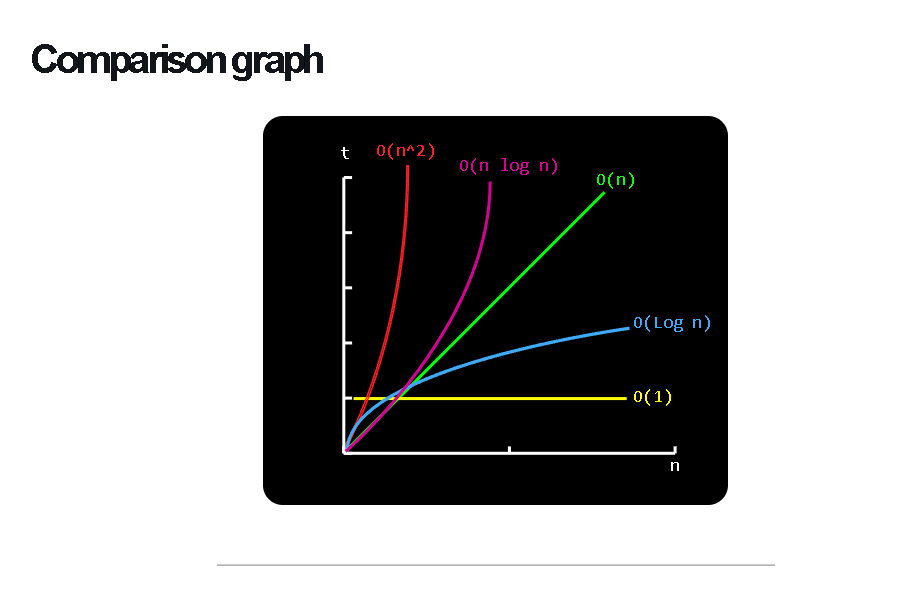
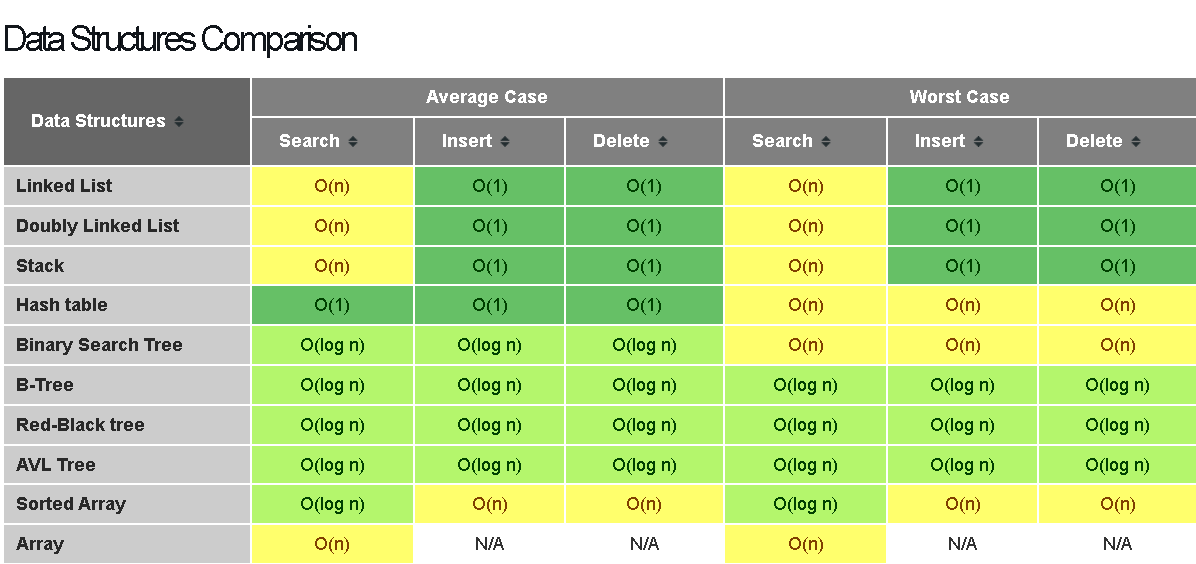
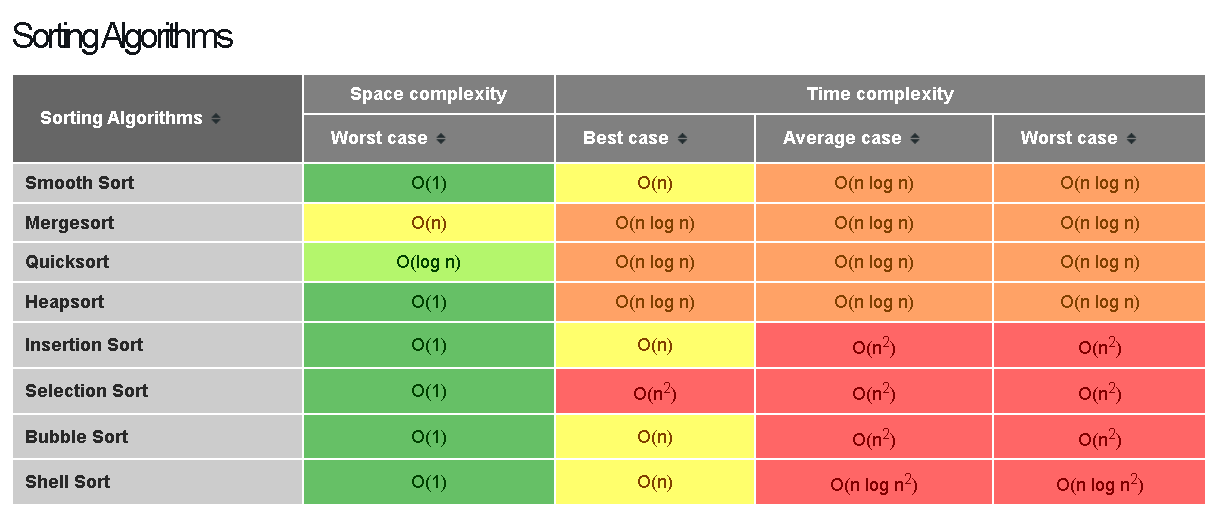
**ALGORITHMS & DATA STRUCTURES**

****

# INTRO :

## WHAT IS AN ALGORITHM :

=> Algorithms : a series of steps or rules for solving a *computational problem.*

* A *computational problem* is a collection of questions that computers might be able to solve.

