ĐẠI HỌC ĐÔNG Á
Tạo dựng con đường thành công

DATA STRUCTURES AND ALGORITHMS

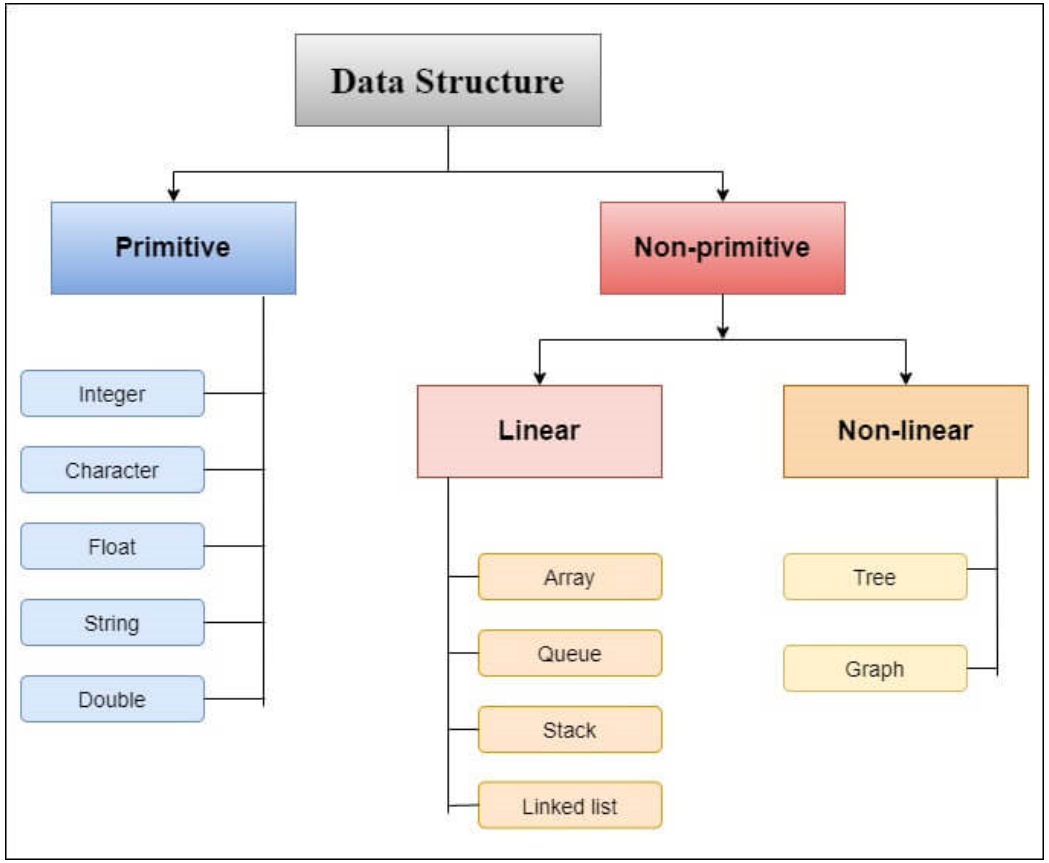
Introduction to Data Structures & Algorithms

What is data structure?

- Data structure is a way of storing data in a computer so that it can be used efficiently.
- A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other.
- Often a carefully chosen data structure will allow a more efficient algorithm to be used.
- The choice of the data structure often begins from the choice of an abstract data structure. A well-designed data structure allows a variety of critical operations to be performed on using as little resources, both execution time and memory space, as possible.

What is data structure?

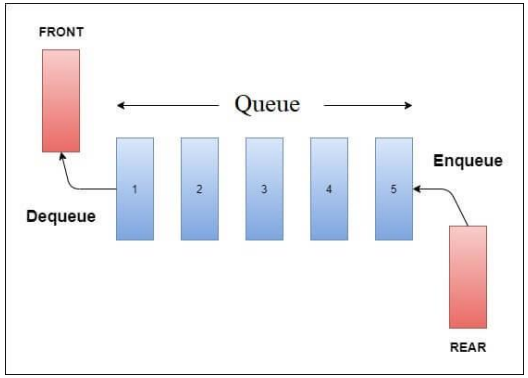
Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.




```
public class QueueDemo {

    public static void main(String[] JavaLatte) {
        Queue queue = Queue.createQueue();
        queue.enQueue(4);
        queue.enQueue(8);
        queue.enQueue(2);
        queue.enQueue(1);
        queue.enQueue(40);
        if(queue.isEmpty()){
            System.out.println("Queue is empty");
        }
        System.out.println(queue.front());
        System.out.println(queue.deQueue());
        System.out.println(queue.deQueue());
        System.out.println(queue.deQueue());

    }
}
```



Why data structure is important in computer science?

