**UNIT I PROBLEM SOLVING AND PROGRAMMING FUNDAMENTALS**

Computational thinking for Problem solving - Algorithmic thinking for Problem solving - Building Blocks - Problem Solving and Decomposition Dealing with Error - Evaluation. Overview of C - Data types - Identifiers - Variables - Storage Class Specifiers - Constants - Operators Expressions - Statements - Arrays and Strings - Single Dimensional - Two Dimensional Arrays - Arrays of Strings - Multidimensional Arrays.

**Introduction**

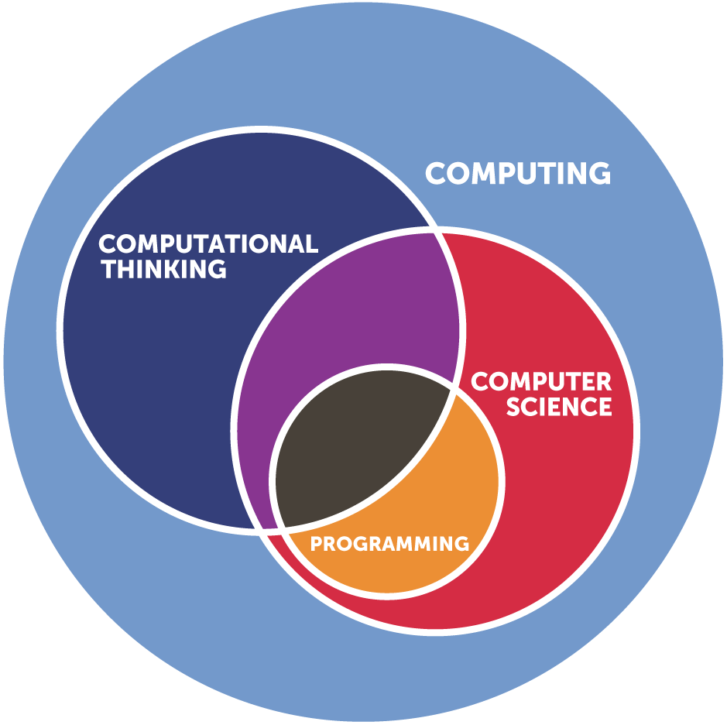
**We Use Computers to**

**C**ontrol our finances - Banking, Mutual Funds, and Share

Handle our social interactions - Facebook. LinkedIn, Whats App, Twitter

Get the latest news - Politics, Sports, Education

Arrange Travel – Tourism, Research, Migration, Pilgrimages, Business

**Computational Thinking in many diverse careers, such as:  
Natural Sciences:**  
Computational Biology  
Genomics  
Applied Physics  
Climate Change  
Astronomy.  
**Social Sciences:**  
Social Studies  
Population Analysis  
**Medicine:**  
Disease Analysis  
Medical Imaging  
Clinical Practice  
**Linguistics;**  
Law  
Music  
Teaching

**Computational Thinking - CT**

Computational thinking (CT) is the mental skill to apply concepts, methods, problem solving techniques, and logic reasoning, derived from computing and computer science, to solve problems in all areas.

**Computational Thinking** is a Problem-Solving Approach

**Computer Science – CS** is an individual academic discipline. Computer Science is the study of computers and computational systems. Computer scientists deal mostly with software and software systems; this includes their theory, design, development, and application.

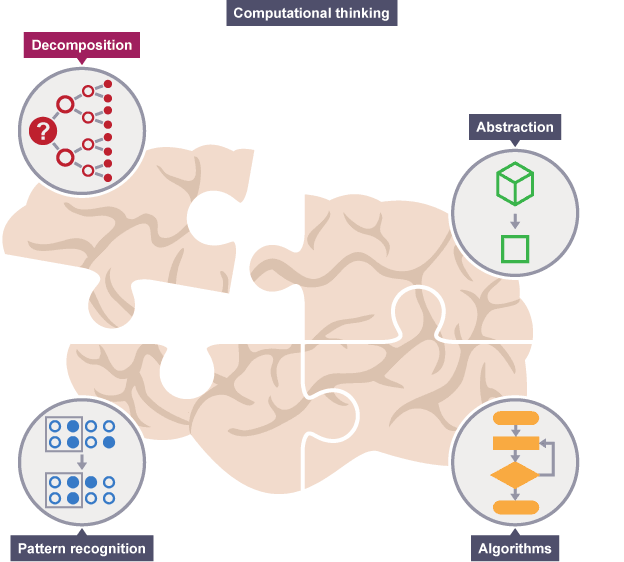
**A list of definitions of Computational Thinking**

Computational Thinking is the **Thought Processes** involved in **Formulating A Problem** and **Expressing Its Solution(S)** in such a way that a computer-human or machine-can effectively carry out.