=> An example to imagine easily :

| If ( clock.getTime() == 7am ) {  coffeMaker.boilWater();  if(water.isBoiled()){  coffeMaker.addCoffee();  coffeMaker.pourCoffee();  }  }else{  coffeMaker.doNothing();  } |
| --- |

## ALGORITHM EFFICIENCY :

### *Definition* :

* + Algorithm efficiency is the study of the amount of resources used by an algorithm.
* The less resources used by the algorithm the more efficient it is.
* The comparison between different algorithms :
  + How much time it needs to complete.
  + How much memory it uses to solve the problem.
  + How many operations it must do in order to solve the problem.

### *Include* :

* + **Time efficiency** : a measure of the amount of time an algorithm takes to solve a problem.
  + **Space efficiency** : a measure of the amount of memory an algorithm needs to solve a problem.
  + **Complexity theory** : a study of algorithm performance based on [*cost functions*](https://en.wikipedia.org/wiki/Cost_function) of statement counts.

# Big O notation :

## HOW TO DETERMINE COMPLEXITIES :

-> Determine complexity based on the type of statements used by a program.

=> *Why is big O notation commonly used?*

### Constant time( const time ) : O(1) => Even many codes -> still const time

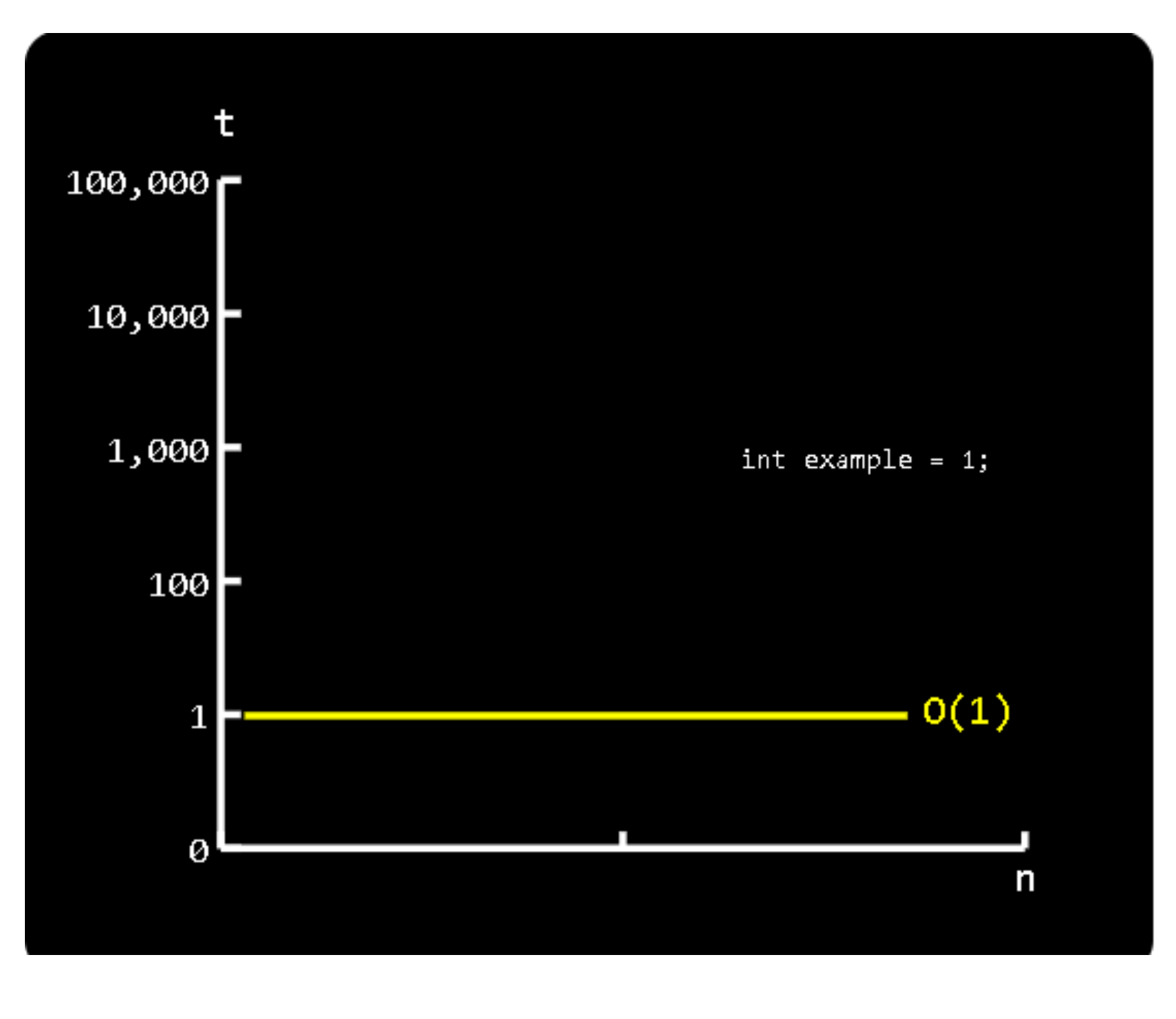
#### Include :

* + Assigning a value to some variable.
  + Inserting an element in an array
  + Determining if a binary number is even or odd.
  + Retrieving elements from an array.
  + Retrieving a value from a hash table(dictionary) with a key.

-> These are *simple* statements.

* Declaring a variable, inserting an element in a stack, inserting an element into an unsorted linked list.
* All these statements take *constant time*.

