Problem Solving Strategies and Computational Approaches SCS1304

Handout 1 : Introduction to Computational Thinking

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

Overview

- This course introduces students to the techniques and methodologies for analyzing and solving complex problems in a structured manner by equipping them with the skills necessary to effectively analyze system requirements, identify problems, and devise solutions using structured approaches.

- CS1304 aims to lay sound a foundation or designing algorithms for real world problems using pseudocodes and designing flowcharts (i.e without any coding)

Indicative Content

- **Introduction to Computational Thinking**
  - ❖ Decomposition, Pattern Recognition, Abstraction, Algorithmic Thinking
  - ❖ Linear Search, Binary Search, Fibonacci Sequence, Introduction to Recursion, Types of Recursion (Linear Recursion, Nested Recursion, Tail Recursion, Tree Recursion, Head Recursion & Indirect Recursion etc.,), Applications of Recursion, Best Practices and Pitfalls in Recursive Algorithms, Recursion vs Loops/Iteration
- **Asymptotic Notation**
  - ❖ Analysis of Algorithms, Order of Algorithms, Practical Considerations, Every-case time complexity, Worst-case time complexity, Best-case time complexity, Average-case time complexity.

Indicative Content ctd..

- **Problem-Solving Strategies**
  - Typical Strategies, Trial and Error, Algorithm and Heuristic, Means-Ends Analysis, The problem-solving Process
  - Brute Force: Origins of "Brute Force“ approach, Brute Force in Computing, Brute Force Algorithms, Time Complexities of Brute Force Algorithms, Advantages and Disadvantages, Optimizing Brute Force, Alternatives to Brute Force, Brute Force Applications
  - Divide and Conquer: Merge Sort, Quick Sort, Strassen’s Matrix Multiplication, Further examples of Divide and Conquer Algorithms.
- **Dynamic Programming**
  - Introduction to Dynamic Programming (DP), Divide Conquer vs Dynamic Programming, Examples of DP, Applicability of DP in solving problems, Approaches to DP – Tabulation & Memoization, Further examples of DP.

Indicative Content ctd..

- **Greedy** Algorithms
  - Coin Change Problem, Greedy vs Not Greedy Algorithms, Activity Selection Problem, dynamic programming vs Greedy Approach, Knapsack Problem, Spanning Trees, Algorithms to find Minimum Spanning Tree, Kruskal's Minimum Spanning Tree, Algorithm, Prim's Minimum Spanning Tree Algorithm, Further examples of Greedy algorithms.
- **Backtracking**
  - Maze Problem, State Space Trees, Backtracking Applications, Sudoku, N-queen problem using Backtracking, Hamiltonian Path, Hamiltonian Cycle using Backtracking, Optimizing backtracking, Further examples of Backtracking algorithms.
- **Branch Bound**
  - Traveling Salesman Problem, Branch Bound (FIFO, LIFO, Least Cost-Branch and Bound), Further examples of Branch Bound algorithms

Course Objectives

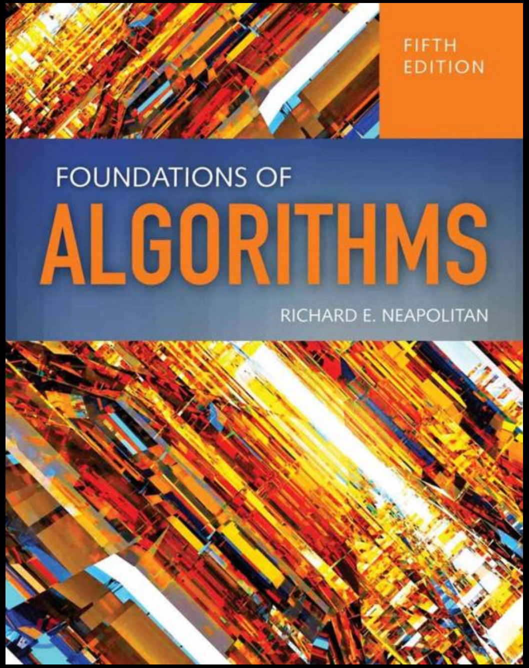
- to provide students with a basic of understanding of how to write and analyze algorithms.
- to impart to them the skills needed to write algorithms using the standard algorithm design strategies.
- to provide a sound foundation required for developing strong software engineering skills.
- to provide an understanding of some common algorithms, as well as some general approaches to developing algorithms.
- to develop skills to device a solution to a computational problem by answering to a problem, but the best answer.
- to be able to evaluate an algorithm and analyze how its performance is affected by the size of the input so that one can choose the best algorithm to solve a given problem.

Course Overview

- Credits : 2C (2L + 0P)
  Rubric:
  - 60% Exam, 40% Assignments (Supervised Quizzes)
- Delivery: 2-hour Lecture + 2-hour Tutorials per week (important!)
- References: BOOKS and online Materials

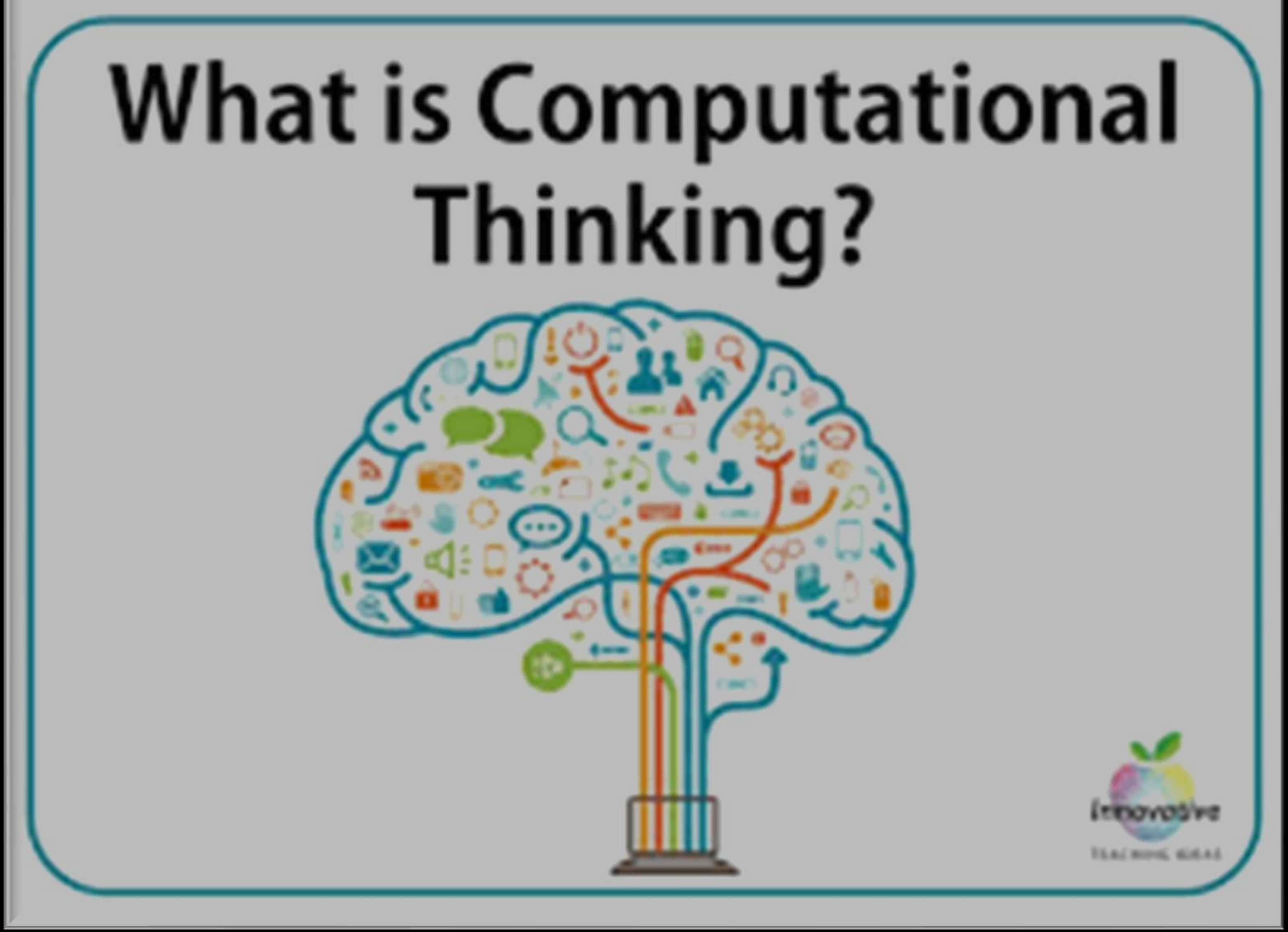
  Neapolitan, R. and Naimipour, K., 2015. Foundations of Algorithms. 5th Edition, Jones & Bartlett Publishers.

  https://github.com/davidkmw0810/argorithm/tree/master/knap



Handout 1: Overview

- Introduction to Computational thinking
  - decomposition, pattern recognition, abstraction, and algorithm design.
- Significance of understanding each and their roles in computational problem solving



Computational thinking is a set of skills and processes that enable one to navigate complex problems

Refer

- **Computational Thinking**
- https://www.youtube.com/watch?v=dHWmnayy8MY&t=15s

- **Solving Problems at Google Using Computational Thinking**
- https://www.youtube.com/watch?v=SVVB5RQfYxk

- **Ref: Computational Thinking Defined**
- **https://towardsdatascience.com/computational-thinking-defined-7806ffc70f5e**

Computer Science & Problem Solving

- Computer science is the discipline of how to solve problems using computers
- We aim for efficient, general solutions that will work on a wide variety of problems
- Although computer science has existed as a field for about 70 years, its roots in mathematics and computation go back thousands of years!
- CS is a field that relies partly on old mathematical ideas but experiences advances in development of new techniques at an extraordinary pace

https://www3.cs.stonybrook.edu/~alexkuhn/cse101-spring2021/slides/CSE101-Chapter1-S21.pdf

Computational Thinking ?

- helps one to think logically and clearly, whether you’re writing code, solving math problems, or planning something in everyday life
- Approaching a problem in a systematic manner
- Creating and describing a solution to a problem
- Work out how a computer or a person can solve a problem

Computational Thinking ?

1. Breaking a big problem into smaller parts (this is called decomposition).
2. Finding patterns in the problem that repeat.
3. Ignoring unnecessary details and focusing on what’s important (abstraction).
4. Creating step-by-step instructions to solve the problem (algorithms).

What is Computational Thinking

- Problem Solving?
  - Computational thinking is a set of skills and processes that guide problem solving.
  - What makes this especially different from other problem-solving processes is that it, in the end, results in an algorithm, which is a series of steps a person or computer uses to perform a task or solve a problem.
  - Computational thinking is derived from the process computer scientists use to develop code and communicate with computers through algorithms

Significance of Computational Thinking Skills

- Computational thinking allows us to take a complex problem, understand what the problem is and develop possible solutions.
- Since there are more than one potential solution to a given problem evaluation of different solutions is also important

  e.g. sorting of given set of items according to alphabetical or numerical order

  e.g Bubble sort Algo ->

  https://www.youtube.com/watch?v=9I2oOAr2okY
- We can then present these solutions in a way that a computer, a human, or both, can understand.

Significance of Computational Thinking Skills ctd..

- Computational thinking is a problem-solving process that involves looking at possible solutions abstractly and algorithmically, in a series of ordered steps.
- Without the ability to problem-solve using computational thinking, it would be difficult to define and replicate innovative solutions to modern computing application

What is computational thinking? Ctd..

- How computer scientists think – how they reason and work through problems
- Computer science encompasses many sub-disciplines that support the goal of solving problems:
  - Computer theory areas: these are the heart and soul of computer science
    - Algorithms, Data structures
  - Computer systems areas
    - Hardware design, Operating systems, Networks
  - Computer software and applications
    - Software engineering, Programming languages, Databases, Artificial intelligence, Computer graphics

A major goal of this course is to help you develop your computational thinking and problem-solving skills

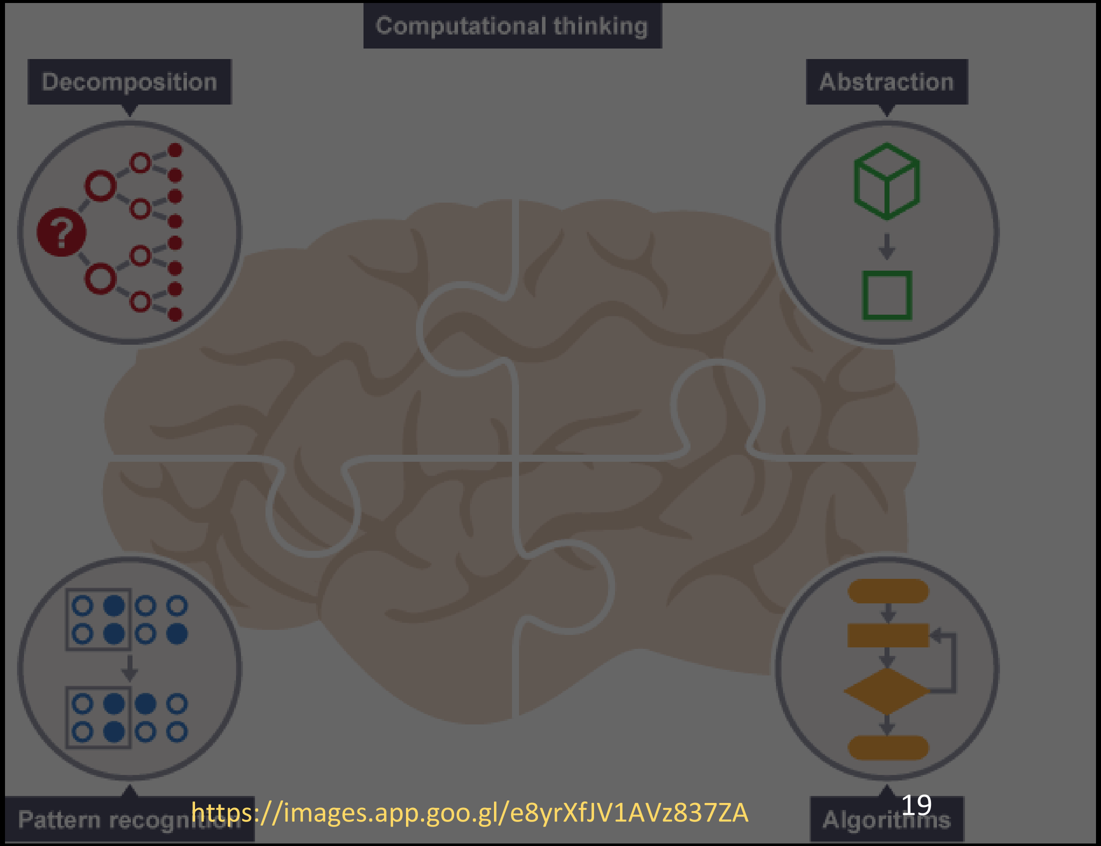
Computational Thinking is Coding?