The **Mental Activity** for **Abstracting Problems** and **Formulating Solutions** that can be **automated**.

The process of **Recognising Aspects of Computation** in the world that surrounds us, and **Applying Tools and Techniques** from Computer Science to understand and  
reason about both **Natural and Artificial Systems and Processes**.

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



**Core concepts of CT to be:**

* Logical Thinking
* Algorithmic Thinking
* Decomposition
* Generalisation and Pattern Recognition
* Modelling
* Abstraction
* Evaluation.

**Other peripheral concepts**

* Data Representation
* Critical Thinking
* Computer Science
* Automation
* Simulation / Visualisation.

**HOW IS COMPUTATIONAL THINKING USED?**

CT can be applied by anyone who is attempting to solve a problem and have a computer play a role in the solution.

**For example,**

**To a Computer scientist,** it means the study of algorithms and their application to different problems.

**To a mathematician,** it means carrying out long division factoring or doing carries in addition or subtraction.

**LOGICAL AND ALGORITHMIC THINKING**

Logic and algorithms are essential to CT both mathematical concepts in nature. Each has its own set of rules, procedures and definitions, which are very precise and systematic.

**Logical Thinking**

It introduces logical reasoning, along with Boolean and symbolic logic, and shows how to turn **intuitive** ideas into mathematically sound, logical expressions.

Logic is a system used for distinguishing between correct and incorrect arguments. Which means the philosophical idea of an argument; namely a chain of reasoning that ends up in a conclusion. Logic includes a set of principles that, when applied to arguments, allow us to demonstrate what is true.

In a sense, applying logic is a way of developing and testing a hypothesis. Using this way of thinking, applying logic assumes you already know at least some things for sure and allows you to use that knowledge to arrive at some further conclusions.

In a logical argument, each individual thing you already know (or assume) is called a **premise.** A premise is like any ordinary statement you or I might make, except that it can be evaluated to obtain an answer of **‘true’ or ‘false’**. A premise, therefore, has a **truth value**

**Inductive vs deductive arguments**

Inductive and deductive are commonly used in the context of logic, reasoning, and science. Scientists use both inductive and deductive reasoning as part of the [scientific method](https://www.dictionary.com/browse/scientific-method).

**Inductive** is used to describe reasoning that involves using **specific observations**, such as observed patterns, to make a general **conclusion.** This method is sometimes called **induction**. Induction starts with a set of [premises](https://www.dictionary.com/browse/premise), based mainly on experience or experimental evidence. It uses those premises to generalise a conclusion.

For example, let’s say you go to a cafe every day for a month, and every day, the same person comes at exactly 11 am and orders a coffee. The specific observation is that this person has come to the cafe at the same time and ordered the same thing every day during the period observed. A general conclusion drawn from these premises could be that this person always comes to the cafe at the same time and orders the same thing.

**Deductive** reasoning (also called deduction) involves starting from a set of **general** **premises** and then drawing a **specific conclusion** that contains no more information than the premises themselves. Deductive reasoning is sometimes called **deduction**

For example of *deductive* reasoning: chickens are birds; all birds lay eggs; therefore, chickens lay eggs. Another way to think of it: if something is true of a general class (birds), then it is true of the members of the class (chickens).

*Deductive* reasoning can go wrong, of course, when you start with incorrect premises. For example, look where this first incorrect statement leads us: all animals that lay eggs are birds; snakes lay eggs; therefore, snakes are birds.

## Difference between inductive and deductive reasoning

*Inductive* reasoning involves starting from specific premises and forming a general conclusion, while *deductive* reasoning involves using general premises to form a specific conclusion.

These aspects of reasoning are important in CT because computers are involved. The answer a computer gives is only as reliable as its reasoning, and since a computer is automating your reasoning, it’s your responsibility to make sure:

1. That reasoning is valid;

2. Giving the computer reliable input;

3. Know how to interpret what conclusion the computer reports, that is, is the result unquestionably true (the reasoning was deductive) or probably true (the reasoning was inductive)?

