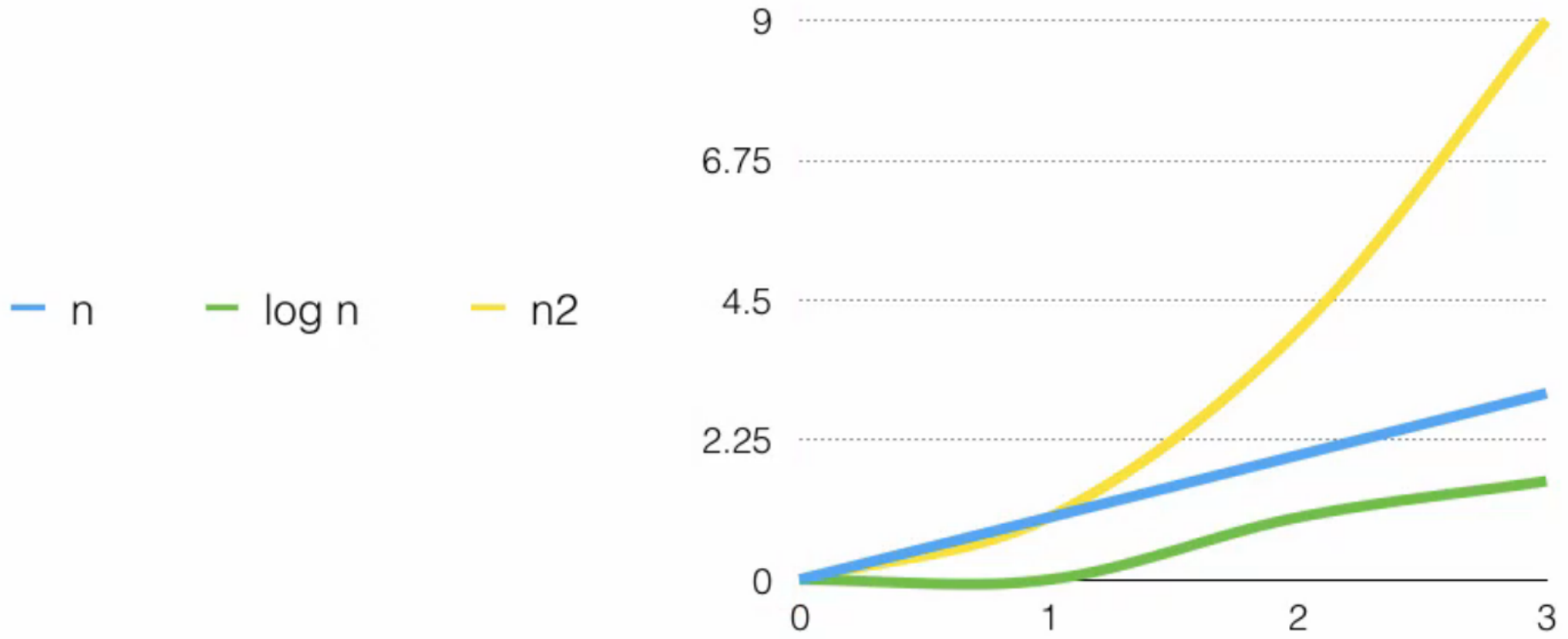
**“So that is why we need to learn Big O Notation first”**

- Big Companies like Google, Amazon , GE all companies ask about Big O Notation in interviews
- They wanted to know if you really understand how scalable an algorithm is
- Knowing Big O make you better Developer and Software Engineer

**O(*n*)** or **O(*log n*)**

O stands for order of the function or growth rate

The actual mathematical function



# Common Big O Algorithms

| Name | Big O Notation | Example |
|---|---|---|
| **Constant** | O(*1*) | return true; |
| **Logarithmic** | O(log *n*) | binary search |
| **Linear** | O(*n*) | for or while loop |
| **Quadratic** | O(*n^2*) | loop within a loop |
| **Exponential** | O(c^*n*) | recursive calls over n and looping over c in the function |
| **Factorial** | O(*n!*) | looping over n and recursive call in the loop to n-1 |

# Arrays

IN COMPUTER SCIENCE, AN ARRAY IS A DATA STRUCTURE CONSISTING OF A COLLECTION OF ELEMENTS, EACH IDENTIFIED BY AT LEAST ONE ARRAY INDEX OR KEY.