- Not quite!. While computational thinking is the problem-solving process that can lead to code, coding is the process of programming different digital tools with algorithms.
- Coding is a means to apply solutions developed through the processes of computational thinking.
- Algorithms, in the case of coding, are a series of logic-based steps that communicate with digital tools and help them execute different actions.
- However, computational thinking results in algorithms for both computers and people, making it much more broadly applied with and without technology.
- At its core, the steps of the computational thinking process enable people to tackle large and small problems

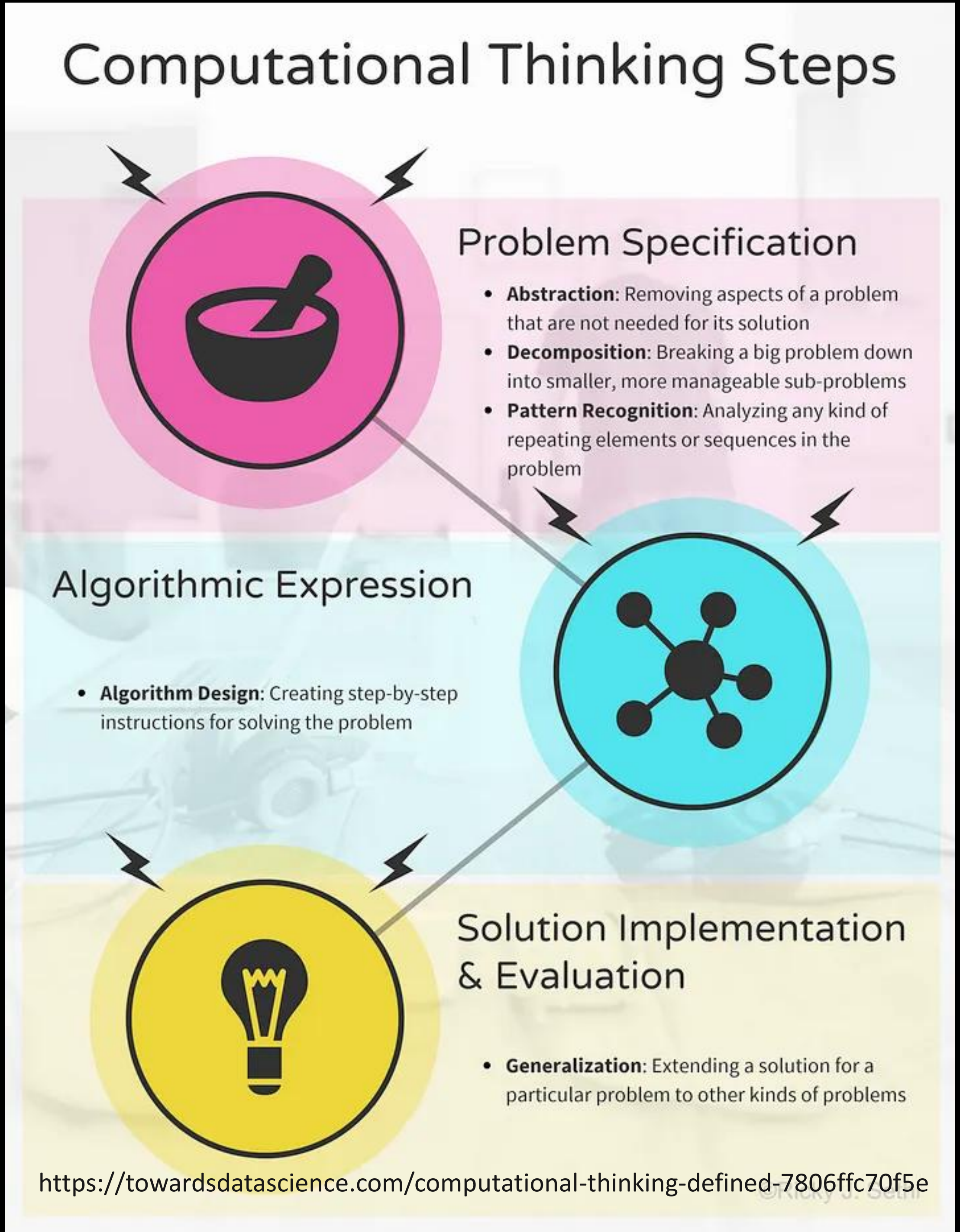
What is Computational Thinking?

- A way of thinking for logically and methodically solving problems
  - –E.g., purposeful, describable, replicable

- Requires skills such as
  - Decomposition
  - Pattern Recognition
  - Abstraction
  - Generalization
  - Algorithm Design
  - Evaluation



https://images.app.goo.gl/e8yrXfJV1AVz837ZA



Computational Thinking Steps

Problem Specification

- **Abstraction**: Removing aspects of a problem that are not needed for its solution
- **Decomposition**: Breaking a big problem down into smaller, more manageable sub-problems
- **Pattern Recognition**: Analyzing any kind of repeating elements or sequences in the problem

Algorithmic Expression

- **Algorithm Design**: Creating step-by-step instructions for solving the problem

Solution Implementation & Evaluation

- **Generalization**: Extending a solution for a particular problem to other kinds of problems

https://towardsdatascience.com/computational-thinking-defined-7806ffc70f5e

Computational Thinking

1. Problem Specification: We start by analyzing the problem, stating it precisely, and establishing the criteria for the solution.

   A computational thinking approach to a solution often starts by breaking complex problems down into more familiar or manageable sub-problems, sometimes called problem decomposition, frequently using deductive or probabilistic reasoning.

   This can also involve the ideas of abstraction and pattern recognition.

Computational Thinking ctd..

2. Algorithmic Expression: We then need to find an algorithm, a precise sequence of steps, that solves the problem using appropriate data representations.

   This process uses inductive thinking and is needed for transferring a particular problem to a larger class of similar problems.

   This step is also sometimes called algorithmic thinking.

   We can further break it down into either imperative, like procedural or modular, and declarative, like functional, approaches to algorithmic solutions.

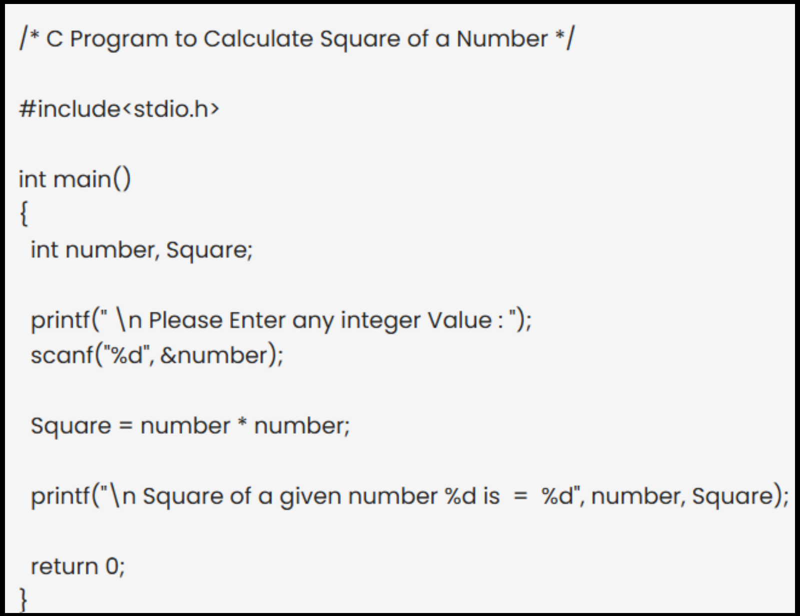
What is the difference between declarative and imperative programming?

- Declarative programming is often simpler and more efficient, while imperative programming is often more complex and requires more careful programming.
  - Declarative programming describes what you want the program to achieve rather than how it should run. In other words, within the declarative paradigm, you define the results you want a program to accomplish without describing its control flow. E.g. SQL
  - imperative paradigm, code describes a step-by-step process for a program's execution

- Ultimately, the decision to use declarative or imperative programming depends on the problem you are trying to solve and your specific programming needs.

Imperative Programming Paradigm?

- Imperative Programming (Tells how to do something)
- You give step-by-step instructions for what the computer should do.
  - Example (in C):



```
/* C Program to Calculate Square of a Number */

#include<stdio.h>

int main()
{
 int number, Square;

 printf(" \n Please Enter any integer Value : ");
 scanf("%d", &number);

 Square = number * number;

 printf("\n Square of a given number %d is  =  %d", number, Square);

 return 0;
}
```

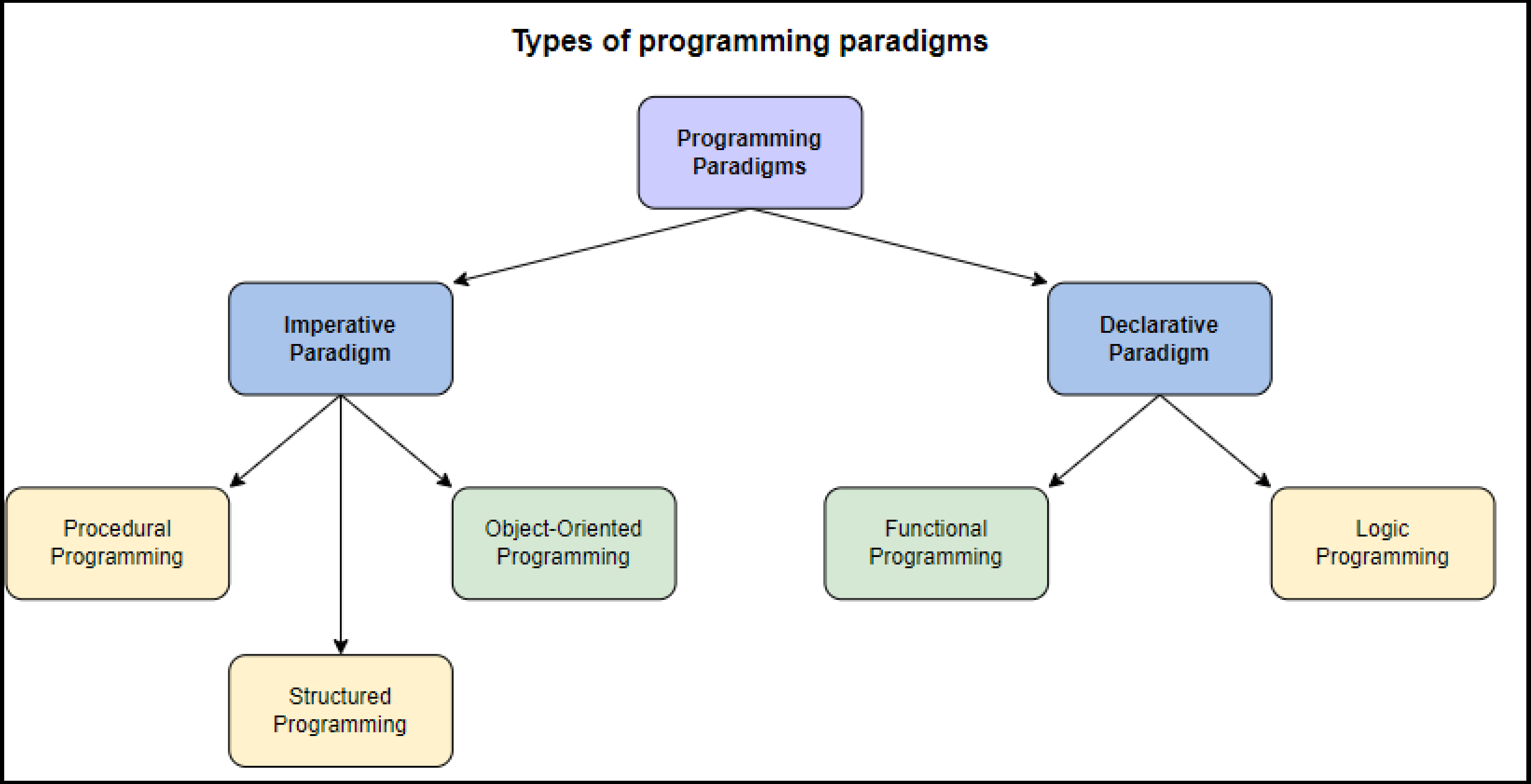
  - This says how to square each number: loop through the list, multiply each, and store it.

Declarative Programming Paradigm?

- Declarative Programming (Tells what you want)
- You describe what you want, not how to do it. The language or system figures out the "how.“
- Example (in SQL):
  - SELECT name FROM students WHERE grade > 90;
- This says *what* you want: the names of students with grades over 90. You do not write the steps to get them.

Types of Programming Languages

- imperative programming languages include:
- Java, C, Pascal
- declarative programming languages include SQL, Prolog
- Python supports both declarative and imperative programming



Computational Thinking ctd..

3. Solution Implementation & Evaluation: Finally, we create the actual solution and systematically evaluate it to determine its correctness and efficiency.

   This step also involves seeing if the solution can be generalized via automation or extension to other kinds of problems.

Computational Thinking and Problem Solving

- When we use computational thinking to solve a problem, what we are really doing is developing an algorithm: a step-by-step series of instructions.
- Whether it is a small task like scheduling meetings or a large task like mapping the planet, the ability to develop and describe algorithms is crucial to the problem-solving process based on computational thinking.

Details of the Computational Thinking Approach ctd..

1. Problem Specification

- Computational Thinking *Principles*: Model Development with Abstraction, Decomposition, and Pattern Recognition
- Computer Science *Concepts*: Problem Analysis and Specification
- Software Engineering *Concepts*: Problem Requirements Document, Problem Specifications Document, UML diagrams, etc.

https://towardsdatascience.com/computational-thinking-defined-7806ffc70f5e

Details of the Computational Thinking Approach ctd..

2. Algorithmic Expression
   - Computational Thinking *Principles*: Computational Problem Solving using Data Representation and **Algorithmic Design**
   - Computer Science *Concepts*: Data representation via some symbolic system and algorithmic development to systematically process information using modularity, flow control (including sequential, selection, and iteration), recursion, encapsulation, and parallel computing
   - Software Engineering *Concepts* : Flowcharts, Pseudocode, Data Flow Diagrams, State Diagrams, Class-responsibility-collaboration (CRC) cards for Class Diagrams, Use Cases for Sequence Diagrams, etc.