Data structures are used in almost every program or software system. Data structures provide a means to manage huge amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Niklaus Emil Wirth (born February 15, 1934) is a Swiss computer scientist said:

Algorithms + Data Structures = Programs

Basic types of data structure

In programming, the term *data structure* refers to a scheme for organizing related pieces of information. The basic types of data structures include:

- **files**
- **lists**
- **arrays**
- records
- trees
- tables

Each of these basic structures has many variations and allows different operations to be performed on the data.

Store Data

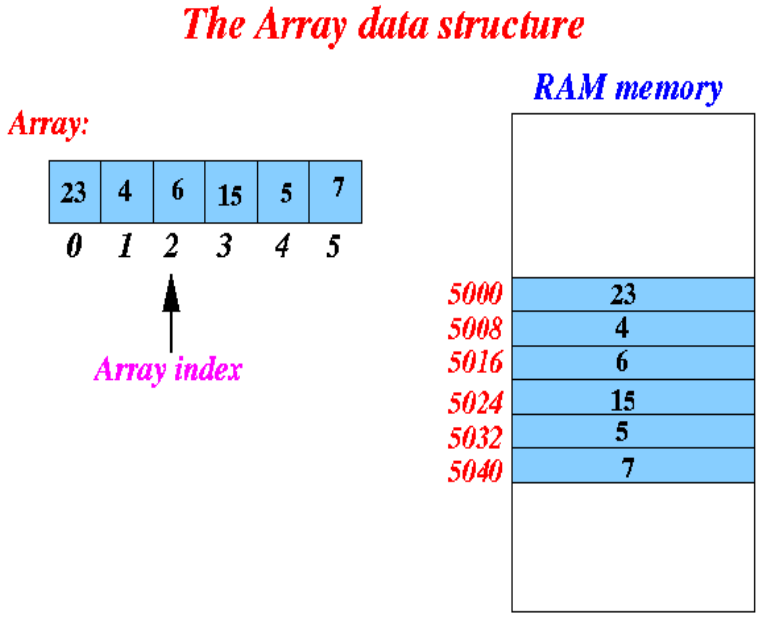
- It is easy to notice, that the way we **read** the data **retrieves the information** of the bits in different ways. However those bi ts have only **0** or **1** as values.
- Example:

Type	Binary Data	Translation
Integer	0000 0100 0001	65
Character	0000 0100 0001	'A'
Double	0000 0100 0001	65.0
Instruction Code	0000 0100 0001	Store 65
Color	0000 0100 0001	

Store Data

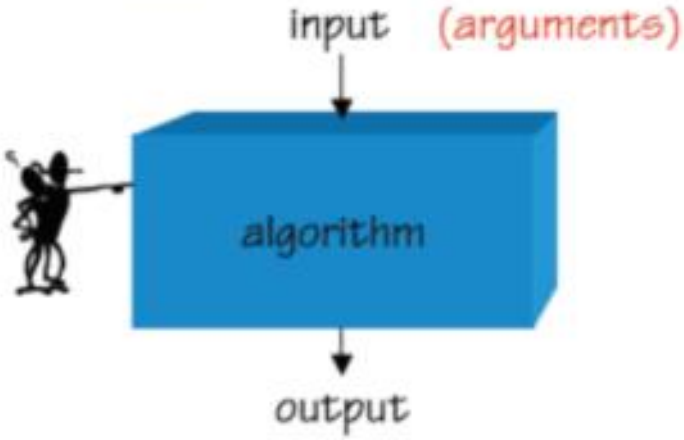
- Storing items **requires memory consumption**:

Data Structure	Size
int	= 4 bytes
float	= 4 bytes
long	= 8 bytes
int[]	≈ (Array length) * 4 bytes
List<Double>	≈ (List size) * 8 bytes
Map<Integer, int[]>	≈ (Map size) * Entry bytes



What is algorithm?

- An algorithm is a sequence of instructions that one must perform in order to solve a well formulated problem.



Problem: Complexity

Algorithm: Correctness
Complexity

A formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language, from natural languages like English or French to programming languages like FORTRAN.

Algorithm example

Definition: steps to solve a problem.

Input → Process → Output

Data structure + Algorithm → program

Example 1:

To find a summation of N numbers in an array of A.

```
1 Set sum = 0
2 for j = 0 to N-1 do :
3 begin
        sum = sum + A[j]
4 end
```

What are the properties of an algorithm?

