Learning Aims

- Understand the nature of and need for abstraction
- Describe the differences between an abstraction and reality
- Devise an abstract model for a variety of situations

Computational Thinking

Think how you can decompose the problem

Think about how the solution can be simplified and removing unnecessary details.

Think about inputs and outputs

Think logically Identify when decisions will need to be made

Think about how to make the program more **robust** by using a test table to check a range of user inputs

Think about which parts of the program can be run concurrently.

Think Procedurally how to break up the code using a function/procedure?

Think about how to make the program more **efficient** (execution time required or the space used)

Computational Thinker

- Computational thinkers aim for solutions that are both **correct and efficient**, not just any solution.
- Programmers must prove their solutions work using **logic, test data, and user feedback**.
- Computational thinking is essential for programming and problem-solving.
- It involves **thinking logically** and using compu**ting techniques to tackle challenges**.

Computational Thinking

- **Abstraction** - Hiding or removing irrelevant details from a problem to reduce the complexity.

- **Decomposition** -breaking down a complex problem or system into smaller, more manageable steps

- **Algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem using pseudocode to highlight problems

The difference between abstraction and reality

- Abstraction is a simplified representation of reality
- Abstraction is hiding unnecessary detail and showing details that are important.

Examples of abstraction:

- Creating simple abstract models of real world objects
- Designing interfaces with icons, symbols and colour coding
- Real-world attributes can be stored as variables and constants
- Implementation in a computer may involve abstract data structures, such as arrays, binary trees, graphs

Types of abstraction

- Representational Abstraction
- Abstraction by generalisation
- Data Abstraction
- Procedural abstraction

Representational abstraction can be defined as a representation arrived at by removing unnecessary details.

Example of Representational abstraction

A map is an abstraction made up of a number of component parts.

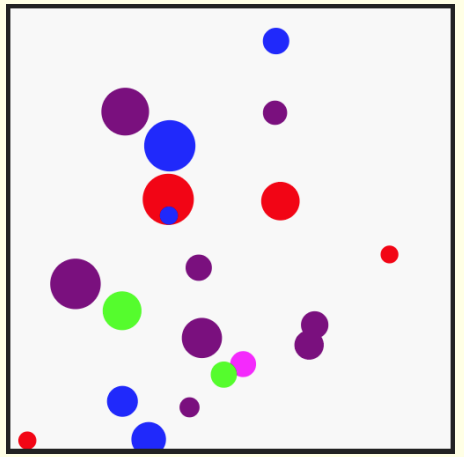
It is a means of hiding detail / only using relevant detail

It is a representation of reality using symbols and labels to show real-life features and irrelevant features, such as buildings and trees are left out.

Examples of Representational abstraction

If you are planning to write a program for a game involving a bouncing ball, you will need to decide:

- What **properties** of the ball to take into account.
- If it's bouncing vertically rather than, say, on a snooker table, gravity needs to be taken into account.
- How elastic is the ball? How far and in what direction will it bounce when it hits an edge?
- What you are required to do is **build an abstract model of a real-world situation**, **which you can simplify**; remembering, however, that the more you simplify, the less likely it becomes that the model will mimic reality.



Simulations and Abstraction

Abstraction in simulations means **simplifying a real-world system by focusing only on the important details** and ignoring unnecessary complexities. This makes it easier to model and understand complex systems.

How Abstraction Helps in Simulations

- **Removes unnecessary details** – Focuses only on key aspects (e.g., in a car simulation, we model speed and fuel but ignore minor engine details).
- **Reduces complexity** – Makes it easier to build and run the simulation.
- **Improves efficiency** – Uses fewer resources, making the simulation run faster.