## Boolean logic

Boolean logic is a form of logic that deals with statements having one of only two values: true or false (usually). Different corresponding values could be used in other contexts: 1 or 0 for example, on or off, black or white

**Propositions**

Statements in Boolean logic are also known as propositions, which have several basic properties. First, a proposition can only have one value at any one time. In other words, a single proposition can’t be both true and false simultaneously. Second, propositions must have clear and unambiguous meaning. For example, a statement like: ‘It is travelling fast’, can certainly be evaluated as either true or false. However, it’s ambiguous as stated. If ‘it’ is a car travelling at 150 mph along the motorway, that’s certainly fast. But if ‘it’ is a spacecraft travelling towards Mars at 150 mph, that’s undoubtedly slow.

Third, it’s possible to combine individual propositions to make more complex ones (called compound propositions). For example, ‘Jenny is wearing the shirt and the shirt is red.’ This is helpful because we often want to evaluate several statements before reaching a conclusion. We make compound propositions by connecting single propositions together using logical operators.

**Logical operators**

AND: the technical name for this operator is conjunction. It chains propositions together in a way that all of them must be true for the conclusion to be true. If any of them are false, the conclusion is rendered false also.

OR: the technical name for this operator is disjunction. This operator chains propositions together in a way that at least one of them must be true for the conclusion to be true also. The only way that the conclusion is falsified is if all propositions are false.

NOT: the technical name for this operator is negation. This operator doesn’t chain propositions together itself, rather it modifies a single proposition. Specifically, it flips the truth value. Sometimes, negating a proposition can make it easier to express the chain of reasoning

IMPLIES: the technical name for this operator is **implication**. Using this operator is to state that there is a correlation between the two statements. If the first statement is true, then the second must be true also.

IF AND ONLY IF: the technical name for this operator is **biconditional**. This behaves very similarly to implication, but a biconditional means that the second proposition is influenced solely by the first. If the first is true, the second is true. If the first is false, the second is false. No exceptions.

**Symbolic logic**

**Table : Logical operators and their symbols**

|  |  |  |
| --- | --- | --- |
| **Operator Name** | **Symbol** | **Example** |
| **AND** | **∧** | A **∧** B |
| **OR** | **∨** | A **∨** B |
| **NOT** | ¬ | ¬A |
| **IMPLIES** | ⇒ | A ⇒ B |
| **IF AND ONLY IF** | ⇔ | A⇔B |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **P** | **Q** | **AND**  **∧** | **OR**  **∨** | **IMPLIES**  **⇒** | **IF AND ONLY IF**  **⇔** |
| **TRUE** | **TRUE** | **TRUE** | **TRUE** | **TRUE** | **TRUE** |
| **TRUE** | **FALSE** | **FALSE** | **TRUE** | **FALSE** | **FALSE** |
| **FALSE** | **TRUE** | **FALSE** | **TRUE** | **TRUE** | **FALSE** |
| **FALSE** | **FALSE** | **FALSE** | **FALSE** | **TRUE** | **TRUE** |

**ALGORITHMIC THINKING**

Logic gives you a set of rules that allow you to reason about some aspect of the world.  
That world does not have to be static. Logic can deal with things that are dynamic and  
continually changing.

**Algorithmic thinking** is a way of getting to a solution through the clear definition of the steps needed – nothing happens by magic.

**Algorithm: a sequence of clearly defined steps that describe a process to follow a  
finite set of unambiguous instructions with clear start and end points.**

Algorithms are a way of specifying a multi-step task, and are especially useful when  
we wish to explain to a third party (be it human or machine) how to carry out steps  
with extreme precision. As with logic, humans already have an intuitive understanding  
of algorithms. But, at the same time, a rich and precise science dictates exactly how  
algorithms work. Gaining a deeper understanding of this will improve your algorithmic  
thinking. This is important because a correct algorithm is the ultimate basis of any  
computer-based solution.

**Defining algorithms**

The definition of an algorithm is complex and involves several properties.