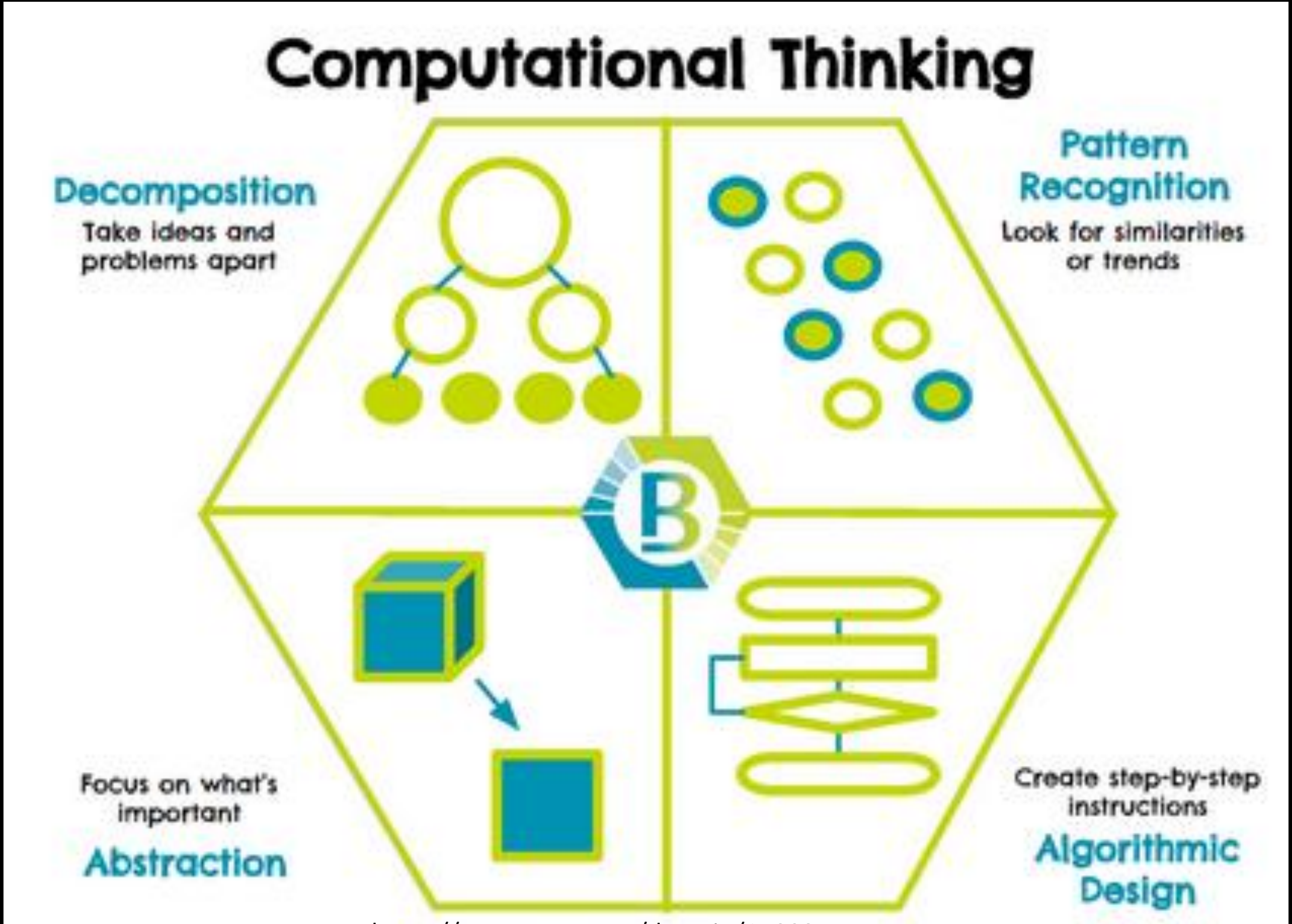
https://towardsdatascience.com/computational-thinking-defined-7806ffc70f5e

Details of the Computational Thinking Approach ctd..

3. **Solution Implementation & Evaluation**
   - Computational Thinking *Principles*: Systematic Testing and Generalization
   - Computer Science *Concepts*: Algorithm implementation with analysis of efficiency and performance constraints, debugging, testing for error detection, evaluation metrics to measure correctness of solution, and extending the computational solution to other kinds of problems
   - Software Engineering Concepts: Implementation in a Programming Language, Code Reviews, Refactoring, Test Suites using a tool like JUnit for Unit and System Testing, Quality Assurance (QA), etc.

https://towardsdatascience.com/computational-thinking-defined-7806ffc70f5e

Key Pillars of Computational Thinking


Computational Thinking
Decomposition
Take ideas and problems apart
Pattern Recognition
Look for similarities or trends
Focus on what's important
Abstraction
Create step-by-step instructions
Algorithmic Design

https://www.wcpss.net/domain/17003

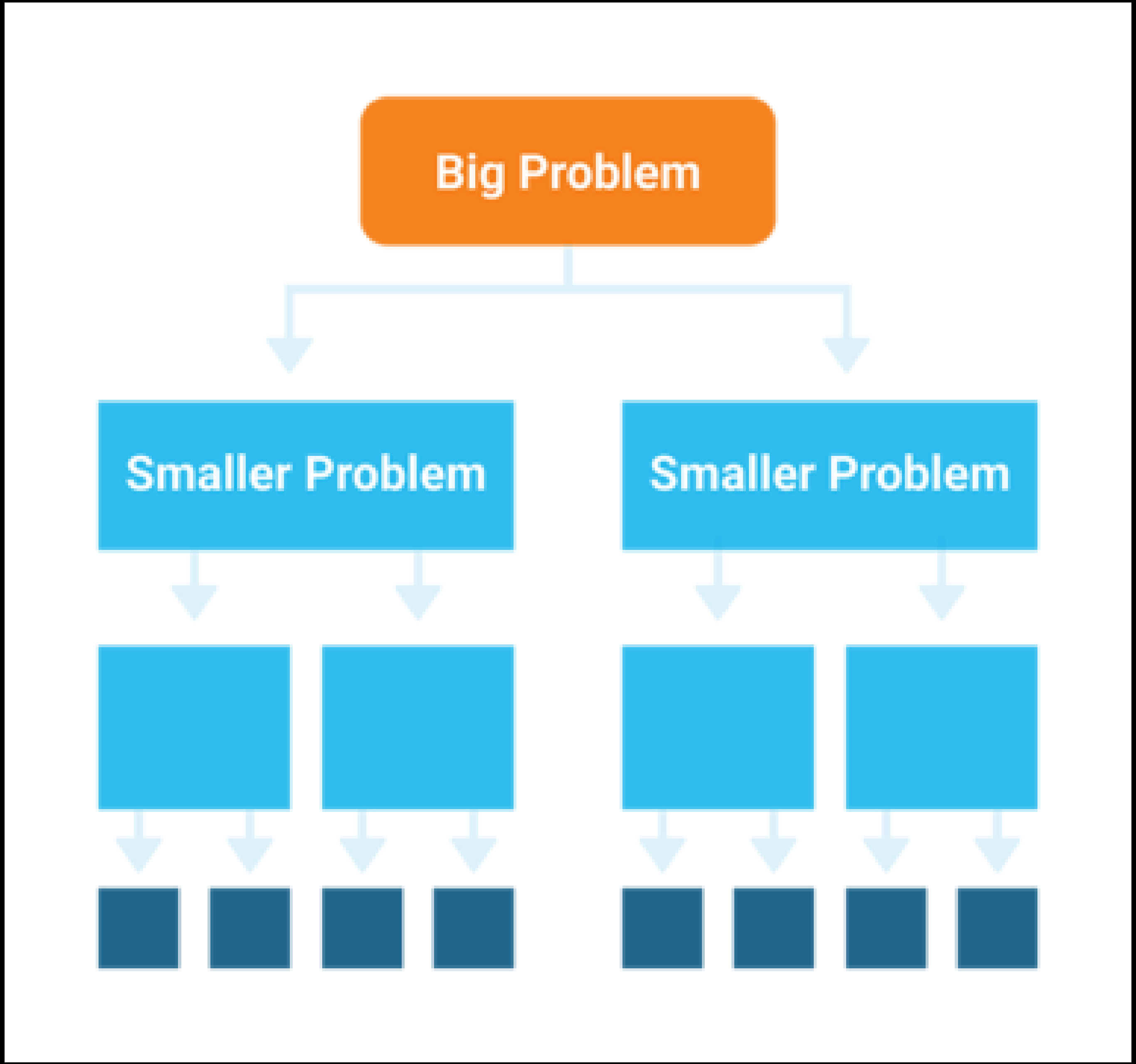
Computational Thinking

- Computational Thinking is an approach to problem solving with four key thinking processes;
  1. decomposition- taking ideas and problems apart, taking that complex problem and breaking it down into a series of small, more manageable problems
  2. pattern recognition- looking for similarities or trends, smaller problems can then be looked at individually, considering how similar problems have been solved previously
  3. abstraction- focusing on what's most important, focusing only on the important details, while ignoring irrelevant information and
  4. algorithm design- creating step-by-step instructions to solve a problem, simple steps or rules to solve each of the smaller problems can be designed.

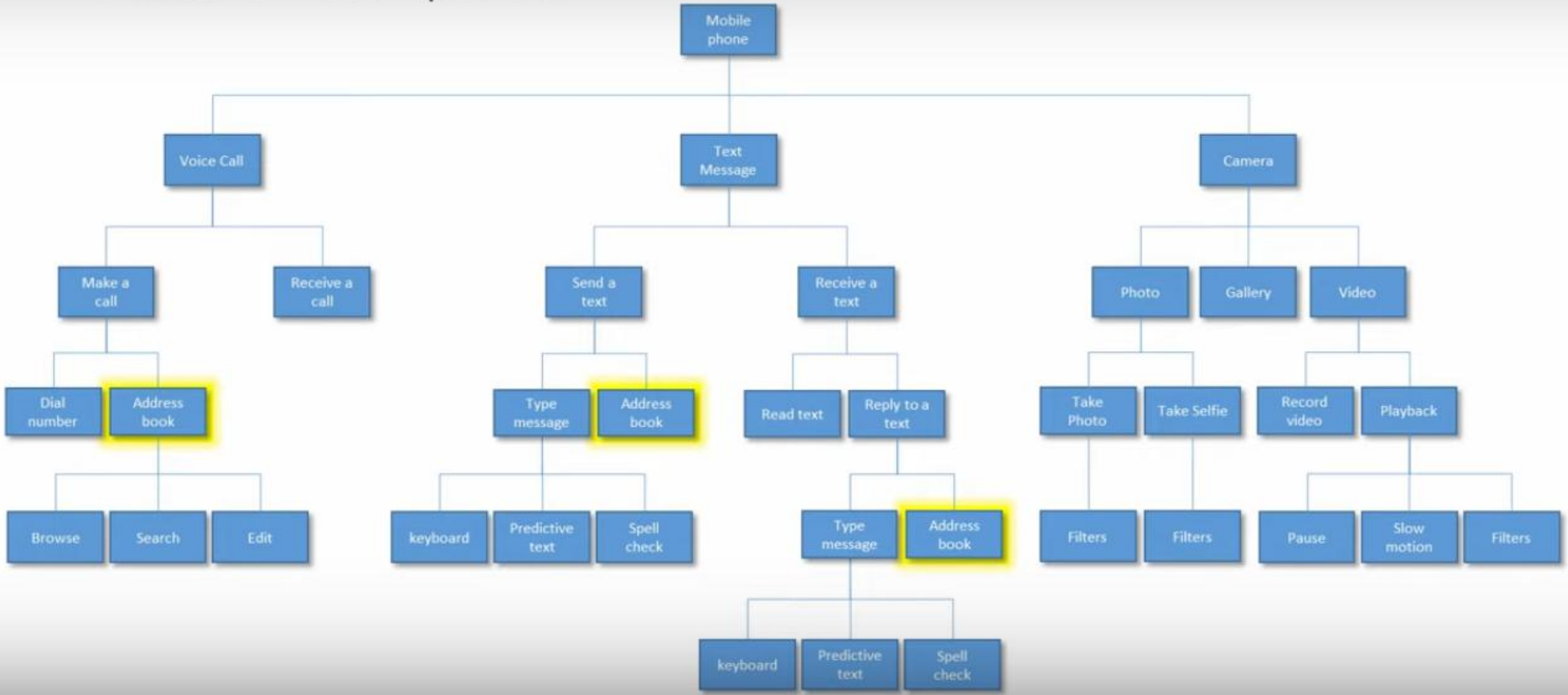
Decomposition

- It is a problem-solving approach that allows individuals to tackle complex tasks by dividing them into simpler subtasks.
- Process allows individuals can better understand complex problems and find efficient solutions.
- Enables to focus on addressing each component individually, making it easier to manage and solve the overall problem.
- phelps in identifying patterns and relationships among the smaller parts, leading to a deeper understanding of the problem as a whole.
- When faced with a complex problem, decomposition allows individuals to prioritize and allocate their time effectively.
- By dividing the problem into smaller parts, they can allocate time to address each subtask based on its importance and urgency

https://www.structural-learning.com/post/computational-thinking


Big Problem
Smaller Problem
Smaller Problem

Functional Decomposition



https://www.youtube.com/watch?v=dHWmnayy8MY&t=15s

Decomposition

- Another benefit of decomposition is the opportunity it provides for delegation and collaboration.
- Breaking down a complex problem into smaller parts enables individuals to distribute the workload among a team, improving efficiency and productivity.
- By breaking down complex problems into smaller, more manageable parts, individuals can develop a deeper understanding of the problem and approach it more effectively.
- Decomposition enhances critical thinking, time management, delegation, and collaboration skills, making it an essential skill for problem-solving in various domains

https://www.structural-learning.com/post/computational-thinking

Decomposition

- The smaller parts can then be examined and solved, or designed, individually, as they are simpler to work with.
- If a problem is not decomposed, it is much harder to solve. Dealing with a complex problem is much more difficult than breaking a problem down into a number of smaller problems and solving each one, one at a time.
- Smaller problems are easier to understand and can be examined in more detail.
  - For example, suppose that a crime has been committed. Solving a crime can be a very complex problem as there are many things to consider.

https://www.bbc.co.uk/bitesize/guides/zmhpfcw/revision/2

Decomposition

- A police officer would need to know the answer to a series of smaller problems:
    i. What crime was committed
    ii. When the crime was committed
    iii. Where the crime was committed
    iv. What evidence there is
    v. If there were any witnesses
    vi. If there have recently been any similar crimes

- The complex problem of the committed crime has now been broken down into simpler problems that can be examined individually, in detail.
- Once the individual information has been gathered and collated, the police officer may be able to solve the crime.