Can contain multiple instances of a type

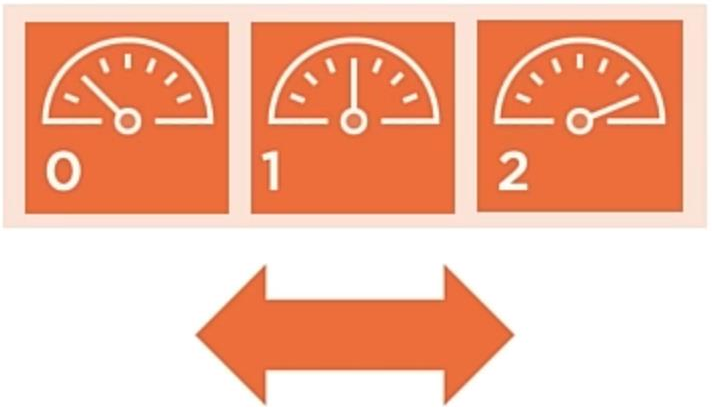

Can contain multiple instances of a type

Numeric indexing


Can contain multiple instances of a type

Numeric indexing

Access individual items

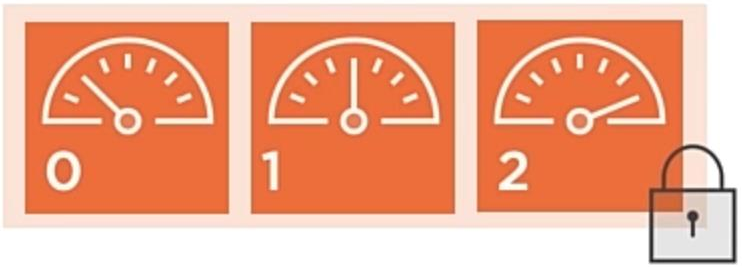



Can contain multiple instances of a type

Numeric indexing

Access individual items

Static or dynamic sizing




Can contain multiple instances of a type

Numeric indexing

Access individual items

Static or dynamic sizing

Fixed size once created

```
String[] cars;
```

# Creating an Array

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

# Adding Data to an Array

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
```

# Accessing Array Data

```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.length; i++) {
   System.out.println(cars[i]);
}
```

# Accessing Array Data

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
```

# Updating Array Values

# Asymptotic Analysis of Algorithms

Resources

**Operations**

The number of times we need to perform some operations

**Memory**

How much memory is consumed by the algorithms

**Others**

Network transfer, compression ratios, disk usage

# Queues

IN COMPUTER SCIENCE, A QUEUE IS A COLLECTION OF ENTITIES THAT ARE MAINTAINED IN A SEQUENCE AND CAN BE MODIFIED BY THE ADDITION OF ENTITIES AT ONE END OF THE SEQUENCE AND THE REMOVAL OF ENTITIES FROM THE OTHER END OF THE SEQUENCE

# What is Queue?

"A queue is an ordered line or sequence"

# Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

# Queue Characteristics

Queue Characteristics

"I'm first off to the powder, FIFO's rule"

| | Big O ? |
|---|---|
| enQueue | O(1) |
| deQueue | O(1) |
| Search | O(n) |
| Access | O(n) |

- Enqueue is a queue operation where you add an item at the back of a queue.
- Dequeue is a queue operation where you remove an item from the front of a queue.
- Search and Access in data/queue is linear in performance. Size of the queue how long it takes to find or access data in it

# HashMap

HASHMAP IS A DATA STRUCTURE THAT USES A HASH FUNCTION TO MAP IDENTIFYING VALUES, KNOWN AS KEYS, TO THEIR ASSOCIATED VALUES. IT CONTAINS “KEY-VALUE” PAIRS AND ALLOWS RETRIEVING VALUE BY KEY

# Hash Map

What is a HashMap used for?