#### Graph :

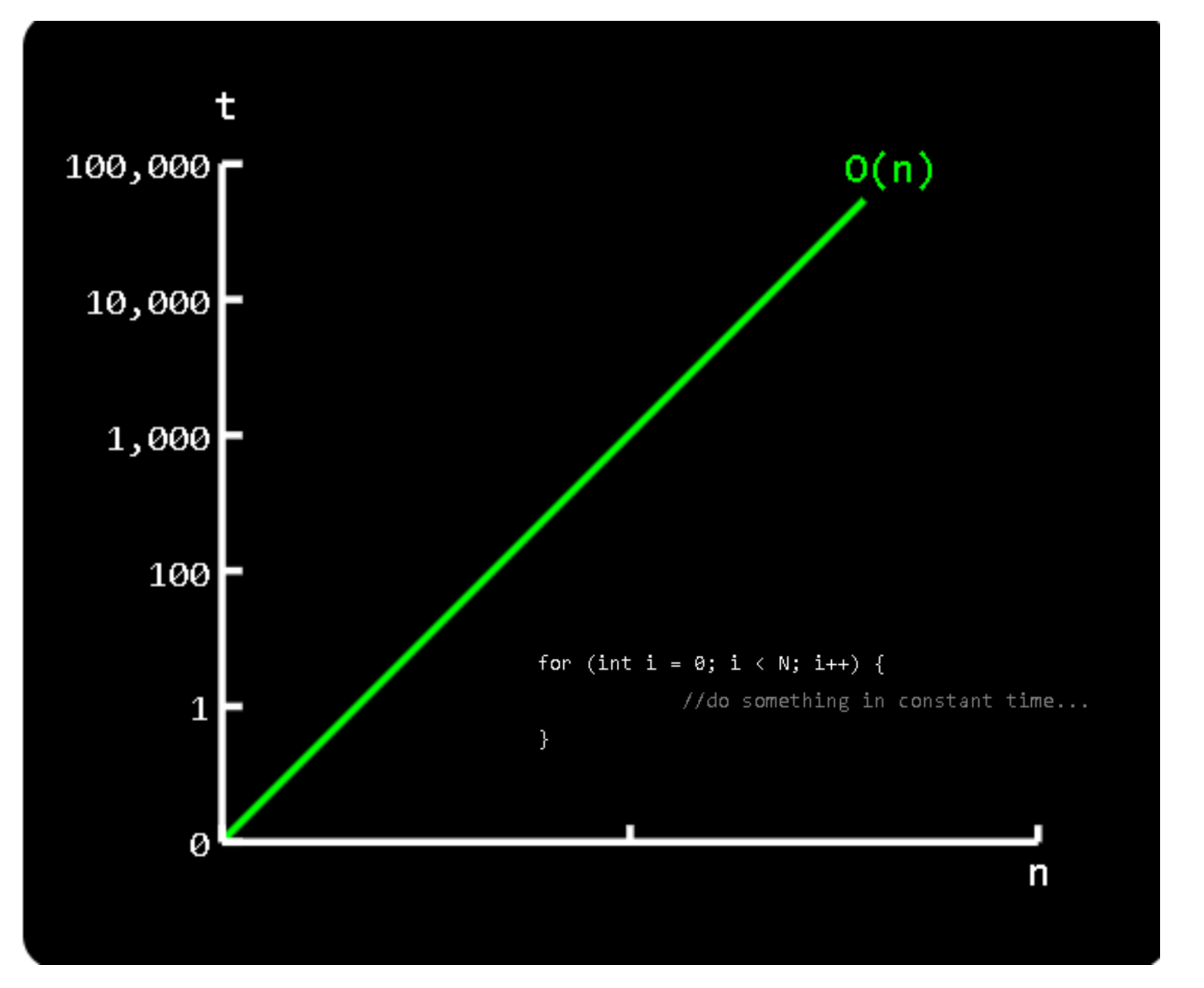


### Linear time: O(n)

#### Include :

* + Finding an item in an unsorted collection or an unbalanced tree (worst case).
  + Sorting an array via bubble sort

#### Graph :



### **Quadratic time** : **O( n2)**

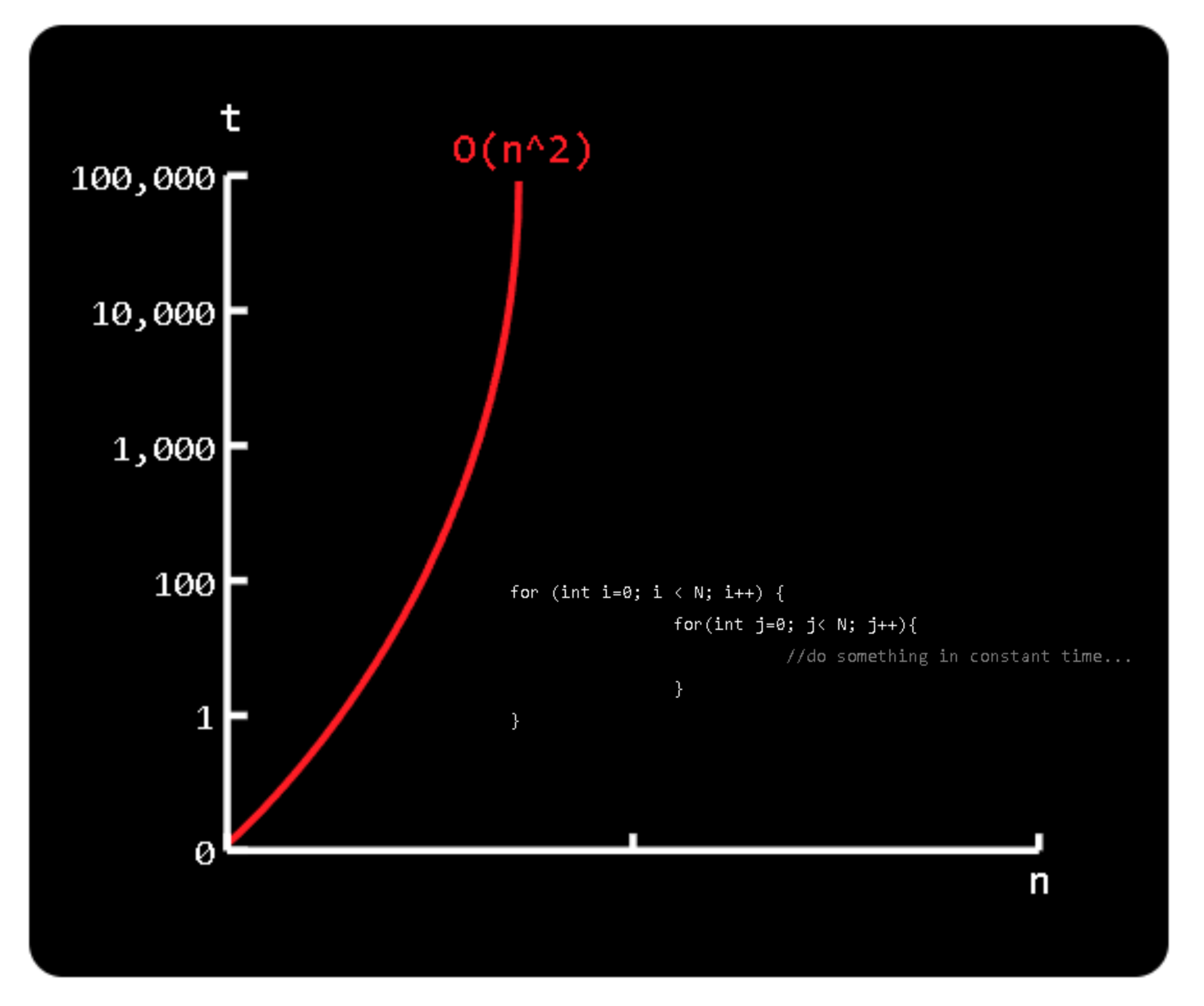
#### Include :

* + Performing linear search in a matrix.
  + Time complexity of quicksort, which is highly improbable.
  + Insertion sort.