https://www.bbc.co.uk/bitesize/guides/zmhpfcw/revision/2

Decomposition

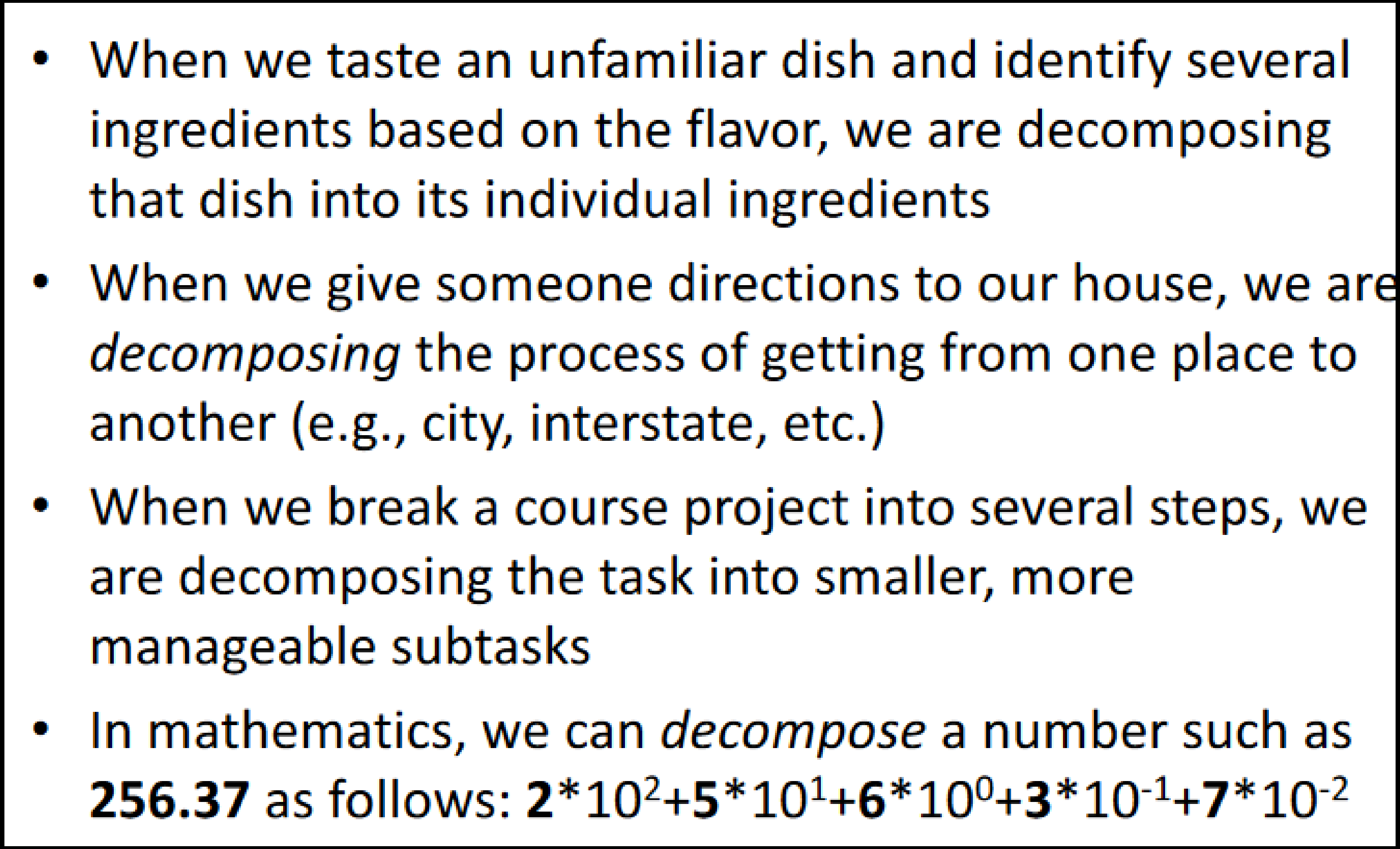
- Question: How might a programmer decompose the complex problem of how to create an app?
- The problem might decompose into these simpler problems:
  - what kind of app they want to create
  - what the app will look like
  - who the target audience for the app is
  - what the graphics will look like
  - what audio will be included
  - what software they will use to build the app
  - how the user will navigate the app
  - how they will test the app
  - where they will sell the app
- These smaller problems, solved individually, will help the programmer create an app.

https://www.bbc.co.uk/bitesize/guides/zmhpfcw/revision/2

Examples of Decomposition in Everyday Life

- Decomposition is something we inherently do in our daily lives, even if we don’t realize it.
- If you arrange a meal, you can use decomposition to select the menu, enlist support from others in the kitchen, task people with what to bring, determine the process by which to cook the different elements, and set the time for the meal

Examples of Decomposition

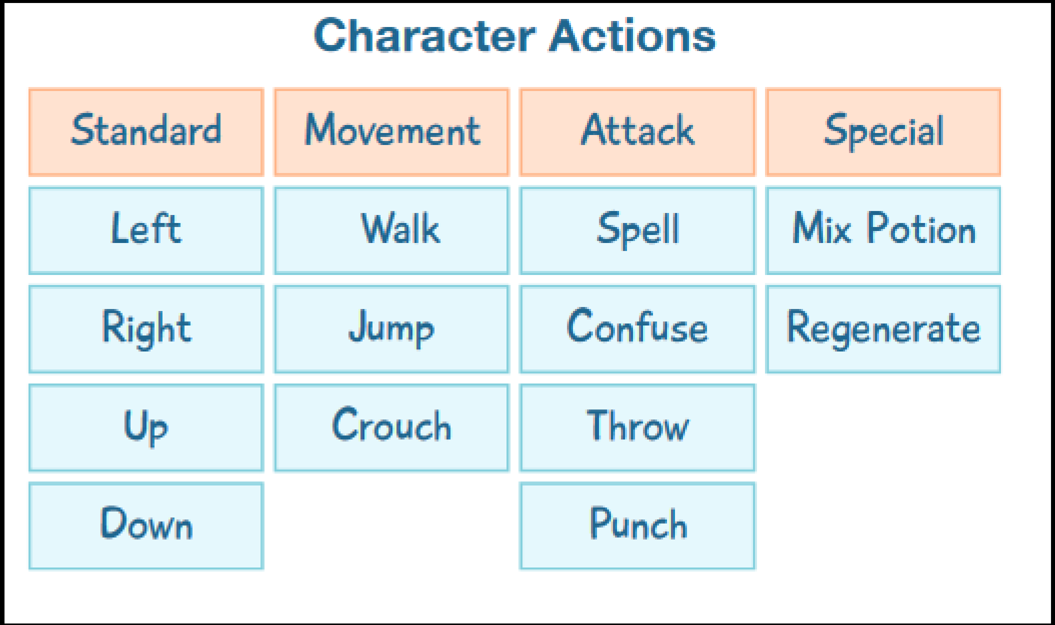


- When we taste an unfamiliar dish and identify several ingredients based on the flavor, we are decomposing that dish into its individual ingredients
- When we give someone directions to our house, we are *decomposing* the process of getting from one place to another (e.g., city, interstate, etc.)
- When we break a course project into several steps, we are decomposing the task into smaller, more manageable subtasks
- In mathematics, we can *decompose* a number such as **256.37** as follows: $\mathbf{2}*10^{2}+\mathbf{5}*10^{1}+\mathbf{6}*10^{0}+\mathbf{3}*10^{-1}+\mathbf{7}*10^{-2}$

https://cse.unl.edu/~lksoh/Classes/CSCE100_Fall20/handouts/01ComputationalThinkingCSCoding.pdf

Examples of Decomposition in Computer Science

- From a computer science and coding perspective, decomposition can come into play programming a new game.
- For example, need to consider the characters, setting, and plot, as well as consider how different actions will take place, how it will be deployed, and so much more



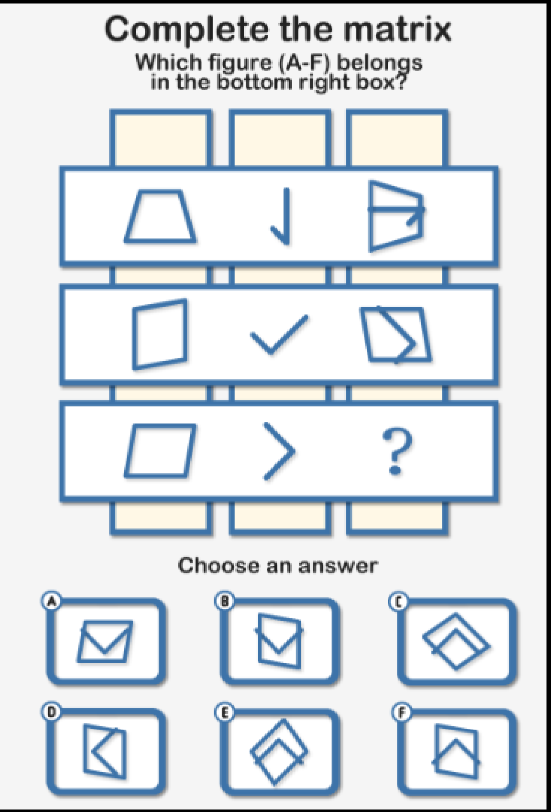
Character Actions

Standard	Movement	Attack	Special
Left	Walk	Spell	Mix Potion
Right	Jump	Confuse	Regenerate
Up	Crouch	Throw	
Down		Punch	

Pattern Recognition

- It involves the ability to identify similarities and differences in the details of a problem, allowing individuals to simplify complex problems by focusing on the underlying patterns.
- The ability to recognize patterns is vital because it helps individuals break down a problem into smaller, more manageable parts.
- By identifying similarities across different components of a problem, individuals can apply a single solution to multiple instances, saving time and effort.
- Similarly, recognizing differences between components helps individuals understand the unique aspects of each part and create specific solutions accordingly.

https://www.structural-learning.com/post/computational-thinking



- For example let us say we want to calculate the first 5 square numbers. This involves 5 sums:

- 1*1
- 2*2
- 3*3
- 4*4
- 5*5
- For each number we are doing the same thing: multiplying it by itself.
- We can easily take advantage of that fact to write a function, or use a for loop or similar construct.

- Sometimes it's harder to define a pattern.

For example let us say we want to calculate the first 5 triangular numbers. This involves 5 sums:

1
1+2
1+2+3
1+2+3+4
1+2+3+4+5
We can see a pattern here but it's not as simple as the square number example.
It is a pattern of two stages: first we need to create a list or sequence of the numbers, before we can sum them.
Now that we have identified the pattern, we can try to construct some code to automate/simplify its implementation.

Pattern Recognition

- All about recognizing patterns. Specifically, with computational thinking, pattern recognition occurs as different decomposed problems are studied.
- Through analysis, it is possible to recognize patterns or connections among the different pieces of the larger problem.
- These patterns can be both shared similarities and shared differences.
- This concept is essential to building understanding complex information and goes well beyond recognizing patterns amongst sequences of numbers, characters, or symbols

https://info.learning.com/hubfs

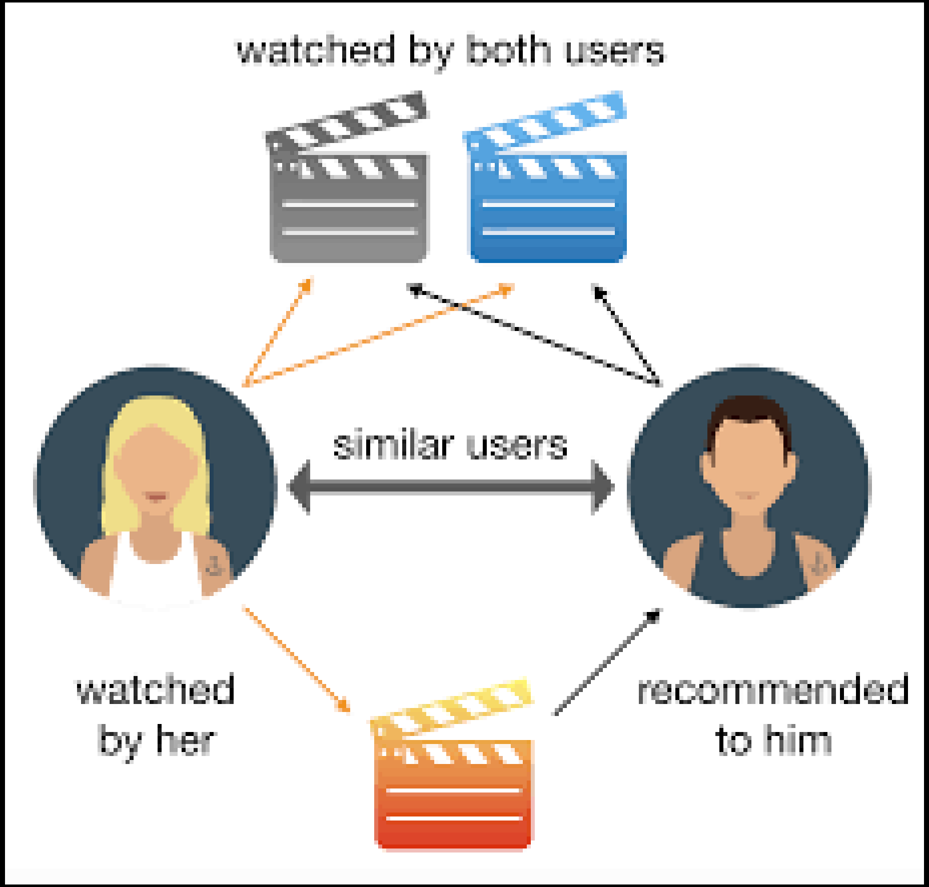
Examples of Pattern Recognition in Everyday Life

- Pattern recognition is the foundation of our knowledge.
- As infants, we used patterns to make sense of the world around us, to begin to respond verbally and grow our language skills, to develop behavioral responses, and to cultivate connections in this world.
- Beyond this, pattern recognition also occurs when scientists try to identify the cause of a disease outbreak by looking for similarities in the different cases to determine the source of the outbreak
- Question: Pattern Recognition in Medical and Financial Domains?

https://info.learning.com/hubfs

Examples of Pattern Recognition in Everyday Life

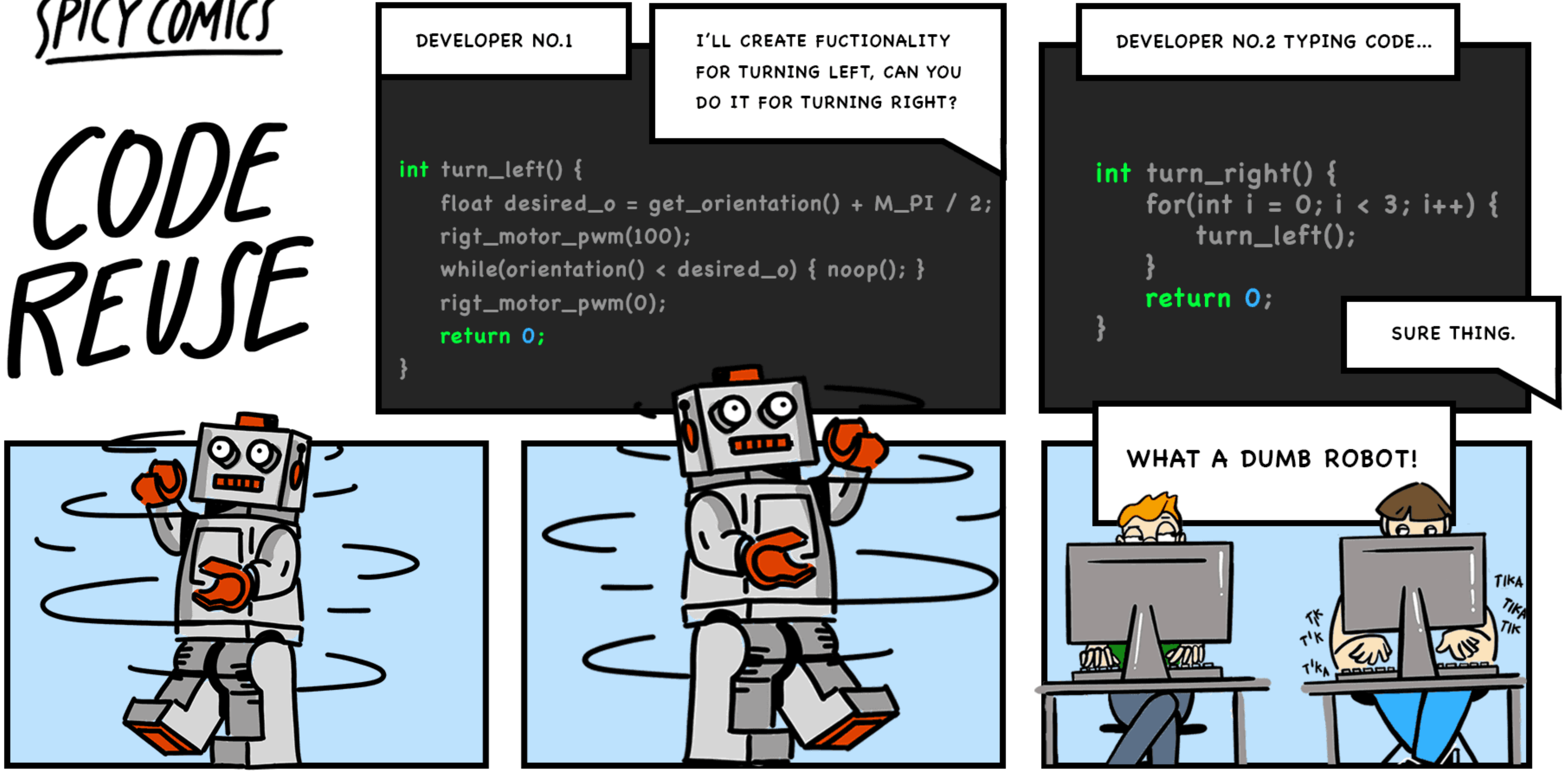
- When Netflix recommends shows based on your interests or a chat bot annoys you on a website, the technology (e.g Artificial Intelligence and Machine Learning) relies on pattern recognition.
- Movie Recommendation Systems, Clickstream Analysis on browser behavior
- Animals can be classified based on their Characteristics



https://rpubs.com/ezrasote/movie_recommendation

SPICY COMICS

CODE REUSE


DEVELOPER NO.1
I'LL CREATE FUCTIONALITY FOR TURNING LEFT, CAN YOU DO IT FOR TURNING RIGHT?
DEVELOPER NO.2 TYPING CODE...
SURE THING.
WHAT A DUMB ROBOT!
TIKA
TIKA
TIK
TIK
TIK
TIKA

```
int turn_left() {
    float desired_o = get_orientation() + M_PI / 2;
    rigt_motor_pwm(100);
    while(orientation() < desired_o) { noop(); }
    rigt_motor_pwm(0);
    return 0;
}
```