1. **Input** - an algorithm accepts zero or more inputs
2. **Output** - it produces at least one output.
3. **Finiteness** - an algorithm terminates after a finite numbers of steps.
4. **Definiteness** - each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.
5. **Effectiveness** - it consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.
6. **Generality** - it must work for a general set of inputs.

How to represent algorithms?

- Use natural languages
  - too verbose
  - too "context-sensitive"- relies on experience of reader
- Use formal programming languages
  - too low level
  - requires us to deal with complicated syntax of programming language
- Pseudo-Code (alias programming language) - natural language constructs modeled to look like statements available in many programming languages.
- Flowchart (diagram) - A flowchart is a common type of chart, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Algorithm representation examples

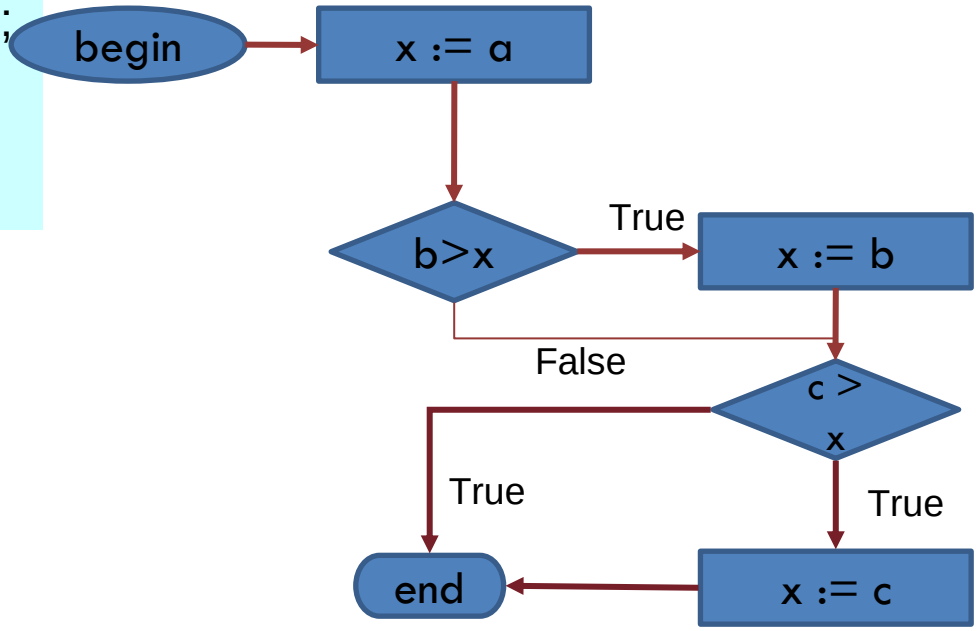
Problem: find the greatest of three numbers

Natural language:

find max(a,b,c)

- Assign x = a
- If b great than x then assign x=b
- If c great than x then assign x=c;
- Result: x => max(a,b,c)

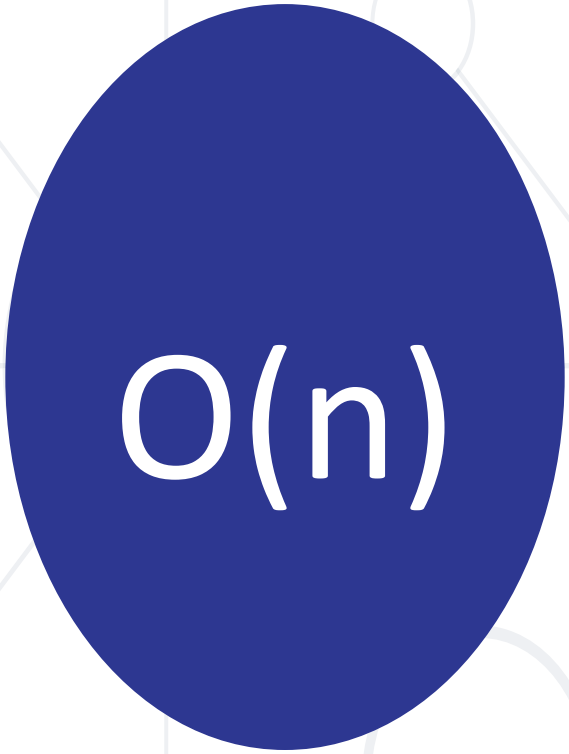
Flowchart:



Pseudo-code:

```
function max(a,b,c)
Input: a, b, c
Output: max(a,b,c)
        x = a;
        if b > x then x = b;
        if c > x then x = c;
        return x;
```

ĐẠI HỌC ĐÔNG Á
Tạo dựng con đường thành công



Algorithmic Complexity

Asymptotic Notation

Algorithm Analysis

- Why should we analyze algorithms?
  - Predict the **resources** the algorithm will need
    - Computational time (**CPU** consumption)
    - Memory space (**RAM** consumption)
    - Communication **bandwidth** consumption
    - **Hard disk** operations

Algorithm Analysis

- There are three main properties we want to analyze:
  - **Simplicity** – this is really a matter of intuition and of course it is subjective quality
  - **Accuracy** – this seems easy to determine, however it may be very difficult to determine is the algorithm correct?
  - **Performance** – the consumption of CPU, Memory and other resources (we really care the most for the first two)

Algorithm Analysis (3)

- The expected **running time** of an algorithm is:
  - The total number of **primitive operations** executed (machine independent steps)
  - Also known as **algorithm complexity**
  - Compare algorithms **ignoring details** such as

language or **hardware**

Time Complexity

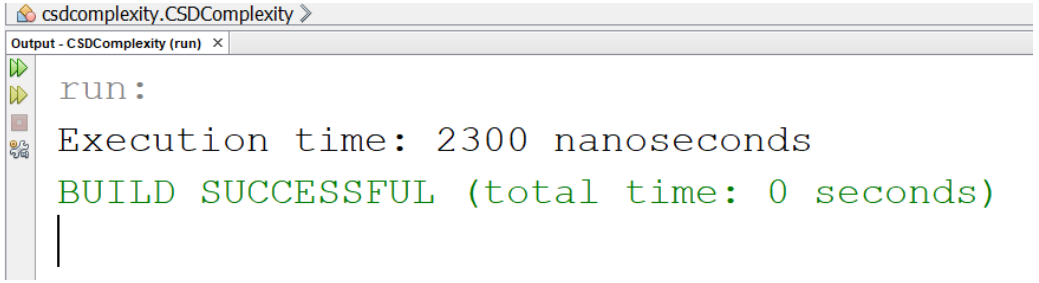
- Define the time complexity of the following code:

```
boolean contains(int[] numbers, int number) {
  for (int i = 0; i < numbers.length; i++) {
      if (numbers[i] == number) {
         return true;
      }
  }
  return false;
}
```

- It is not as simple as the previous, **when** does the code return?

Time Complexity

```
12 public class CSDComplexity {
13
14     boolean contains(int[] numbers, int number) {
15         for (int i = 0; i < numbers.length; i++) {
16             if (numbers[i] == number) {
17                 return true;
18             }
19         }
20         return false;
21     }
22
23     public static void main(String[] args) {
24         // TODO code application logic here
25         // create an object of the Main class
26         CSDComplexity obj = new CSDComplexity();
27         int[] numbers = {1, 2, 99, 5, 32, 4};
28         int number = 4;
29         // get the start time
30         long start = System.nanoTime();
31         // call the method
32         obj.contains(numbers, number);
33         // get the end time
34         long end = System.nanoTime();
35
36         // execution time
37         long execution = end - start;
38         System.out.println("Execution time: " + execution + " nanoseconds");
39     }
40
41 }
42
```