=> ***Algorithms that scale in quadratic time are better to be avoided.***

=> *The larger the input, the longer the time.*

#### **Graph** :



### **Logarithmic time** : **O(log n)**

#### **Include** :

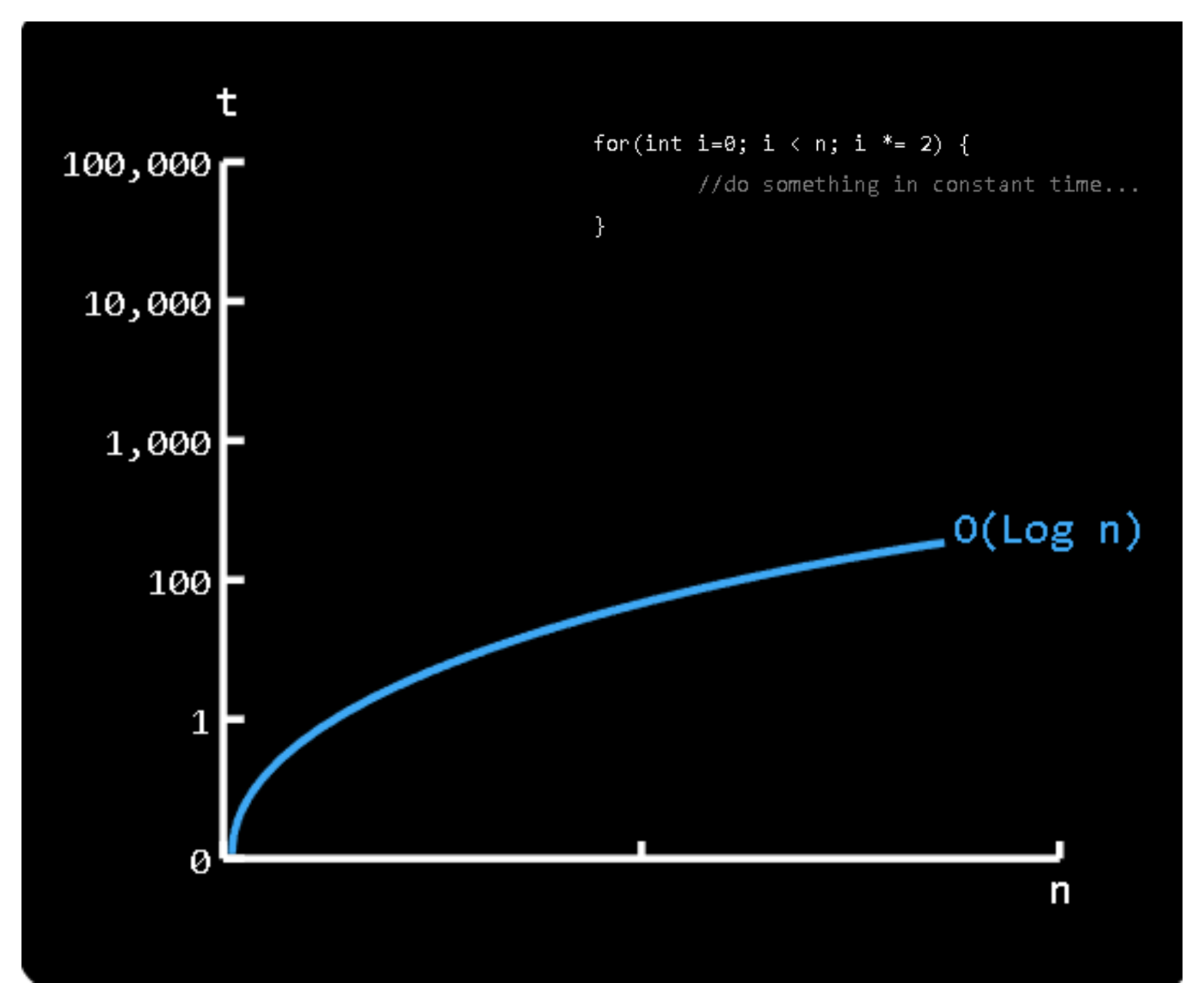
* + Binary search.
  + Insert or delete an element into a heap.

=> *Logarithms are the inverse operation of exponentiating something*

* *Logarithms appear when things are constantly halved or doubled.*

*=> Logarithmic algorithms have excellent performance in large data sets.*

#### **Graph** :



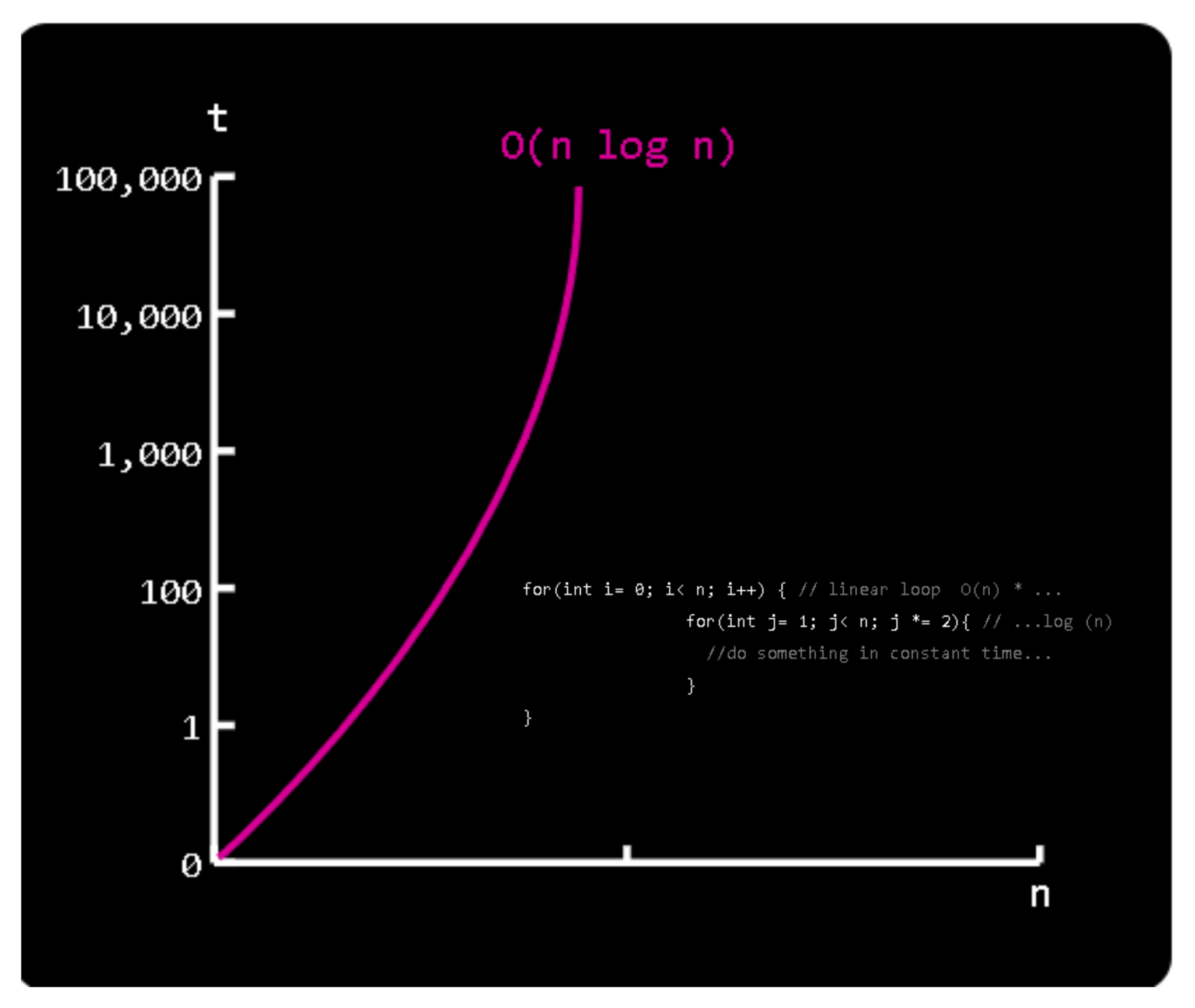
### **Linearithmic time**: **O(n\*log n)**