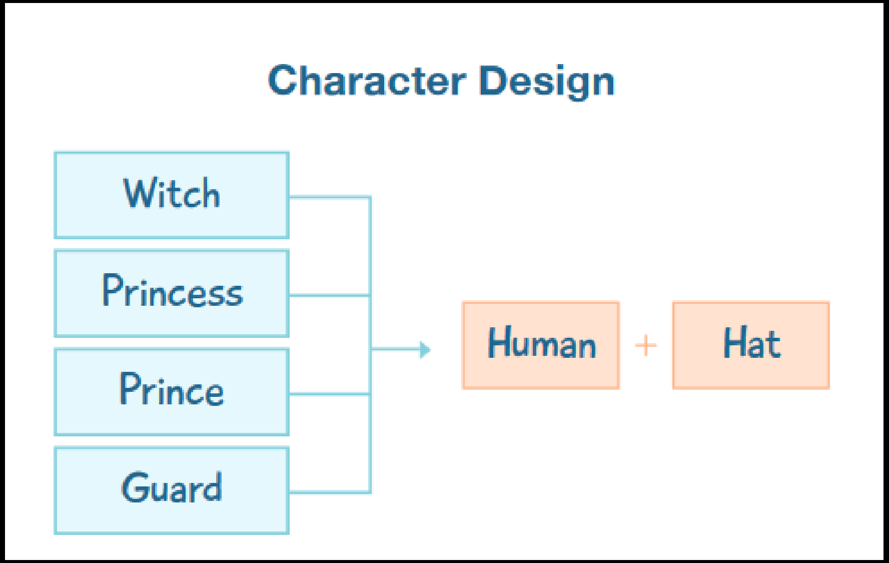
```
int turn_right() {
    for(int i = 0; i < 3; i++) {
        turn_left();
    }
    return 0;
}
```

spinning around it's axis

TEXT: STEFAN VUKANIĆ, SPFR   ILLUSTRATION: DRAGANA KRTINIĆ, SPFR   SPICY COMICS

Examples of Pattern Recognition in Computer Science

- In computer science, pattern recognition helps students identify similarities between decomposed problems. If they are coding a game, they may recognize similar objects, patterns, and actions.
- Finding these allows them to apply the same, or slightly modified, string of code to each, which makes their programming more efficient



Examples of Pattern Recognition

- Drivers look for patterns in traffic to decide whether and when to switch lanes
- People look for patterns in stock prices to decide when to buy and sell
- Scientists look for patterns in data to derive theories and models
- We look for patterns and learn from them to avoid repeating the same mistake

  example scenario in computing:

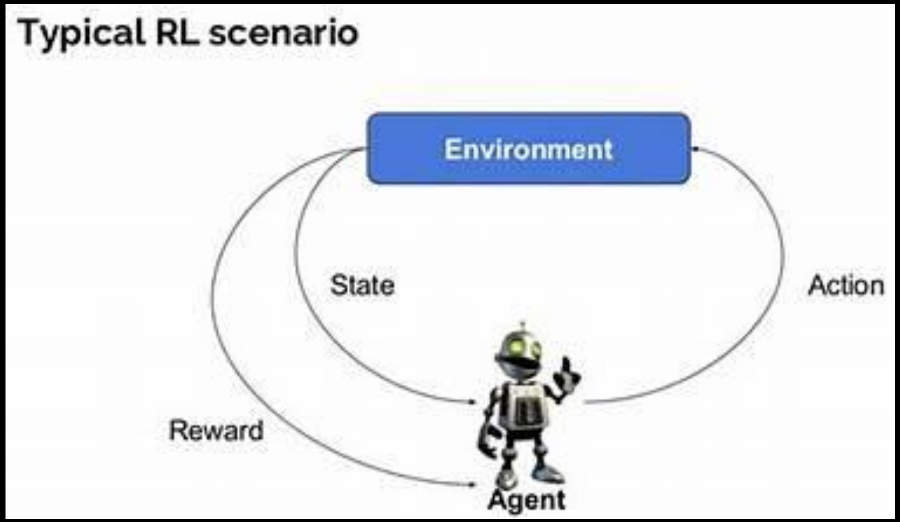
  (a) Certain classes of ML algorithms focus on looking for hidden patterns or clusters of features in the data

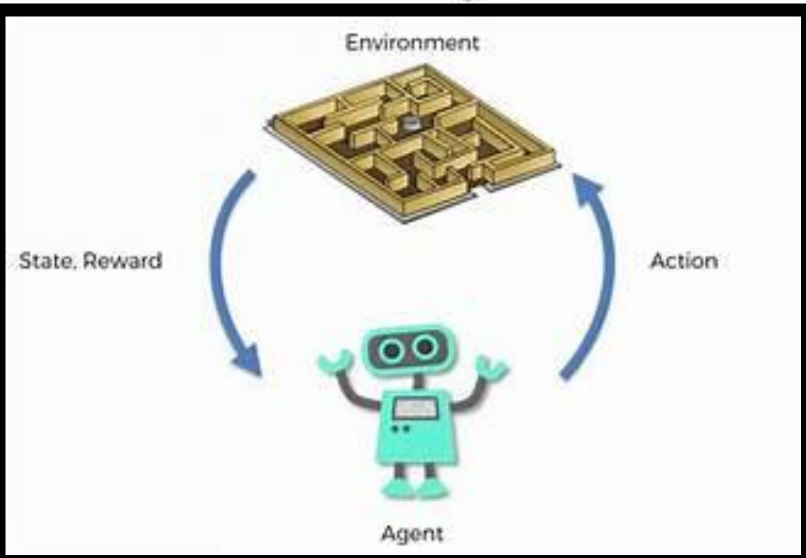
  (b) Reinforcement

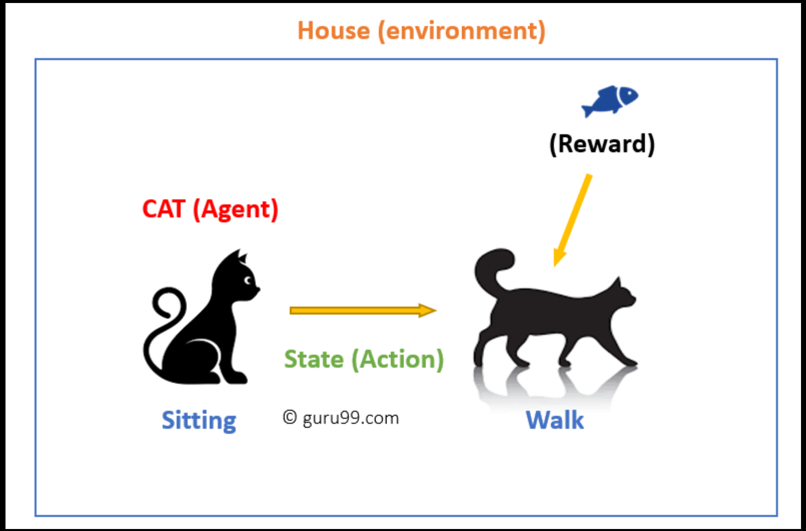
  Learning(Reward and Punishments)

  (**to be leaned in later modules in degree program)

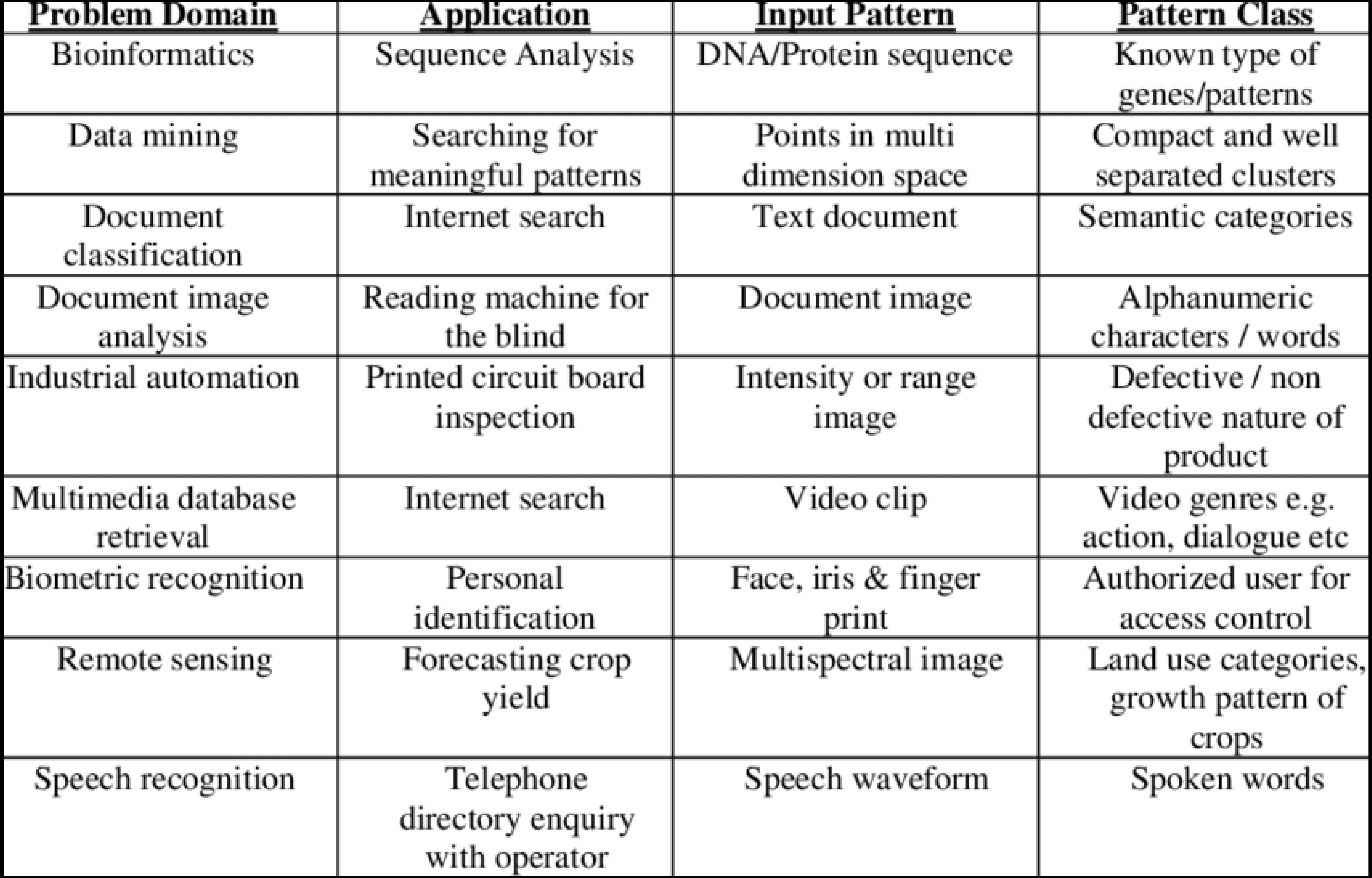
Reinforcement Learning Uses Patterns







Examples of Pattern Recognition

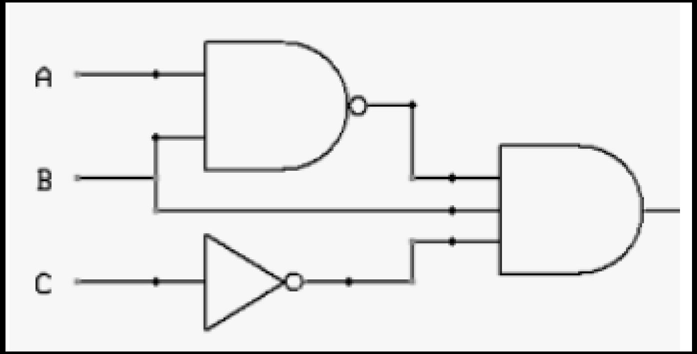


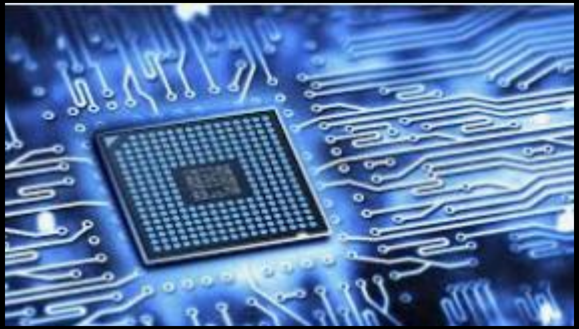
Problem Domain	Application	Input Pattern	Pattern Class
Bioinformatics	Sequence Analysis	DNA/Protein sequence	Known type of genes/patterns
Data mining	Searching for meaningful patterns	Points in multi dimension space	Compact and well separated clusters
Document classification	Internet search	Text document	Semantic categories
Document image analysis	Reading machine for the blind	Document image	Alphanumeric characters / words
Industrial automation	Printed circuit board inspection	Intensity or range image	Defective / non defective nature of product
Multimedia database retrieval	Internet search	Video clip	Video genres e.g. action, dialogue etc
Biometric recognition	Personal identification	Face, iris & finger print	Authorized user for access control
Remote sensing	Forecasting crop yield	Multispectral image	Land use categories, growth pattern of crops
Speech recognition	Telephone directory enquiry with operator	Speech waveform	Spoken words