HashMap is a data structure that uses a hash function to map identifying values, known as keys, to their associated values. It contains “key-value” pairs and allows retrieving value by key

Key? : Value?

name : venkat

Here **name** is key and value is **venkat**

The most impressive feature is it’s fast lookup of elements especially for large no. of elements. It is not synchronized by default but we can make it so by calling

# Hash Map example


SAMSUNG Galaxy Z Fold3 5G (Phantom Black, 512 GB)

4.1 ★ 89 Ratings & 4 Reviews

- 12 GB RAM | 512 GB ROM
- 19.3 cm (7.6 inch) QXGA+ Display
- 12MP + 12MP + 12MP | 10MP Front Camera
- 4400 mAh Lithium-ion Battery
- Qualcomm Snapdragon 888 Octa-Core Processor
- 1 Year Manufacturer Warranty for Device and 6 months Manufacturer Warranty for In-Box Accessories

{
Title: SAMSUNG Galaxy z Fold 5G(…)
Ram: 12 GB ram
Rom: 512 gb
Front camera: 10 mp
Rear camera: 12mp + 12 mp + 12 mp
Battery : 4400 mah
…

}

# Create HashMap in Java

Create a `HashMap` object called **capitalCities** that will store `String` **keys** and `String` **values**:

```
import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();
```

The `HashMap` class has many useful methods. For example, to add items to it, use the `put()` method:

## Example

```
// Import the HashMap class
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

## Access an item

```
capitalCities.get("England");
```

## Remove an item

```
capitalCities.remove("England");
```

## Clear all

```
capitalCities.clear();
```

## Read/loops Hashmap

```
for (String i : capitalCities.keySet()) {
      System.out.println(i);
}
```

# Quicksort

QUICKSORT IS A SORTING ALGORITHM, WHICH IS LEVERAGING THE DIVIDE-AND-CONQUER PRINCIPLE

# Quick Sort

Quicksort is a sorting algorithm, which is leveraging the divide-and-conquer principle. It has an average O(n log n) complexity and it’s one of the most used sorting algorithms, especially for big data volumes.

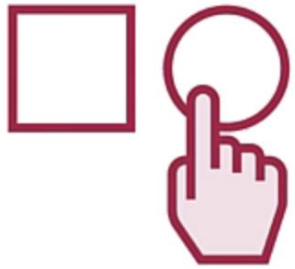
It's important to remember that Quicksort isn't a stable algorithm. A stable sorting algorithm is an algorithm where the elements with the same values appear in the same order in the sorted output as they appear in the input list.

The input list is divided into two sub-lists by an element called pivot;

one sub-list with elements less than the pivot and another one with elements greater than the pivot.

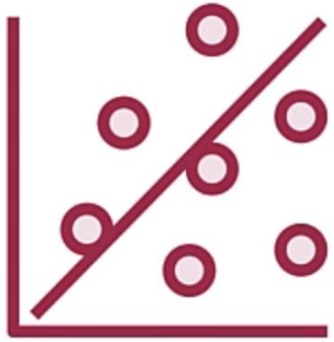
This process repeats for each sub-list.

# Quicksort



**Pivot**

Pick the pivot value in the array



**Partition**

Reorder the elements around the pivot point

**Repeat**

Repeat for each partition

# Algorithm Steps

- We choose an element from the list, called the pivot. We'll use it to divide the list into two sub-lists.
- We reorder all the elements around the pivot – the ones with smaller value are placed before it, and all the elements greater than the pivot after it. After this step, the pivot is in its final position. This is the important partition step.
- We apply the above steps recursively to both sub-lists on the left and right of the pivot.

As we can see, quicksort is naturally a recursive algorithm, like every divide and conquer approach.

# Example:

Let's take a simple example in order to better understand this algorithm.

```
Arr[] = {5, 9, 4, 6, 5, 3}
```

- Let's suppose we pick 5 as the pivot for simplicity
- We'll first put all elements less than 5 in the first position of the array: {3, 4, 5, 6, 5, 9}
- We'll then repeat it for the left sub-array {3,4}, taking 3 as the pivot
- There are no elements less than 3
- We apply quicksort on the sub-array in the right of the pivot, i.e. {4}
- This sub-array consists of only one sorted element
- We continue with the right part of the original array, {6, 5, 9} until we get the final ordered array

# Choosing optimal pivot

The crucial point in QuickSort is to choose the best pivot. The middle element is, of course, the best, as it would divide the list into two equal sub-lists.

But finding the middle element from an unordered list is difficult and time-consuming, that is why we take as pivot the first element, the last element, the median or any other random element.

# Implementing in java

```
public void quickSort(int arr[], int begin, int end) {
    if (begin < end) {
        int partitionIndex = partition(arr, begin, end);