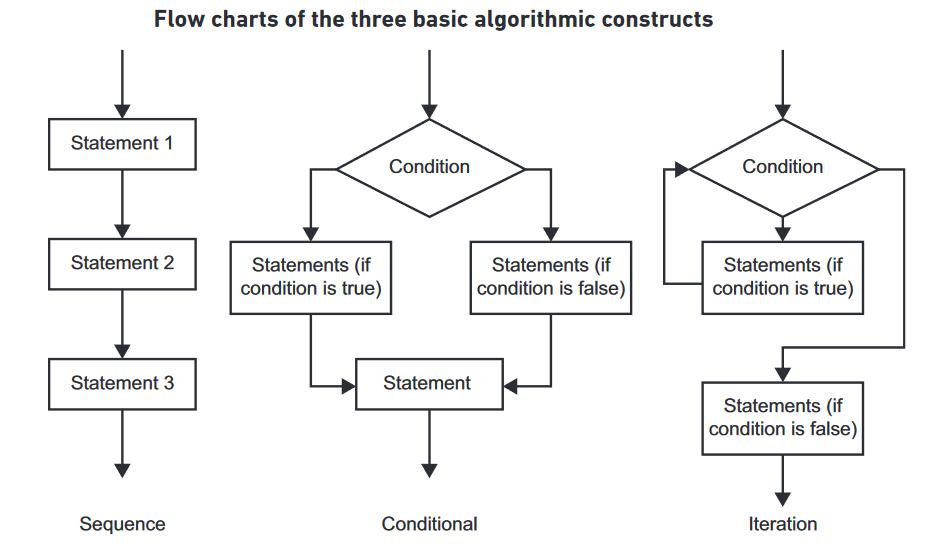
**Collection of individual steps**

The first property to mention just restates something said earlier: an algorithm is a  
collection of individual steps.

**Sequential**  
Algorithms are also sequential. The steps that make up the process must be carried  
out in the order specified. Failing to do this means that the result of executing the  
algorithm is likely incorrect.

**Conditionals:** a branching point where instructions are executed only if a certain  
condition is met.

**Iterations:** a sequence that is executed repeatedly so long as a certain condition  
holds.



**Definiteness**

Following on from that property is definiteness, meaning that every step must be  
precisely defined. Each step in an algorithm can have one and only one meaning, other-  
wise it is ambiguous

**State in algorithms**

State: the current configuration of all information kept track of by a program at any  
one instant in time.

**Facts about Computers**

1. A computer will do exactly as it is told. If it is told to do something impossible, it will crash.
2. A computer has no innate intelligence of its own. It will not do anything that it has not been instructed to do.
3. A computer has no common sense. It will not try to interpret your instructions in different ways. It will not make assumptions or fill in the obvious blanks in an incomplete algorithm.

**PROBLEM-SOLVING AND DECOMPOSITION**

A step-by-step procedure for problem-solving would be an obvious benefit. Unfortunately, problem-solving is partly a creative process. Like painting a landscape or writing a novel, it cannot be totally systematised, but strategies, heuristics and good practices exist to help you during your creative endeavours. These are things that previous generations of problem  
solvers found useful for attacking a problem’s complexity

**A systematic approach**

How to Solve It takes an approach to problem-solving inspired by the best traditions of  
mathematical and natural sciences.

1. Understand the problem
2. Devise a plan
3. Execute the plan
4. Review and extend

**DEFINING THE PROBLEM**

The hardest part of problem solving is characterising the problem. Problem-solving involves transforming an undesirable state of affairs (the start point)  
into a desirable state of affairs (the goal). The start point and the goal are intimately  
linked.

By examining the start point, we nail down exactly what is undesirable and why.  
In doing so, we reveal more about what your goal should be.

**Find things undesirable for any number of reasons.**

1. Maybe your current process is too slow, in which case your goal will involve  
   making measurable improvements to the process’s speed.
2. Maybe you regularly need to make certain decisions, but you have too much  
   data to handle. This implies your goal may be to somehow automate your  
   decision-making strategy or filter information before you analyse it.
3. Maybe you have missing information about how something behaves. This  
   suggests producing a model or simulation of it.

When trying to understand the problem, follow the advice given:

1. If someone else gave it to you, try restating the problem in your own words.
2. Try and represent the problem using pictures and diagrams. Humans deal  
   better with visual representations.
3. There will be knowns and unknowns at the start. You should ensure that  
   enough information is known for you to form a solution. If there isn’t, make the  
    unknowns explicit.

Important characteristics about the problem-solving process to keep in mind.

**Quality**  
First, notice that **A solution** and not **The solution**. For any problem, there are usually  
multiple solutions: some good, some terrible and others somewhere in-between.  
Should focus on finding the best solution.

**Collaboration**

Making problem-solving a collaborative effort is often helpful. Something as simple  
as explaining your current work out loud often helps you to spot mistakes or potential  
improvements. Seek out the views of others. People’s minds work in different ways.