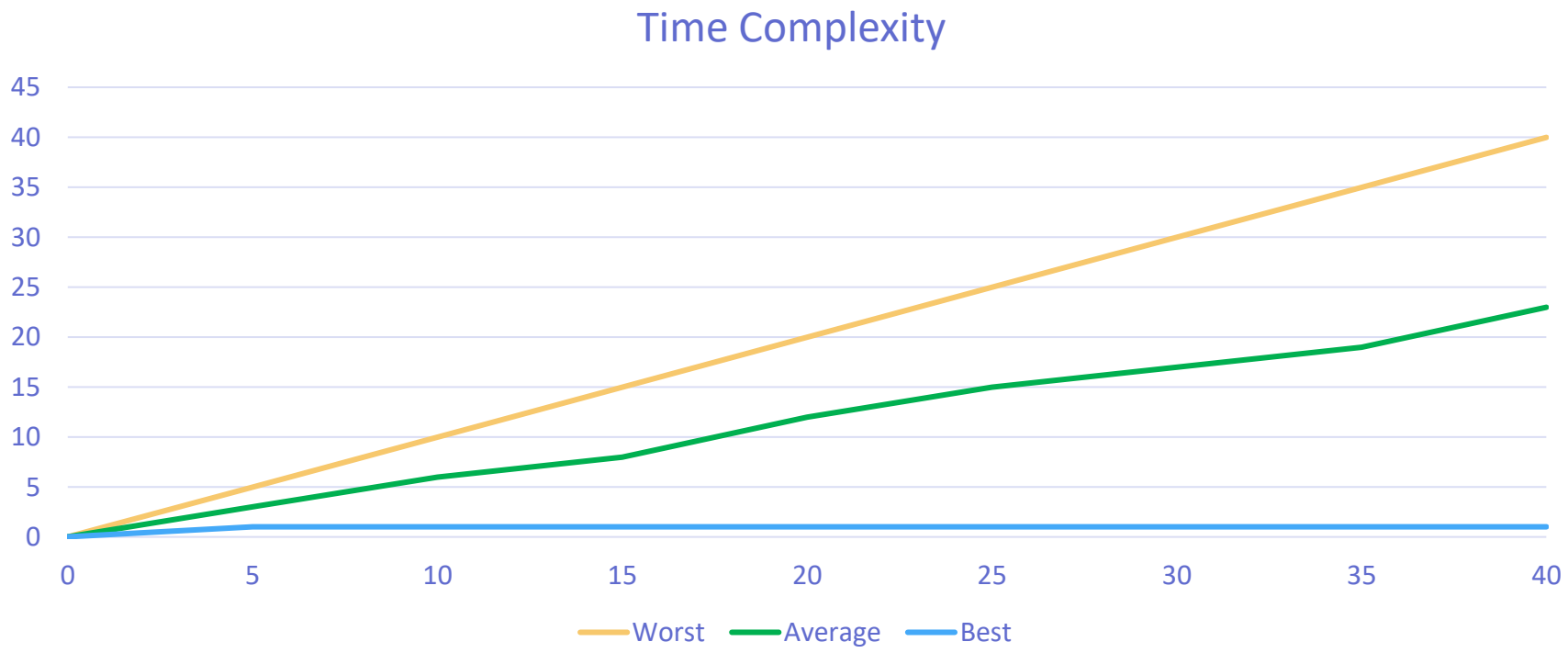


Time Complexity

- Worst-case
  - An **upper** bound on the running time
- Average-case
  - **Average** running time
- Best-case
  - The **lower** bound on the running time (the optimal case)

Time Complexity

- Therefore, we need to measure **all** the possibilities:



Asymptotic notations

- **Asymptotic notations** are descriptions that allow us to examine an algorithm's running time by expres sing its **performance** as the input size, **n**, of an algo rithm or a function **f increases**. There are **three** com mon asymptotic notations:
  - Big **O** – **O(f(n))**
  - Big **Theta** – **Θ(f(n))**
  - Big **Omega** – **Ω(f(n))**

Algorithms Complexity

- **Algorithmic complexity** – rough estimation of the number of st eps performed by given computation, depending on the size of the input
- Measured with asymptotic notation
  - `O(f(n))` – upper bound (worst case)
  - `Θ(f(n))` – average case
  - `Ω(f(n))` – lower bound (best case)
    - Where `f(n)` is a function of the size of the input data

Typical Complexities

Complexity	Notation	Description
constant	O(1)	n = 1 000 → 1-2 operations
logarithmic	O(log n)	n = 1 000 → 10 operations
linear	O(n)	n = 1 000 → 1 000 operations
linearithmic	O(n*log n)	n = 1 000 → 10 000 operations
quadratic	O(n2)	n = 1 000 → 1 000 000 operations
cubic	O(n3)	n = 1 000 → 1 000 000 000 operations
exponential	O(n^n)	n = 10 → 10 000 000 000 operations

Data Structures and Algorithms

- Most algorithms operate on data collections, so define
- Collection Abstract Data Type (ADT)
  - Methods
    - Constructor / Destructor
    - Add / Edit / Delete
    - Find
    - Sort
    - ….

Abstract Data Type (ADT)

- An **Abstract Data Type (ADT)** – the way the real objects will be modulated as **mathematical** objects, alongside the **set of operations** to be executed upon them, **without** the implement ation itself.

```
public interface List<E> {
    boolean add(E e);
    int size();
    boolean remove(Object o);
    boolean isEmplty();
}
```

Data Structures Implementation

- An **implementation** – definitive way of ADS to be presented ins ide the computer memory, alongside the implementation o f the operations.

```
public class ArrayList<E> implements List<E> {
    public boolean add(E e) {
        this.elements[this.index++] = e;
        this.size++;
        return true;
    }
}
```