Abstraction

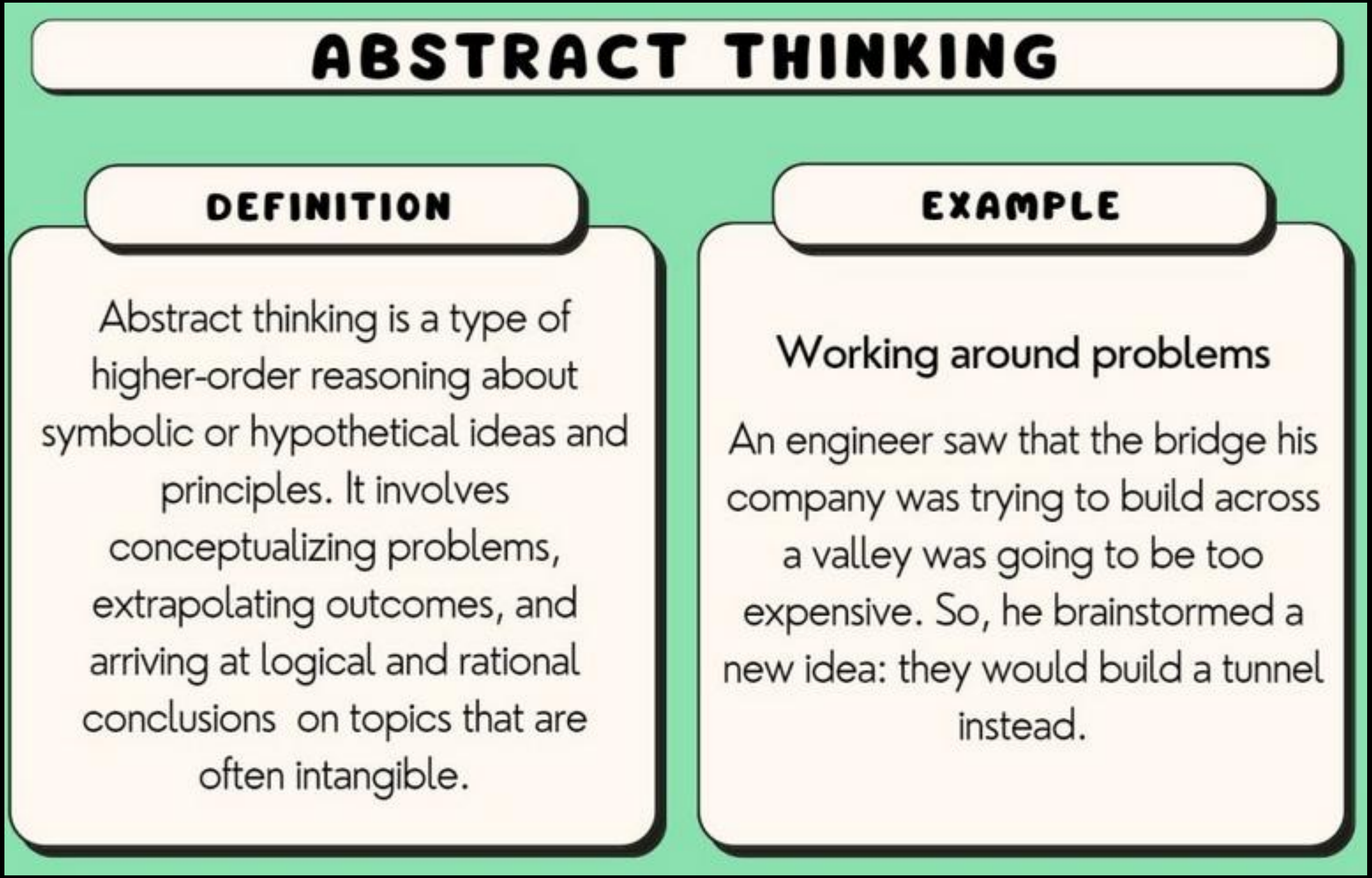
- Abstraction allows us to create a general idea of what the problem is and how to solve it.
- The process instructs us to remove all specific detail and any patterns that will not help us solve our problem.
- This helps us to form our idea of the problem.
- It allows individuals to focus on the essential aspects of a problem and disregard irrelevant details that may distract from finding a solution.

Examples of Abstraction





Abstract Thinking Example



ABSTRACT THINKING

DEFINITION

Abstract thinking is a type of higher-order reasoning about symbolic or hypothetical ideas and principles. It involves conceptualizing problems, extrapolating outcomes, and arriving at logical and rational conclusions on topics that are often intangible.

EXAMPLE

Working around problems

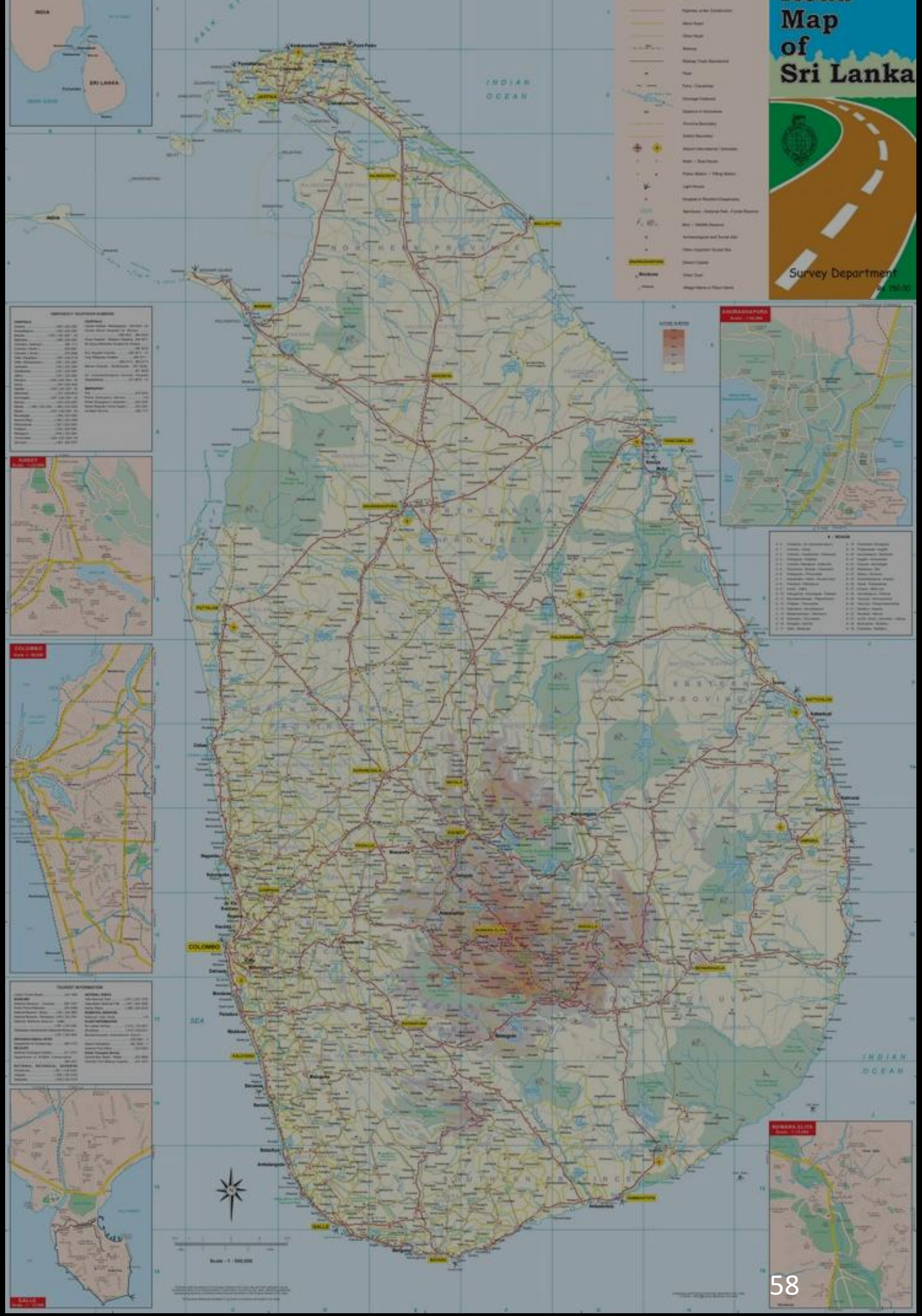
An engineer saw that the bridge his company was trying to build across a valley was going to be too expensive. So, he brainstormed a new idea: they would build a tunnel instead.

Example

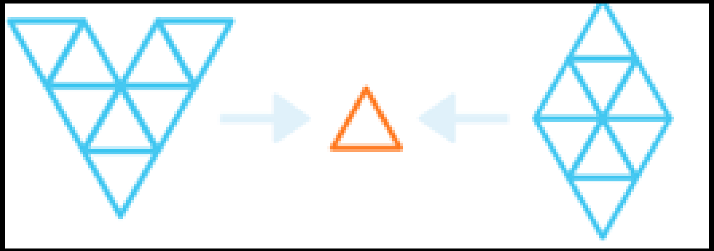
- Consider the problem of creating a program to calculate the **area of shapes**. The problem could first be decomposed into modules, each of which would be a particular shape, for example rectangle, square and triangle. Abstraction can then be followed for each module.
- For example, for the rectangle module the first step would be to notice that all rectangles share general characteristics:
  - a width
  - a height
  - area = width × height
- When abstracting, certain details are discarded but others are kept:
  - all rectangles have a width, but for the program design the actual rectangle width is not needed
  - all rectangles have a height, but for the program design the actual rectangle height is not needed
  - area is always width × height
- To solve this problem, all the program needs to be able to do is receive a width and a height, then calculate the area from those numbers. The actual numbers are irrelevant - they change with every rectangle - so they are discarded.

Abstraction

- An example of abstraction is the use of a road map. It details roads(Types A,B etc), Cities, rail tracks etc .
- Those information are sufficient for a person to plan a journey .
- Other details, such as real geographical location, width of roads, whether carpeted road etc are not included as they are irrelevant to journey planning



Abstraction



- Some algorithms focus on looking for hidden patterns or clusters of features in the data called, pattern generalization, abstraction enables us to navigate complexity and find relevance and clarity at scale.
- Decomposition and pattern recognition broke down the complex, and abstraction figures out how to work with the different parts efficiently and accurately.
- This process occurs through filtering out the extraneous and irrelevant in order to identify what's most important and connect each decomposed problem.
- Abstraction is similar to the selective filtering function in our brains that gates the neural signals with which we are constantly bombarded so we can make sense of our world and focus on what is essential to us.

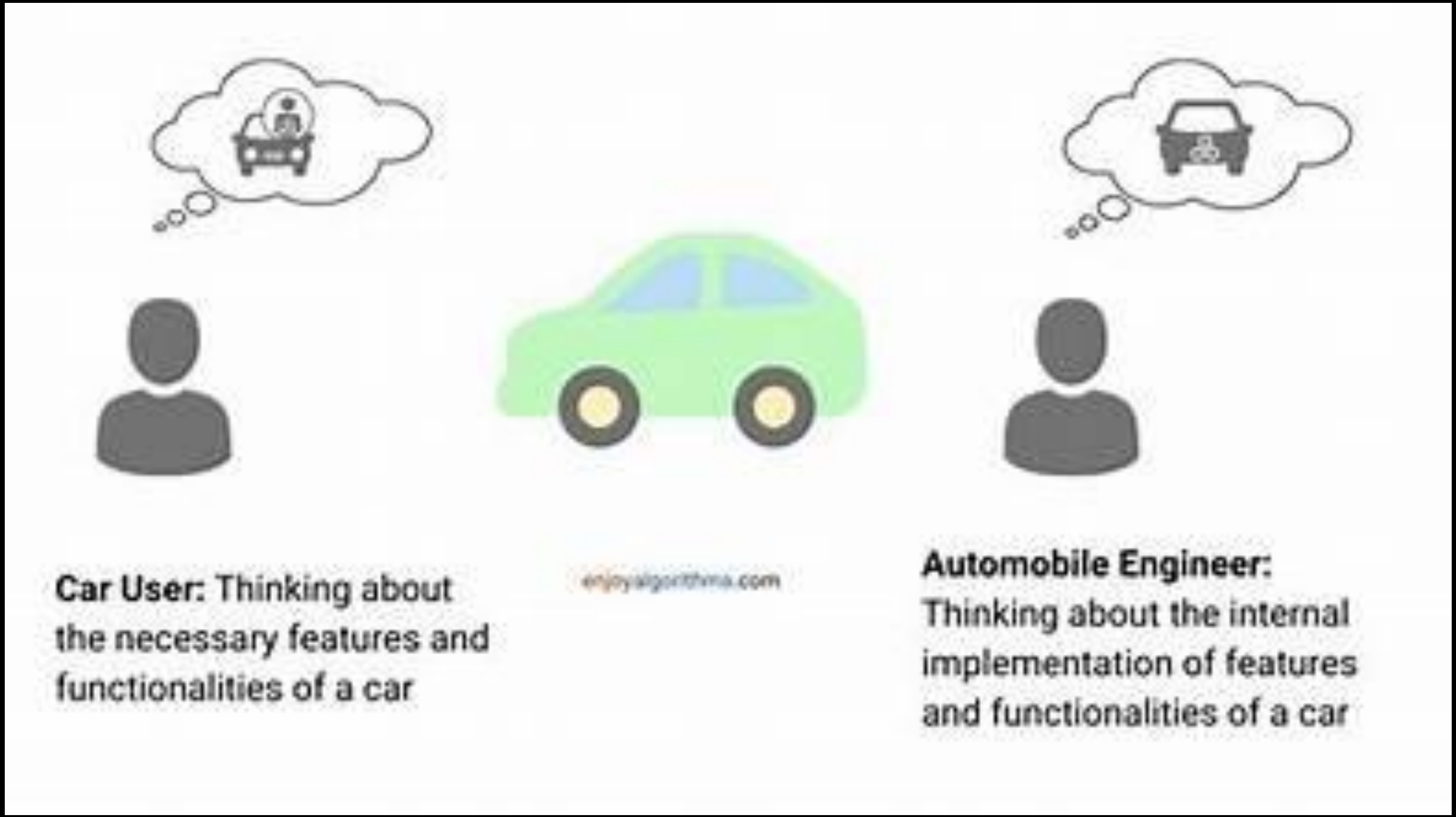
Examples of Abstraction in Everyday Life

- Another way to think about abstraction is in the context of those large/complex concepts that inform how we think about the world like Newton’s Laws of Motion, the Law of Supply and Demand, or the Pythagorean Theorem.
- All of these required the people behind them
    - to think about large, broad, and complex concepts;
    - to break down the problem and to experiment;
    - to find patterns amongst the experimentations; and
    - to eventually abstract this concrete knowledge to package it into these statements that free us from the complexity and difficulty went through to arrive at this law.