#### Include :

* + Heapsort
  + Merge sort
  + Quick sort

=> *Linearithmic algorithms are capable of good performance with very large data sets.*

#### Graph :



## BIG O NOTATION AND WORST CASE ANALYSIS :

* Big O notation focuses on the worst-case scenario.

#### Note :

* + The **best case** : O(1) -> happen **if** and **only if** the key is at the beginning of the list-> become more unlikely as n grows.
  + The **average case** : -> just make the task of analyzing an algorithm even more complex.
  + The  **worst case**  : O(n) -> When the key is at the end or never present in the list -> might happen.

# DATA STRUCTURES :

## DATA STRUCTURES :

### **Definition** :

* Data structure: are the fundamental constructs around which u build ur applications.
* Determine the way data is stored, and organized in the computer.
* When data exists -> It must have some kind of data structure -> to be stored in a computer

### **Contiguous or linked data structures** :

* Data structures can be classified as either contiguous or linked -> depending whether they are based on arrays or pointers :
  + **Contiguous-allocated structures :** are made of single slabs of memory.
    - Some of those data structures : Arrays, Matrices, Heaps, Hash tables.
  + **Linked data structures :** are composed as distinct chunks of memory linked together by pointers.
    - Some of those data structures : Lists, Trees, Graph adjacency lists.

### **Comparison** :

* Advantages of *linked lists over static arrays* :
  + Overflow is more difficult to occur on linked structures than it is in an array -> Only happens when the memory is actually full.
  + Insertion and deletion are simpler than for contiguous data structures such as arrays.
  + Linked lists don't need to know size on initialization.
* Advantages of *arrays* :
  + Linked structures require allocating extra space for storing pointers.
  + Arrays allow efficient access to any item.

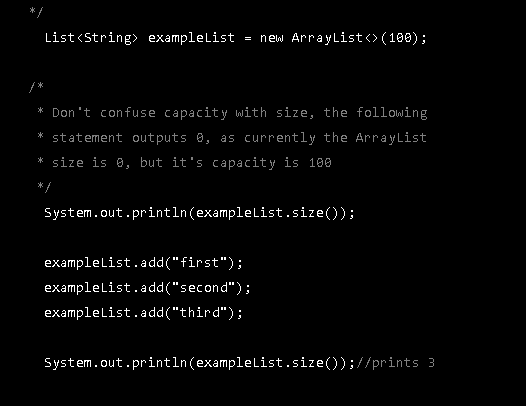
## ARRAY :

### **Definition** :

**=> Array :** the fundamental contiguously allocated data structure.

* + Have a fixed size and each element can be efficiently located by its index.

### **Example** ( JAVA ) :



## SET :

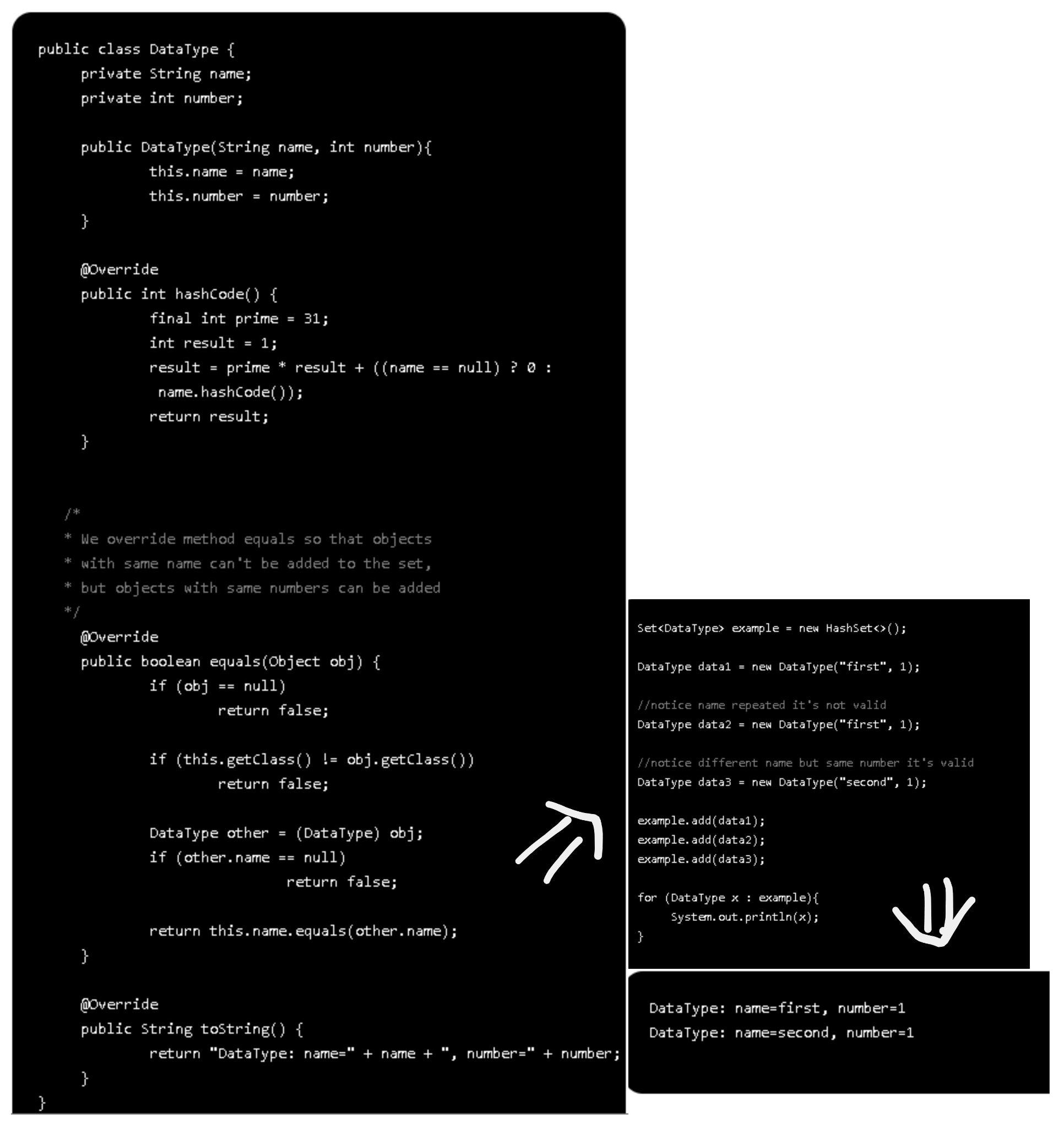
### **Definition** :