**Iteration**

You should accept that your first attempt at a solution will rarely be the best one. Instead of trying to solve everything in one fell swoop, take an iterative approach. Go back and repeat some of the previous steps in an attempt to improve your current solution

**DECOMPOSITION**

Computational thinking promotes one of these heuristics to a core practice:

Decomposition, which is an approach that seeks to breaking down a complex problem or system into smaller parts that are more manageable and easier to understand. The smaller parts can then be examined and solved, or designed individually, as they are simpler to work with.

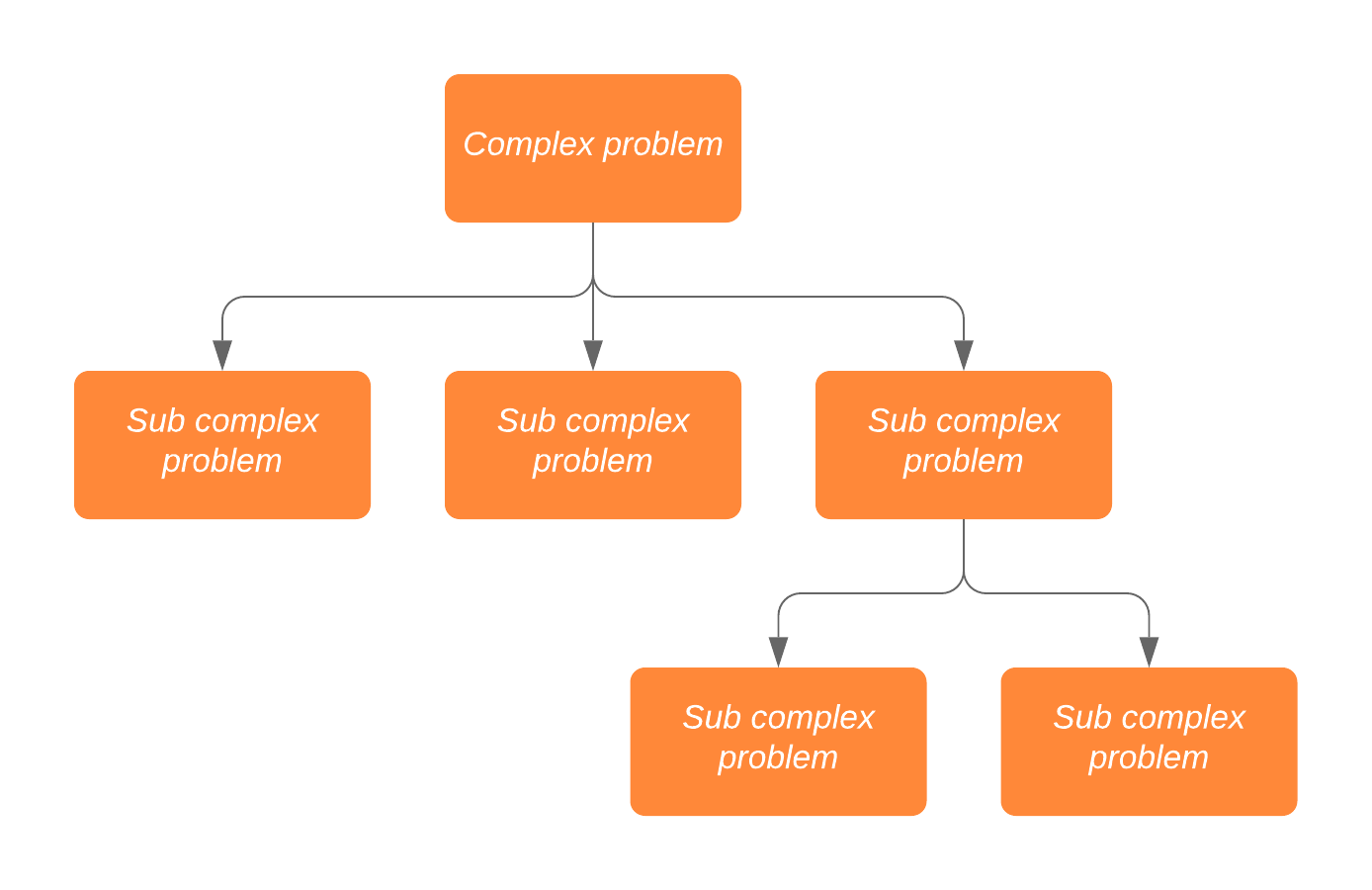
Its particular importance to CT comes from the experiences of computer science. Programmers and computer scientists usually deal with large, complex problems that feature multiple interrelated parts. While some other heuristics prove useful some of the time, decomposition almost invariably helps in man aging a complex problem where a computerised solution is the goal.