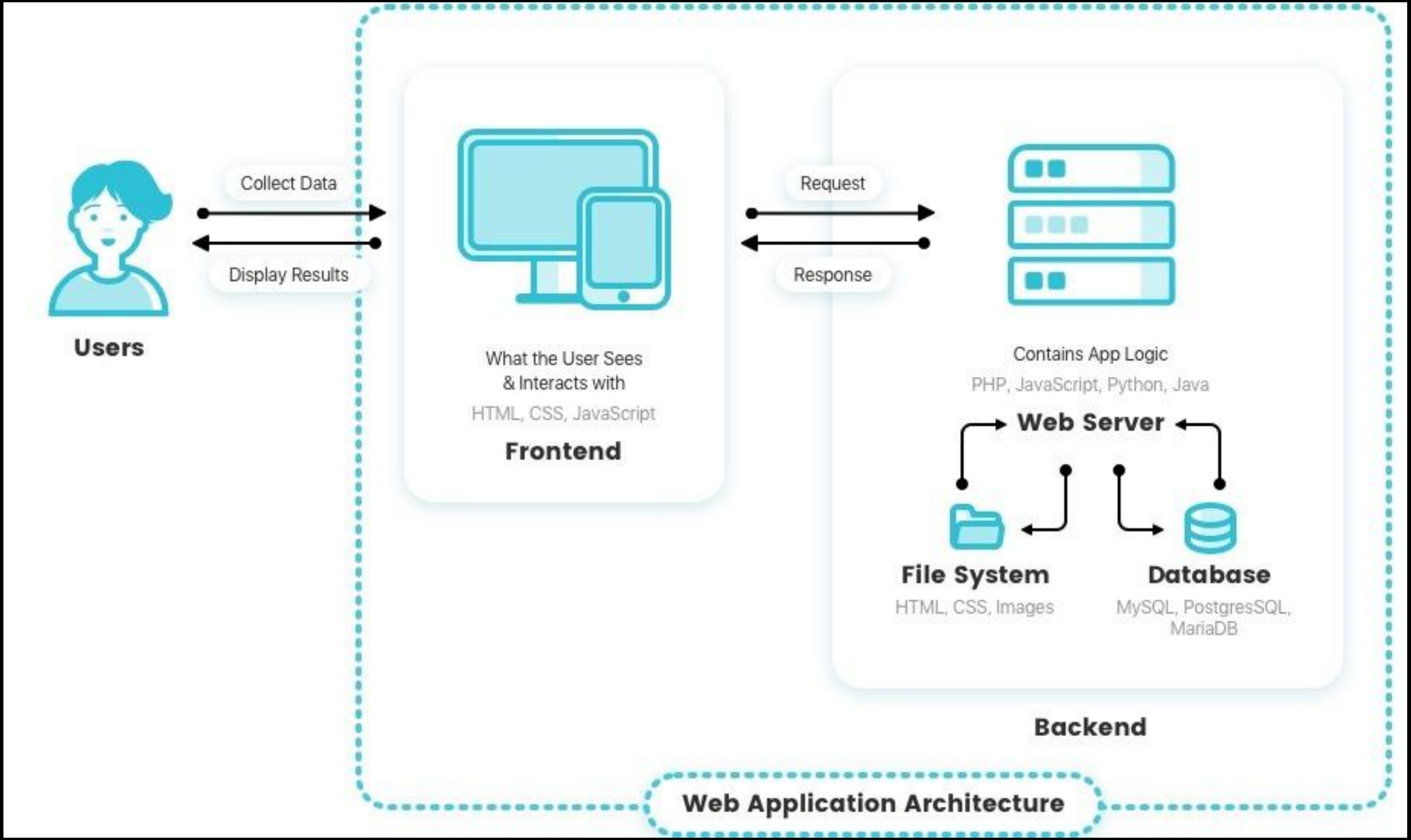
Computing & Abstraction

- Computing mostly operates independently of the concrete world.
- The hardware implements a model of computation that is interchangeable with others.
- The software is structured in architectures to enable humans to create the large/complex systems by concentrating on a few issues at a time.
- These architectures are made of specific choices of abstractions.

Abstraction

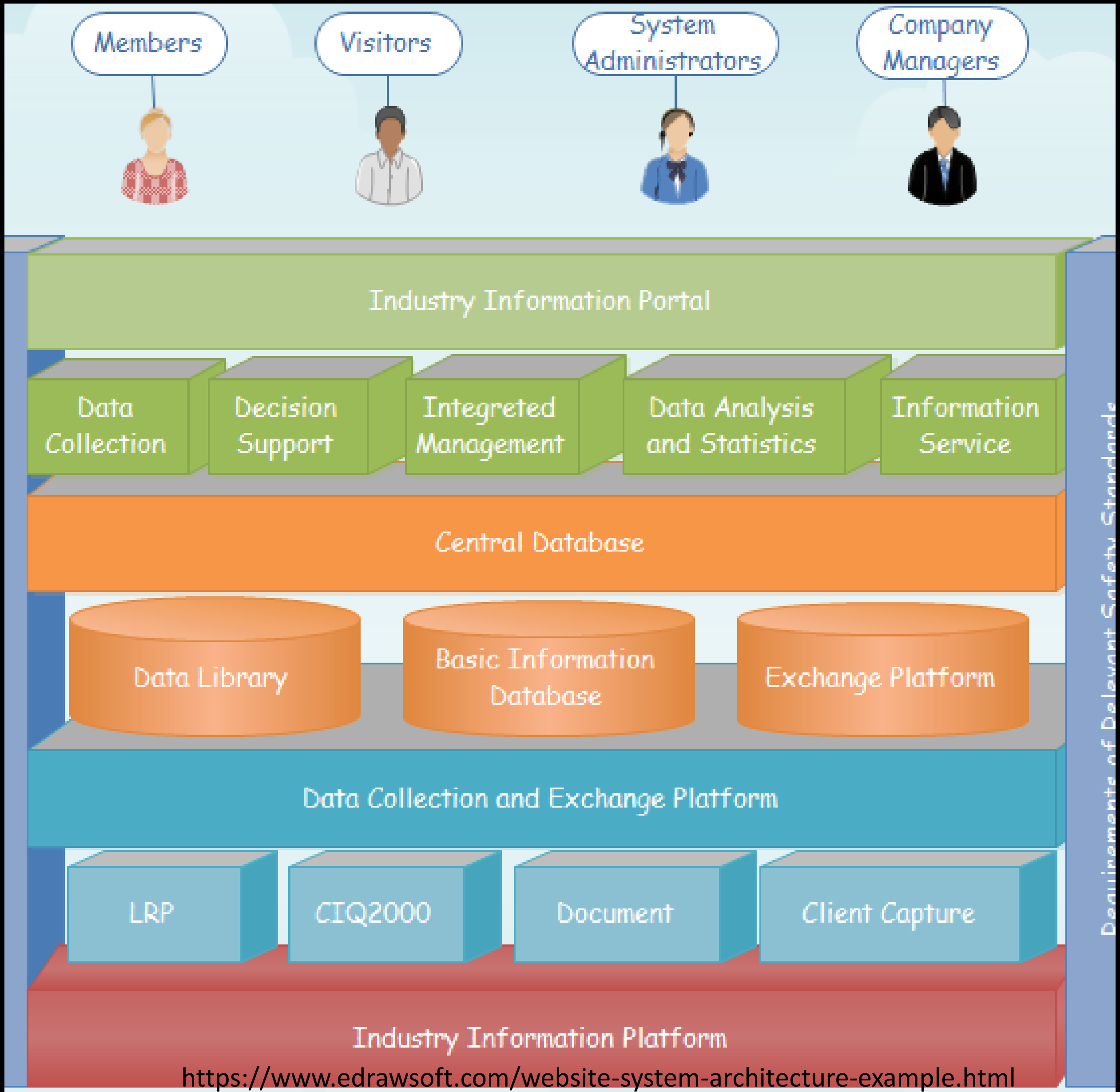

Car User: Thinking about the necessary features and functionalities of a car
enjoyalgorithms.com
Automobile Engineer: Thinking about the internal implementation of features and functionalities of a car

Example Web Application Architecture



Web Application Architecture: The Basics
https://www.intellectsoft.net/blog/wp-content/uploads/web-app-architecture-scheme.jpeg

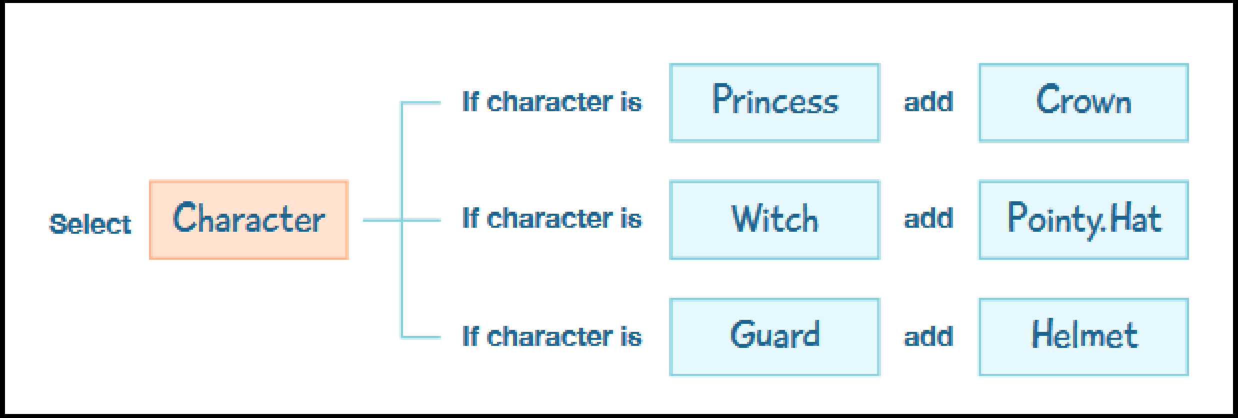
Website System Architecture Example



https://www.edrawsoft.com/website-system-architecture-example.html

Examples of Abstractions in Computer Science

- Abstraction in coding (writing programs) is used to simplify strings of code into different functions.
- It hides the underlying complexity in a programming language, which makes it simpler to implement algorithms and communicate with digital tools.

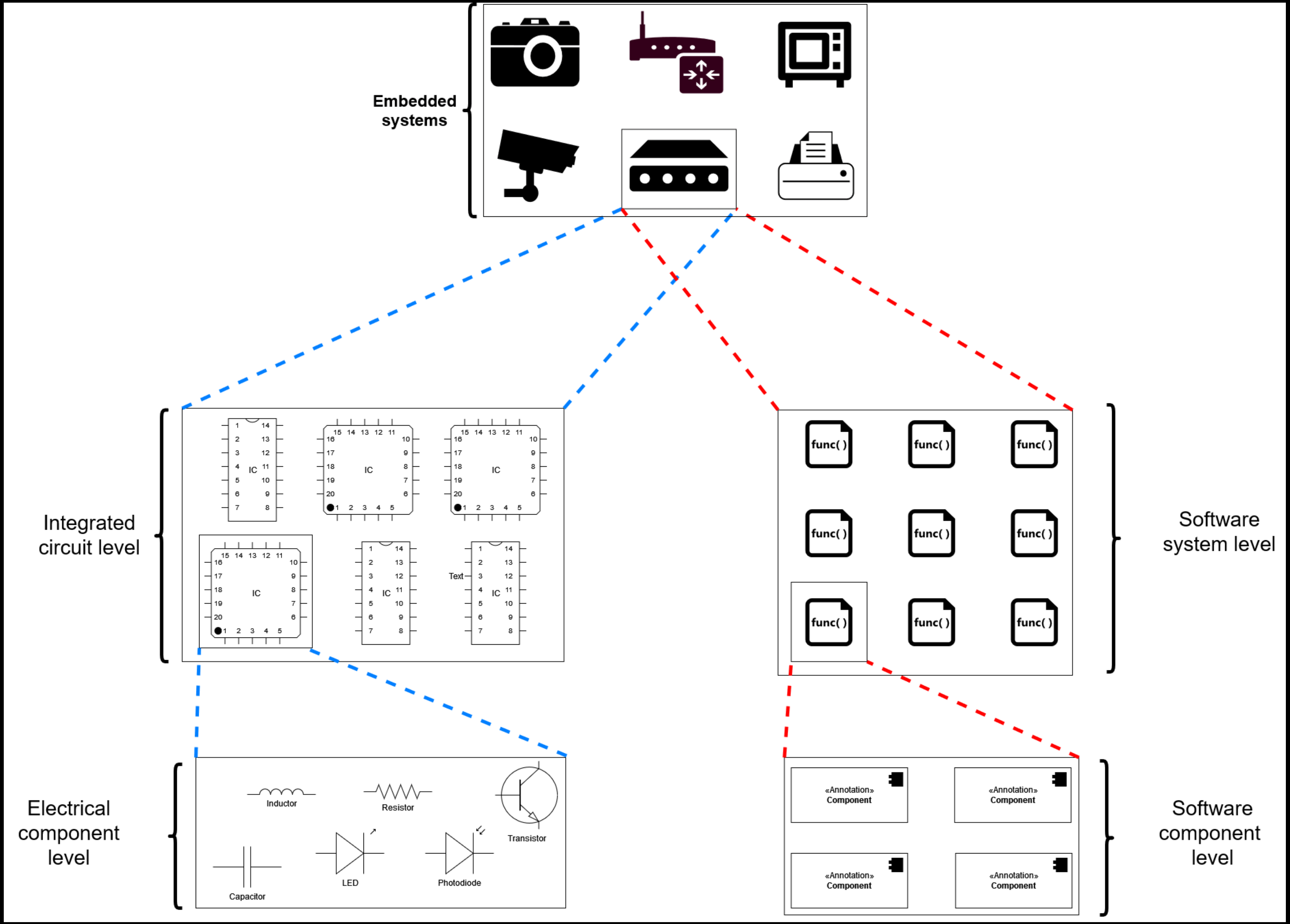


https://info.learning.com

Importance of Abstraction in Computing

- Understanding abstraction enables one to make sense of problems they encounter, helping them to not be surprised and confused in the face of something complex and to persist, compute, iterate, and imagine

Example of different levels of abstraction



https://www.freecodecamp.org/news/what-is-abstraction-in-programming/

Question

https://www.goodfon.com/rendering/wallpaper-art-avtomobili-avtodrom.html

- Imagine that a computer game is being designed to simulate cars on a race track. Abstraction has been used in the design.
- Explain how abstraction may be applied in the creation of the game.
  - The controls would most likely be overly simplified (e.g. accelerator, brake, and reverse simplified to forwards and backwards, gears likely removed)
  - The 3D models would likely be simplified polygonal models
  - The cars would likely not need re-fuelling – perhaps instead they have infinite fuel

https://starwort.github.io/computer-science/programming_project/lessons/computational_method_features.html

Algorithmic Thinking

- An algorithm is a set of instructions that helps solve a specific problem or accomplish a particular task. These instructions are typically presented in a clear and unambiguous manner, allowing individuals or computers to follow them precisely.
- Algorithmic Thinking is a fundamental concept within Computational Thinking that involves defining a step-by-step solution to a problem that can be replicated for a predictable outcome, whether by humans or computers.
- ** It is the process of breaking down a complex task into smaller, manageable steps and organizing them in a logical sequence.
- In Algorithmic Thinking, emphasis is placed on the design and structure of algorithms.

https://www.structural-learning.com/post/computational-thinking

Algorithmic Thinking

- Algorithmic thinking is a derivative of computer science and the process to develop code and build applications.
- This approach helps the problem-solving process by creating a series of systematic, logical steps that accept a defined set of inputs and produce a defined set of outputs based on these.
- In other words, algorithmic thinking is not solving for a specific answer; instead, it solves how to build a sequential, complete, and replicable process that has an end point – an algorithm. e.g Sorting, Merging, Searching(Google Search Algo)
- Designing an algorithm helps to both communicate and interpret clear instructions for a predictable, reliable output.