* A collection that cannot contain duplicate elements.( Each element must only appear 1 time ).

### **Methods declared** :

* **add() :** Add an object to the collection.
* **clear() :** Remove all objects from the collection.
* **contains() :** Return true if specified object is an element within the collection.
* **isEmpty() :**  Return true if the collection has no elements.
* **iterator() :** Return an Iterator object for the collection which may be used to retrieve an object.
* **remove() :** Remove a specified object from the collection.
* **size() :** Returns the number of elements in the collection.

### **Example** :

****

### **Multiset** :

* **Multiset** is similar to a **set**  but allows repeated values.
* This is perfect for when we need to perform statistical data that needs no sorting ( calculating the average or Standard Deviation of a multiset ).
* **Example** :
  + Apache Commons Collections provides the Bag and SortedBag interfaces, with implementing classes like HashBag and TreeBag.
  + Google Guava provides the Multiset interface, with implementing classes like HashMultiset and TreeMultiset.

## **STACKS AND QUEUES** :

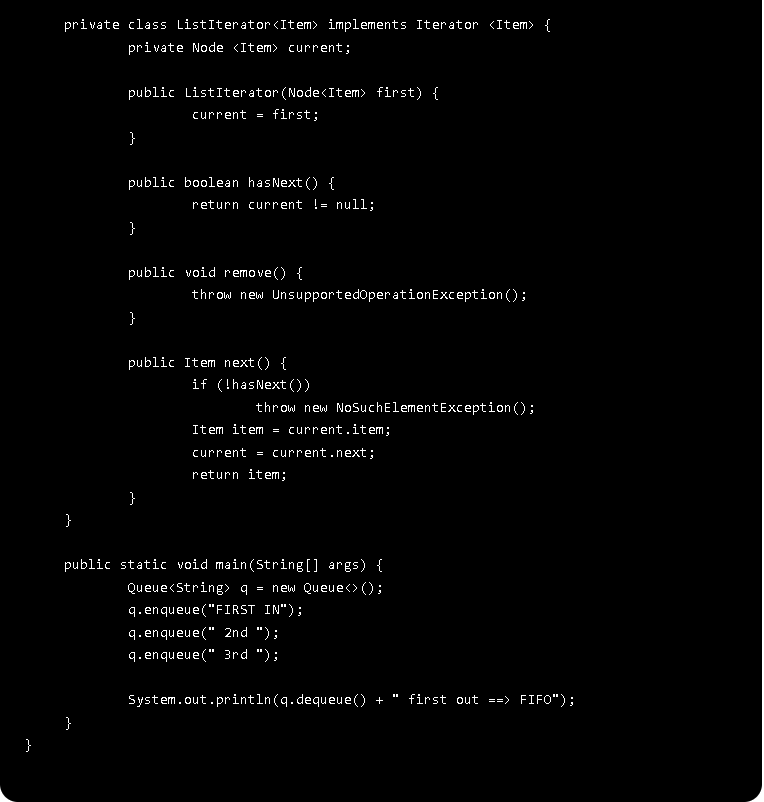
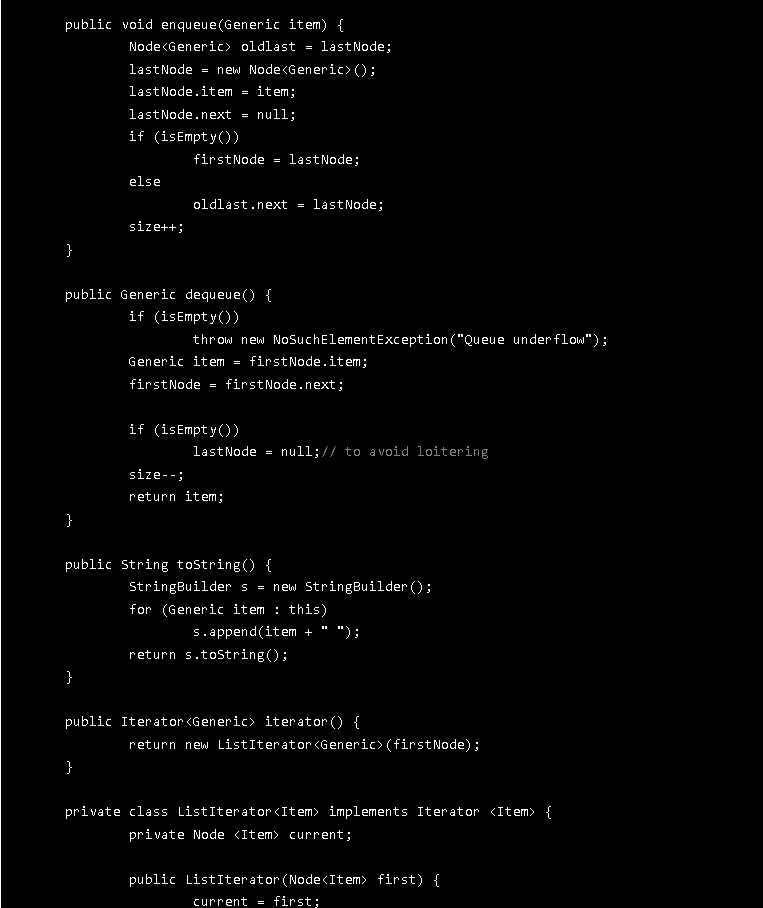
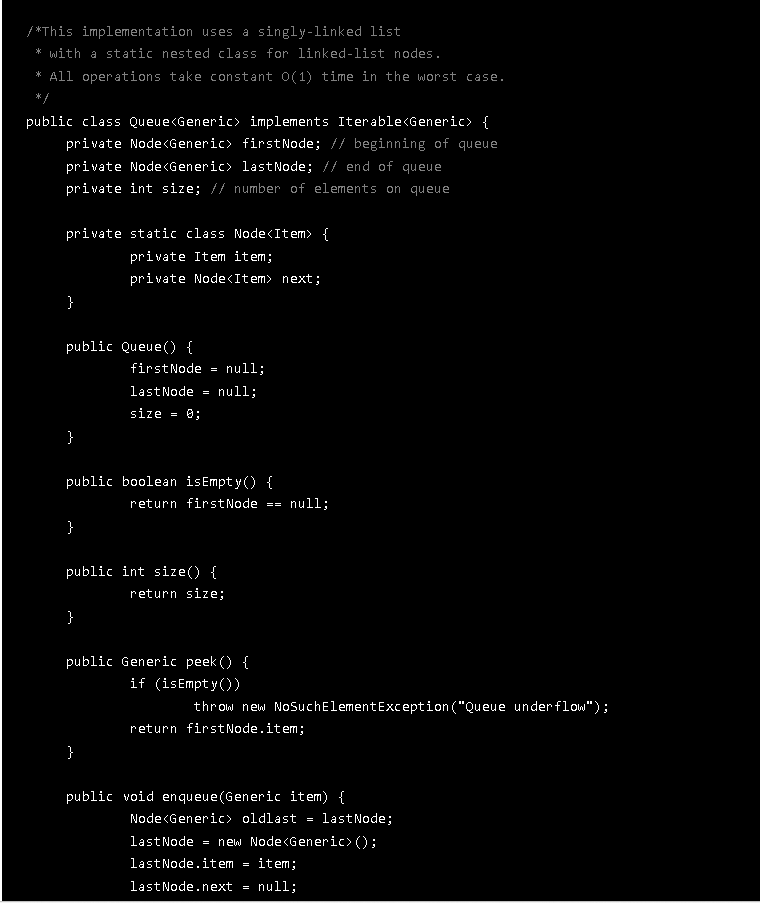
### **Definition** :