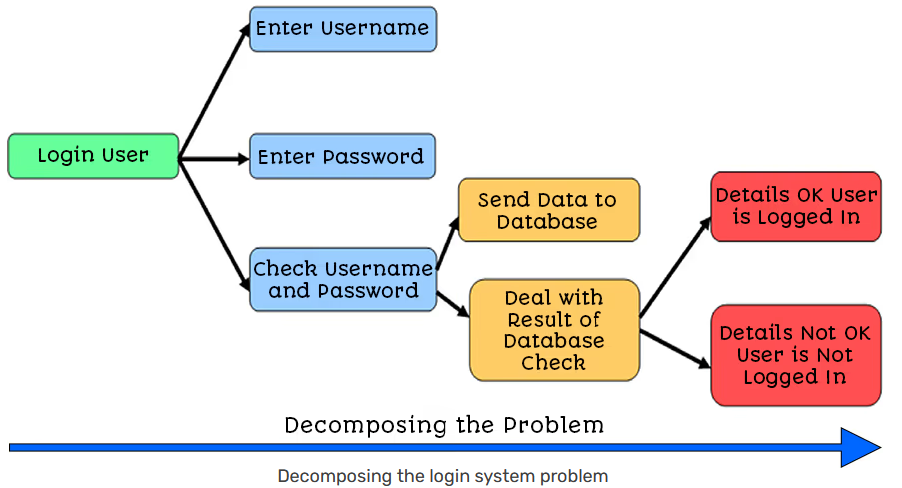
Within the realm of CT, you use divide and conquer when the problem facing is too large or complex to deal with all at once. For example, a problem may contain several  
interrelated parts, or a particular process might be made up of numerous steps that  
need spelling out

**Why is decomposition important?**

If a problem is not decomposed, it is much harder to solve. Dealing with many different stages all at once is much more difficult than breaking a problem down into a number of smaller problems and solving each one, one at a time. Breaking the problem down into smaller parts means that each smaller problem can be examined in more detail.

Similarly, trying to understand how a complex system works is easier using decomposition. For example, understanding how a bicycle works is more straightforward if the whole bike is separated into smaller parts and each part is examined to see how it works in more detail.

1. It helps in solving complex problems easily.
2. The decomposed sub-tasks can be carried or solved by different persons or a group of persons (if one is not having enough knowledge about the full problem).
3. When the problem is divided into sub-tasks each sub-task can be examined in detail.



In programming terms, every password-based login process on the planet must check the password entered by the user against the password stored in the system. If the password matches, the user is logged in.

However, there is no mention of what to do if the user enters the wrong password.

This is where you need to define your problem. If an incorrect password is entered, the system should show the user an error message. But what happens then? How many times does the user get to enter their password? There is not enough information here and we would need to go back to our Client to gather complete requirements.

**Pattern Recognition, Generalisation & Abstraction:**

Once a problem has been decomposed into smaller tasks, Pattern recognition It involves finding the similarities or patterns among small, decomposed problems that can help us solve more complex problems more efficiently. This helps the programmer to save time reinventing the wheel when a solution to a given problem may already exist.

Finding patterns is extremely important. Patterns make our task simpler. Problems are easier to solve when they share patterns, because we can use the same problem-solving solution wherever the pattern exists. The more patterns we can find, the easier and quicker our overall task of problem solving will be.

* Patterns are things that are the same within a problem and between problems.
* Identifying patterns means that there is probably an existing solution already out there.
* Pattern recognition is based on the 5 key steps of:
  + Identifying common elements in problems or systems
  + Identifying and Interpreting common differences in problems or systems
  + Identifying individual elements within problems
  + Describing patterns that have been identified
  + Making predictions based on identified patterns.
* Pattern abstraction is hiding the complexities of one pattern from another.
* Pattern generalisation is spotting things that are common between patterns.
* We can represent parts of a system in general terms, including Variables, Constants, Key Processes, repeated Processes, Inputs and Outputs.

**ANTICIPATING AND DEALING WITH ERRORS**

**Bug: a fault in a solution that can cause erroneous behaviour.**

**Error: any behaviour observed when executing a solution that does not match  
expected behaviour**

NOT: the technical name for this operator is negation. This operator doesn’t chain propositions together itself, rather it modifies a single proposition. Specifically, it flips the truth value. Sometimes, negating a proposition can make it easier to express the chain of reasoning