        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
```

# Implementing in java

```java
public void quickSort(int arr[], int begin, int end) {
    if (begin < end) {
        int partitionIndex = partition(arr, begin, end);

        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
```

The first method is *quickSort()* which takes as parameters the array to be sorted, the first and the last index. First, we check the indices and continue only if there are still elements to be sorted.

We get the index of the sorted pivot and use it to recursively call *partition()* method with the same parameters as the *quickSort()* method, but with different indices:

Let's continue with the partition() method. For simplicity, this function takes the last element as the pivot. Then, checks each element and swaps it before the pivot if its value is smaller.

By the end of the partitioning, all elements less then the pivot are on the left of it and all elements greater then the pivot are on the right of it. The pivot is at its final sorted position and the function returns this position:

```
private int partition(int arr[], int begin, int end) {
    int pivot = arr[end];
    int i = (begin-1);

    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            i++;

            int swapTemp = arr[i];
            arr[i] = arr[j];
            arr[j] = swapTemp;
        }
    }

    int swapTemp = arr[i+1];
    arr[i+1] = arr[end];
    arr[end] = swapTemp;

    return i+1;
}
```

# Quicksort Algorithm Analysis (Time Complexity)

In the best case, the algorithm will divide the list into two equal size sub-lists. So, the first iteration of the full n-sized list needs O(n). Sorting the remaining two sub-lists with n/2 elements takes 2*O(n/2) each. As a result, the QuickSort algorithm has the complexity of O(n log n).

In the worst case, the algorithm will select only one element in each iteration,

so O(n) + O(n-1) + … + O(1), which is equal to O(n2).

On the average QuickSort has O(n log n) complexity, making it suitable for big data volumes.

# Merge sort

MERGE SORT IS ONE OF THE MOST EFFICIENT SORTING TECHNIQUES, AND IT'S BASED ON THE “DIVIDE AND CONQUER” PARADIGM.

# Merge Sort

Merge sort is one of the most efficient sorting techniques, and it's based on the “divide and conquer” paradigm.

**Split**

Split the array into sub-arrays of a single item

**Compare**

Compare the individual items

**Merge**

Merge the items into a sorted array

# The Algorithm

**Merge sort is a "divide and conquer" algorithm, wherein we first divide the problem into subproblems.** When the solutions for the subproblems are ready, we combine them together to get the final solution to the problem.

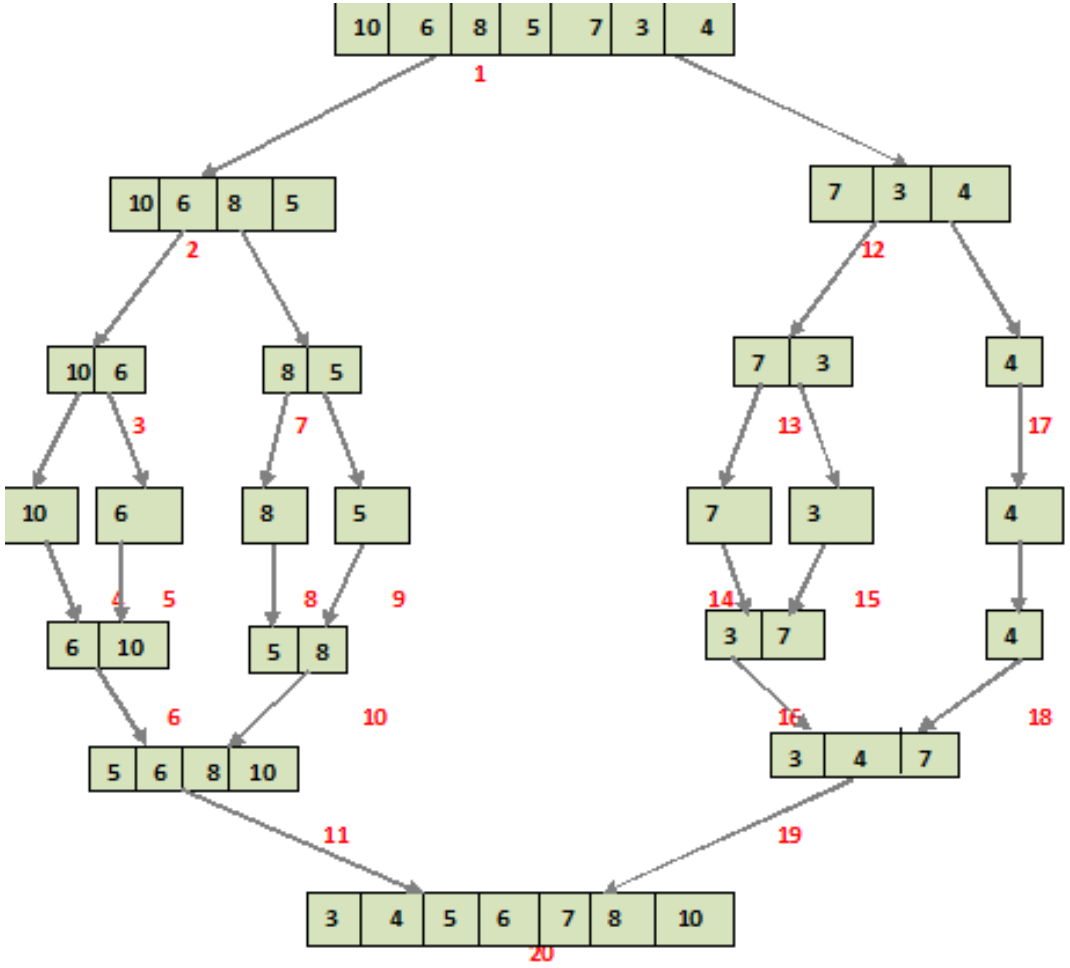
We can easily implement this algorithm using recursion, as we deal with the subproblems rather than the main problem.

We can describe the algorithm as the following 2 step process:

- **Divide: In this step, we divide the input array into 2 halves**, the pivot being the midpoint of the array. This step is carried out recursively for all the half arrays until there are no more half arrays to divide.
- **Conquer: In this step, we sort and merge the divided arrays** from bottom to top and get the sorted array.

The following diagram shows the complete merge sort process for an example array {10, 6, 8, 5, 7, 3, 4}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves until the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back while sorting:



# Implementation in Java

For the implementation, **we'll write a *mergeSort* function that takes in the input array and its length** as the parameters. This will be a recursive function, so we need the base and the recursive conditions.

The base condition checks if the array length is 1 and it will just return. For the rest of the cases, the recursive call will be executed.

**For the recursive case, we get the middle index and create two temporary arrays, *l[]* and *r[]***. Then we call the *mergeSort* function recursively for both the sub-arrays:

```java
public static void mergeSort(int[] a, int n) {
    if (n < 2) {
        return;
    }
    int mid = n / 2;
    int[] l = new int[mid];
    int[] r = new int[n - mid];

    for (int i = 0; i < mid; i++) {
        l[i] = a[i];
    }
    for (int i = mid; i < n; i++) {
        r[i - mid] = a[i];
    }
    mergeSort(l, mid);
    mergeSort(r, n - mid);

    merge(a, l, r, mid, n - mid);
}
```

**Next, we call the *merge* function, which takes in the input and both the sub-arrays, as well as the start and end indices of both the sub arrays.**
**The *merge* function compares the elements of both sub-arrays one by one and places the smaller element into the input array.**

When we reach the end of one of the sub-arrays, the rest of the elements from the other array are copied into the input array, thereby giving us the final sorted array:

```
public static void merge(int[] a, int[] l, int[] r, int left, int right) {

    int i = 0, j = 0, k = 0;
    while (i < left && j < right) {
        if (l[i] <= r[j]) {
            a[k++] = l[i++];
        }
        else {
            a[k++] = r[j++];
        }
    }
    while (i < left) {
        a[k++] = l[i++];
    }
    while (j < right) {
        a[k++] = r[j++];
    }
}
```

# Complexity

As merge sort is a recursive algorithm, the time complexity can be expressed as the following recursive relation:

```
T(n) = 2T(n/2) + O(n)
```

*2T(n/2)* corresponds to the time required to sort the sub-arrays, and *O(n)* is the time to merge the entire array.

When solved, **the time complexity will come to *O(nLogn).***

This is true for the worst, average, and best cases, since it'll always divide the array into two and then merge.

The space complexity of the algorithm is *O(n),* as we're creating temporary arrays in every recursive call.

Questions?