* The best way to complete tasks -> like a ***tool*** to complete and then discard.
* Useful to manage data in a more particular way than arrays and lists.
  + **Queues** are first in, first out ( FIFO )
  + **Stacks** are last in, first out ( LIFO ).

### **When should we use** ?

* Use a **queue**  when you want to get things out in the order that you put them in. ( right direction )
* Use a **stack** when you want to get things out in the reverse order than you put them in. ( opposite direction ).
* Use a **list** when you want to get anything out, regardless of when you put them in ( and don't want them to automatically be removed ).(like specify something)

### **Example** :

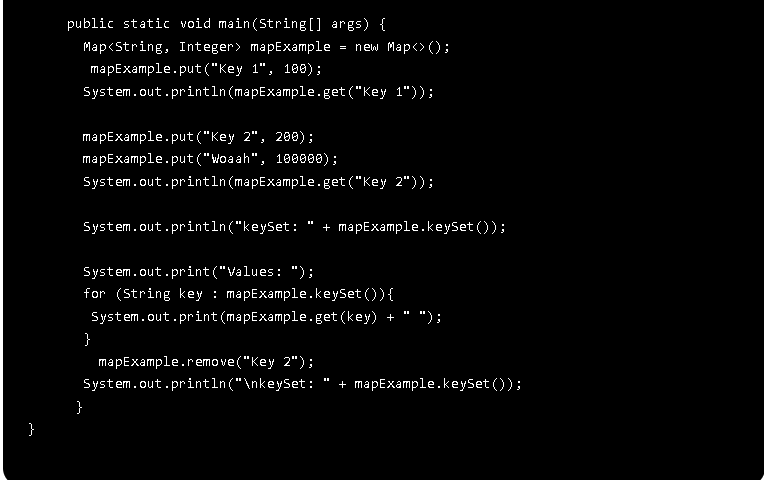
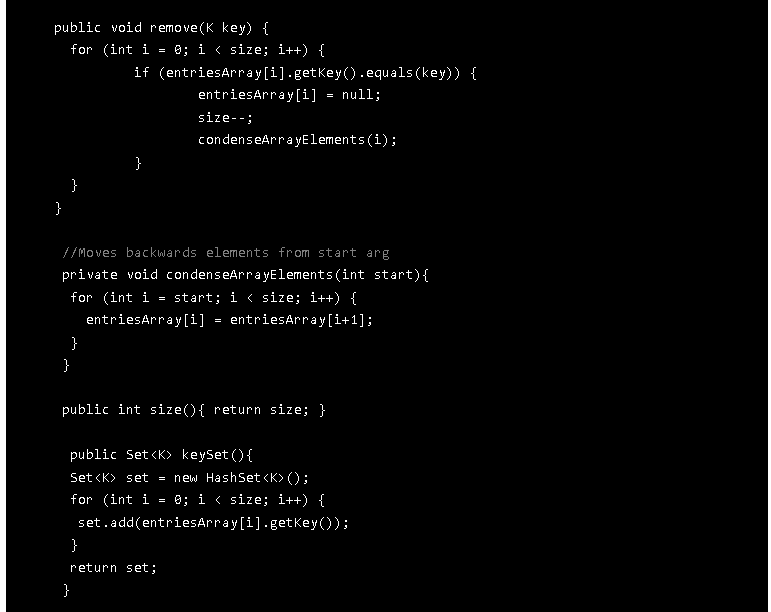
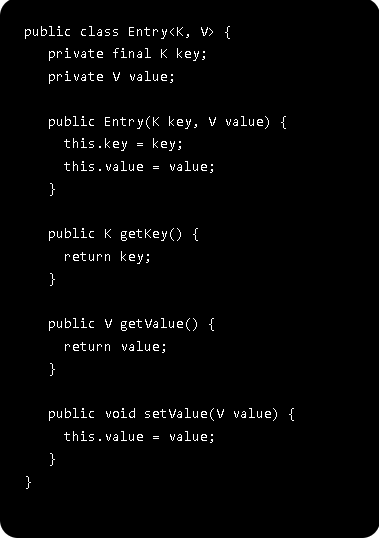


## DICTIONARY :

### **Definition** :

* **Dictionary** : ia a data structure that maps a **key** to a **value**.
* -> Useful in cases where you want to be able to access data via a particular key rather than an integer index.
* **Java** : Dictionary is implemented as a Map : The MAp interface maps unique keys to values.
  + A key is an obj that used to retrieve a value at a later date.
* Given a key and a value -> Can store the value in a Map obj -> Can retrieve by in an obj.