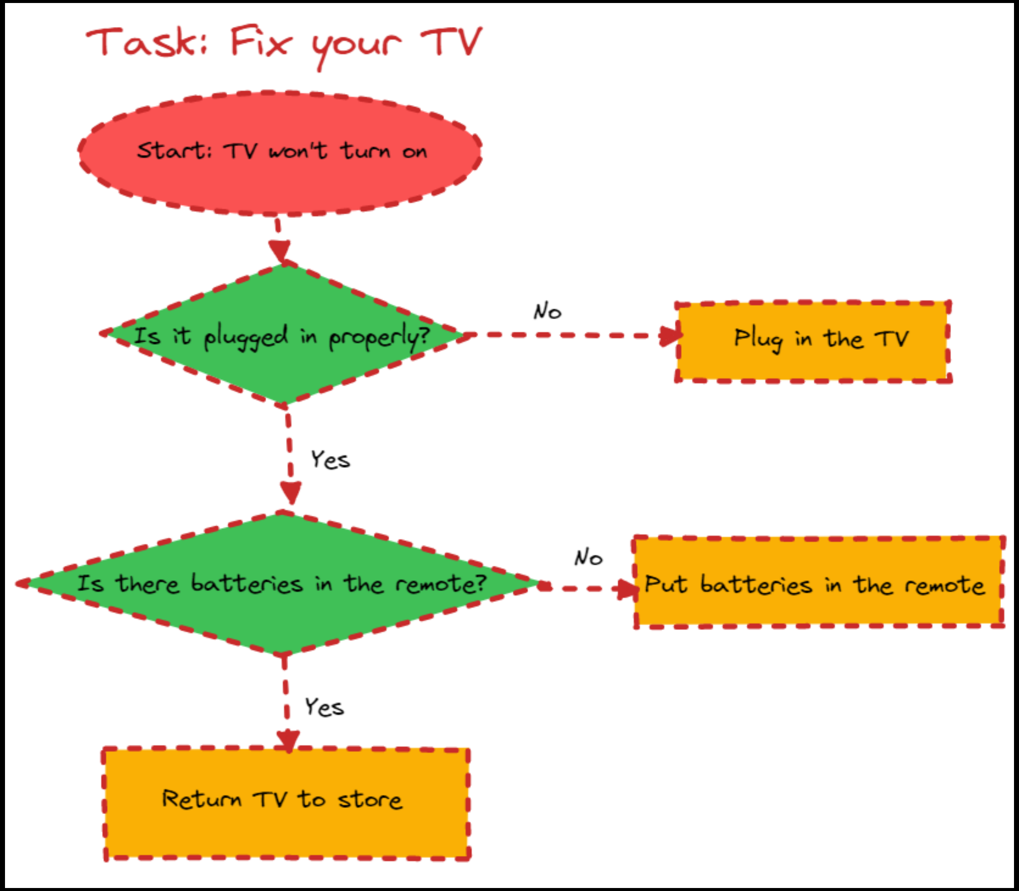
Algorithmic Thinking

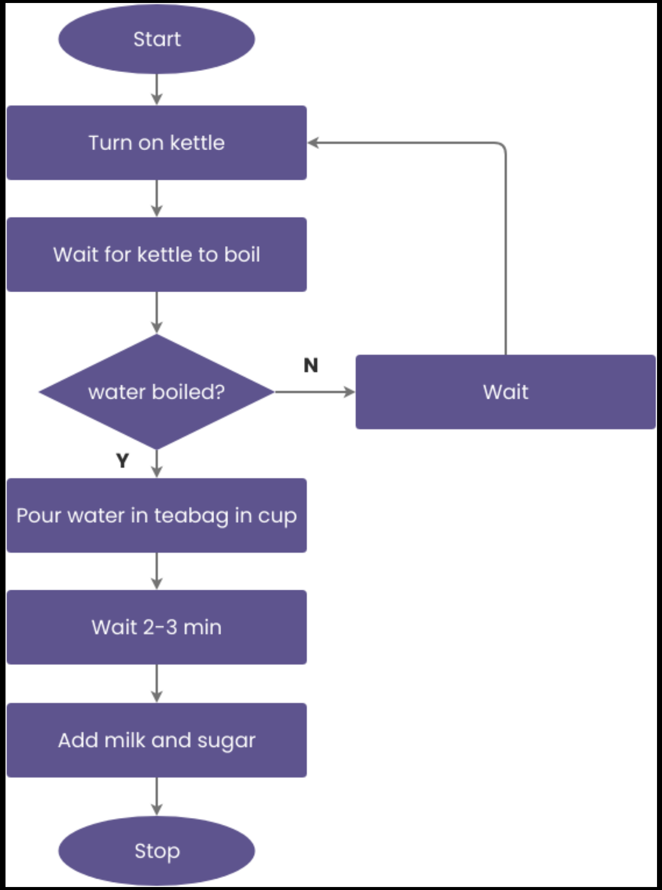
- The ability to think algorithmically is vital in the problem-solving process.
- It enables individuals to approach challenges systematically and methodically.
- By breaking down a problem into smaller steps, identifying patterns, and identifying the appropriate sequence of actions, algorithmic thinking helps to simplify complex problems.
- This structured approach enhances efficiency, accuracy, and effectiveness in finding solutions.
- Algorithm design is crucial in ensuring that the steps of the solution are well-defined, comprehensive, and optimized.
- A properly designed algorithm accounts for various scenarios, considering potential errors or exceptions and providing contingency plans.
- This systematic approach to algorithm design guarantees a more reliable and robust problem-solving process.

https://www.structural-learning.com/post/computational-thinking

Examples of Algorithms in Everyday Life

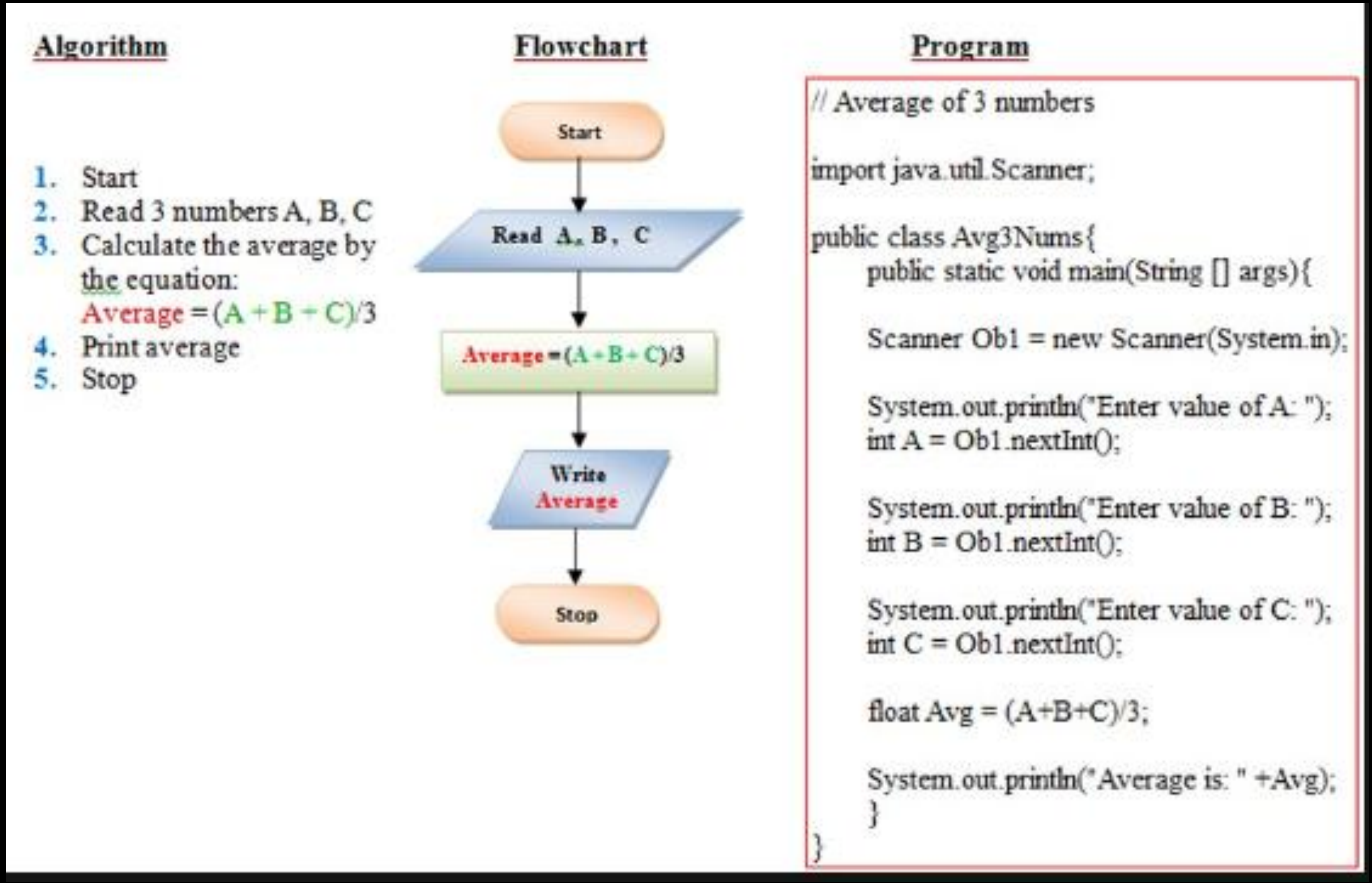
- Similar to Computational thinking and its other elements discussed, algorithms are something we experience quite regularly in our lives.


Task: Fix your TV
Start: TV won't turn on
Is it plugged in properly?
No
Plug in the TV
Yes
Is there batteries in the remote?
No
Put batteries in the remote
Yes
Return TV to store



https://storage.googleapis.com/algodailyrandomassets/curriculum/algorithm_tutorial/realworldexample2.png https://online.visual-paradigm.com/repository/images/42c6e525-7459-42b6-b6b9-6545f5aedfb5.png

Examples of Algorithms in Computations



Algorithm

1. Start
2. Read 3 numbers A, B, C
3. Calculate the average by the equation:
   Average = (A + B + C)/3
4. Print average
5. Stop

Flowchart

Program

```
// Average of 3 numbers

import java.util.Scanner;

public class Avg3Nums{
        public static void main(String [] args){

        Scanner Ob1 = new Scanner(System.in);

        System.out.println("Enter value of A: ");
        int A = Ob1.nextInt();

        System.out.println("Enter value of B: ");
        int B = Ob1.nextInt();

        System.out.println("Enter value of C: ");
        int C = Ob1.nextInt();

        float Avg = (A+B+C)/3;

        System.out.println("Average is: " +Avg);
        }
}
```

https://engineerstutor.com/wp-content/uploads/2018/12/6-1.jpg

Examples of Algorithms in Computer Science

- Algorithms used in coding are often complex.
- To contextualize algorithms in computer science and programming, following are a few examples.

Google and Algorithms

- How does search work in general ?
- The search process generally takes place in three stages:
  - Crawling. The search engine's algorithm directs web crawlers to discover URLs on the internet and examine their content. A crawler is a program that runs through content and automatically indexes it.
  - Indexing. The content contained in URLs is tagged with attributes and metadata(i.e data that provides information about other data) that help the search engine categorize the content.
  - Searching and ranking. The user enters a query, and the search engine ranks and returns content in relation to the query.

- *"Google's algorithms are complex mechanisms used to retrieve information from its search index and present the information to a given query. Algorithms examine billions of pieces of content in Google's index, looking for phrases and keywords that match the query."*

Ref: https://www.techtarget.com/whatis/feature/Google-algorithms-explained-Everything-you-need-to-know

Examples Computational Thinking

1. Automating Repetitive Tasks: A data analyst at a tech company used computational thinking to automate a repetitive task of cleaning and organizing large datasets. By breaking down the task into simple steps and writing a script in a programming language, the analyst was able to save hours of manual work each week.
2. Optimizing Resource Allocation: A logistics manager at a shipping company used computational thinking to optimize the allocation of trucks for deliveries. By abstracting the problem and using computational tools, the manager was able to find the most efficient routes, reducing fuel costs and delivery times.
3. Improving Customer Service: A customer service manager at a retail company used computational thinking to improve the company's response time to customer inquiries. By analyzing patterns in customer complaints and creating an algorithm to prioritize responses, the company was able to improve its customer satisfaction ratings.
4. Enhancing Product Design: A product designer at a software company used computational thinking to enhance the design of a new app. By using logical reasoning to understand user needs and preferences, the designer was able to create a more user-friendly interface.
5. Predicting Market Trends: A financial analyst at an investment firm used computational thinking to predict market trends. By using computational tools to analyze historical data and identify patterns, the analyst was able to make more accurate predictions about future market movements.

https://www.structural-learning.com/post/computational-thinking

How to Enhance your Algorithmic Thinking skills

- Explore Different Programming Paradigms(i.e Models/approaches)
- Expanding your knowledge of different programming paradigms can significantly enhance your algorithmic thinking by providing different perspectives on problem-solving.

- Paradigms to Explore:
  - Functional Programming: Learn languages like Haskell to understand immutability and pure functions.
  - Logic Programming: Try languages such as Prolog to experience declarative problem-solving.
  - Object-Oriented Programming: Learn and master deep into OOP principles Java, C++ etc.
- Each paradigm offers unique approaches to breaking down and solving problems, broadening your algorithmic toolkit.

Develop Abstract Thinking Skills

- Tackle Mathematical and Logical Puzzles
- Mathematical and logical puzzles are excellent for developing the abstract thinking skills crucial for algorithmic problem-solving.

- Types of Puzzles:
  - Sudoku and its variants
  - Logic grid puzzles
  - Mathematical word problems
  - Chess puzzles
- These puzzles train your brain to recognize patterns, think several steps ahead, and approach problems from multiple angles - all essential components of algorithmic thinking.

Develop Abstract Thinking Skills

- Key Concepts to Master:
    - Big O notation
    - Time complexity analysis
    - Space complexity analysis
    - Best, worst, and average case scenarios

    - Will be covered in upcoming lectures

How to Enhance your Algorithmic Thinking skills

- Practice Pseudocode and Flowcharting
- Algorithmic thinking is not about coding in a specific language; it is about logical problem-solving. Pseudocode and flowcharts are excellent tools for developing this skill.

- Benefits:
  - Focuses on logic rather than syntax
  - Improves ability to communicate ideas
  - Helps visualize problem-solving steps
- Exercise:
- Take everyday tasks (like making a tea or travelling to a city/building) and break down tasks into detailed pseudocode or flowcharts. This practice trains your brain to think in logical, step-by-step processes.

How to Enhance your Algorithmic Thinking skills

- Engage in Competitive Programming (Beyond LeetCode)
- Explore and use platforms for competitive programming that can enhance your algorithmic thinking.

- Explore:
  - **Leet Code**: provides coding and algorithmic problems for users to prepare for technical interviews and coding competitions.
  - **HackerRank**: Offers a wide range of programming challenges across various domains.
  - **CodeForces**: Known for its regular coding competitions and challenging problems.
  - **TopCoder**: Provides algorithmic challenges and hosts competitions with real-world applications.
- These platforms often provide diverse problem sets and can offer a fresh perspective on algorithmic challenges.

Questions?