### **Example** :



# **ALGORITHMS** :

* 1. **SEARCHING AND SORTING** :

1. **Why is sorting so important** :

* The first step in organizing data.
* Make lots of tasks easier once a data set of items.
* Some algorithms ( binary search ,..) are built around a sorted data structure.
* Computers have historically spent the most on sorting.
* Remains the most ubiquitous combinatorial algorithm problem in practice.

1. **Considerations** :

**\*\*\*\* : *The questions are* :**

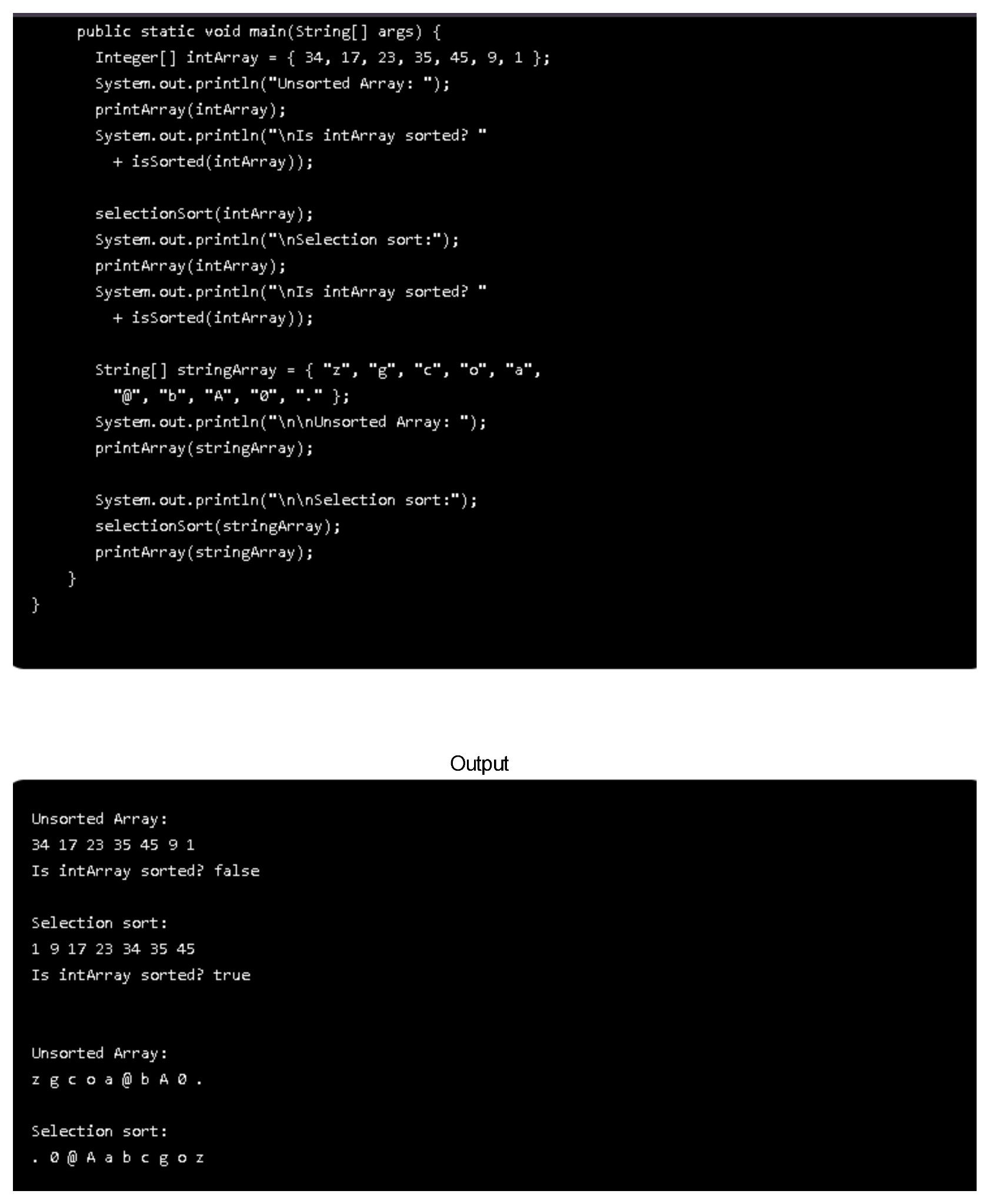
* How to sort : descending order or ascending order?
* Sorting based on what?
  + An obj name : alphabetically (a -> z)
  + By some number defined by its fields/instance variable.
  + Compare dates, birthdays, etc.
* What happens with equals keys?
  + Ex : various people with the same name : John then sort them by Last Name.
* Does your sorting algorithm sorts in place or needs extra memory to hold another copy of the array to be sorted.
  + => More important in embedded systems.
* An example in JAVA :
  + Java uses Comparable interface for sorting and returning:
    - +1 if compared obj is greater.
    - -1 if compared obj if less.
    - 0 if compared objs are equal.
* => Sorting becomes more ubiquitous when we think on all the things we do daily that are previously sorted for us to understand and have better access to them :
  + Imagine trying to find a phone number in an unsorted phone book , or searching for a word in an unsorted dictionary.
  + MP3 player can sort the lists by artist name, genre, song name, ratings.
  + Searching engines display results in descending order of importance.
  + Spreadsheets can be sorted in various ways to work better with their contents.

1. **Searching** :

* 2 types of searching algorithms :
  + Need a previously ordered data structure in order to work properly.
  + Don't need an ordered list.
* Searching is very important for many computing applications :
  + Searching through a search engine.
  + Finding a bank account balance for some clients.
  + Searching in a large data set for a particular value.
  + Searching in the directories for some needed files.

=> If the application is completed but takes too long perform a search and retrieve data -> Discarded as useless.

* 1. **SELECTION SORT** :
* It repeatedly selects the smallest remaining item:
  + Find the smallest element -> Swap it with the first element.
  + Find the second smallest element -> Swap it with the second element.
  + Find the third smallest element -> Swap it with the third element.
  + Repeat finding the smallest element and swapping in the correct position until the list is sorted.
* An example in JAVA :



* 1. **SHELLSORT** :
     1. **Definition** :
* A perfect good for embedded systems as it does not require a lot of extra memory allocation
  + => Useful for small to medium size arrays
    1. **Example** :