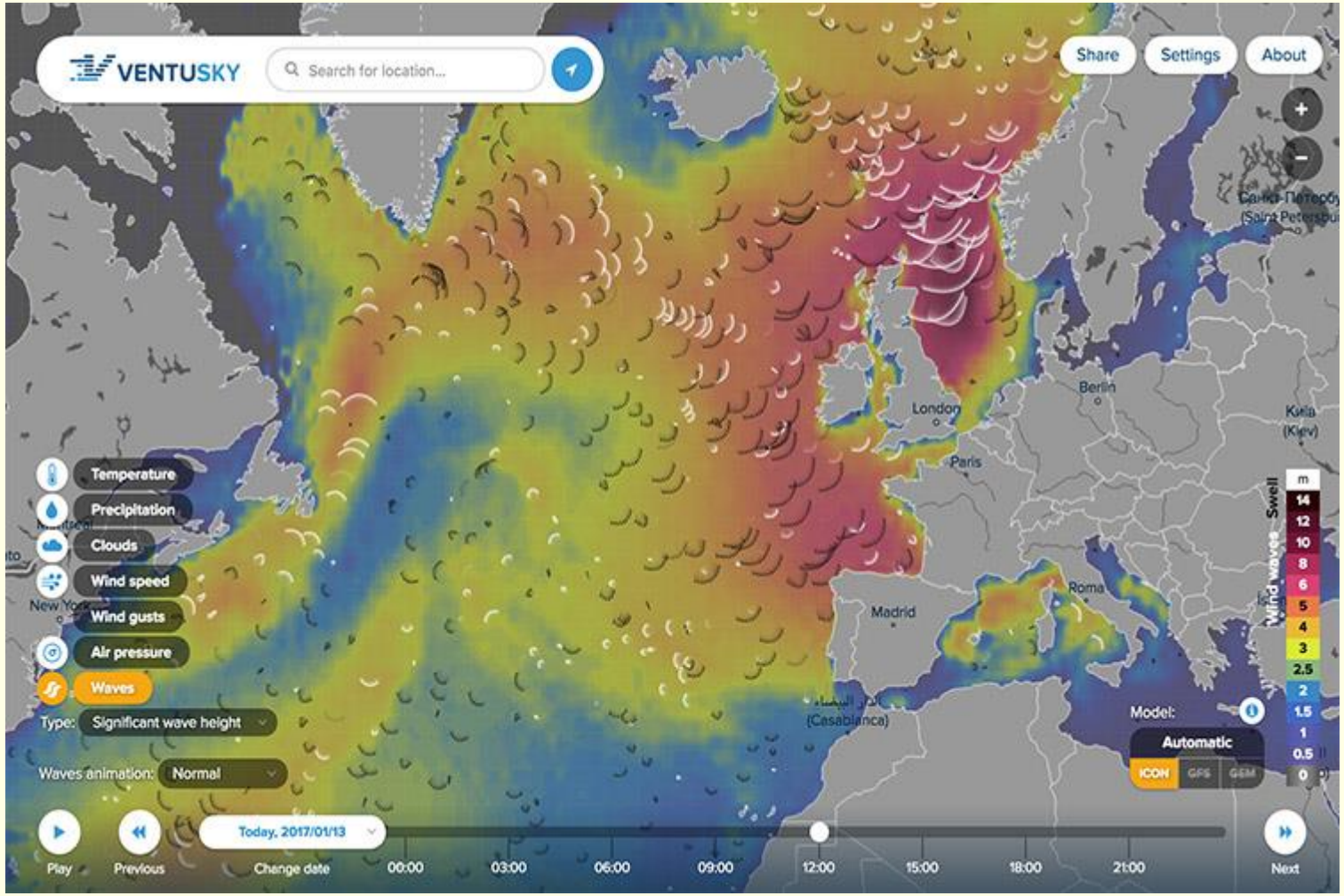
Examples of Simulations Using Abstraction

Example	Real-world system	Abstracted model:
Traffic Simulation	Roads, cars, pedestrians, weather, signals.	Only include **cars, traffic lights, and road rules** while ignoring car brands, engine types, or driver emotions.
Weather Simulation	Includes atmosphere, temperature, wind, humidity, pressure.	Focus only on **temperature, pressure, and wind speed** to predict storms.
Flight Simulator	Airplane mechanics, fuel system, engine power, wind resistance, pilot reactions.	Focus only on **flight controls, aerodynamics, and navigation**, ignoring mechanical details like engine pistons.

Why Use Abstraction in Simulations?

- **Faster and easier to build** – Only key details are included.
- **More understandable** – Helps analyse and predict behaviours efficiently.
- **More flexible** – Can be adjusted based on different needs (e.g., a simple car game vs. a detailed driving simulator).

Abstraction has been used in the design and creation of a weather map



What are the essential features that are included in this abstract model?

What unnecessary detail has been removed?

Abstract models of real-world objects

Car (Vehicle Abstraction)

Real-World Object: A car has many components like an engine, fuel system, transmission, etc.

Abstract Model: A **Car class** in programming might only include essential attributes and methods:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start_engine(self):
        print("Engine started")


    def drive(self):
        print("Car is moving")
```

This abstracts away the **complexity of how a car actually works** (like fuel injection, combustion, and transmission).

This type of abstraction is very common in object-oriented programming

Abstraction applied to high level programming languages

Abstraction is the most important feature of high level programming languages such as Python, C#, Java and hundreds of other languages written for different purposes.

To understand why, we need to look at different generations of programming language.

The first generation of language was machine code – programmers entered the binary 0s and 1s that the computer understands. Writing a program to solve even a short, simple problem was a tedious, time-consuming task largely unrelated to the algorithm itself.

The second generation was an improvement; mnemonic codes were used to represent instructions. It is still an enormously complex task to write an assembly language program and what's more, if you want to run the program on a different type of computer, it has to be completely rewritten for the new hardware.

The third generation of languages, starting with BASIC and FORTRAN in the 1960s, used statements like X = A + 5, freeing the programmer from all the tedious details of where the variables X and A were stored in memory, and all the other fiddly implementation details of exactly how the computer was going to carry out the instruction.

Finally, programmers could focus on the problem in hand rather than worrying about irrelevant technological details, and that is a good example of what abstraction is all about.

Abstraction by generalisation

- Groups similar problems together to form a **general model** or **pattern**.

- Reduces problem-specific details to create a **reusable solution**.

Abstraction by generalisation

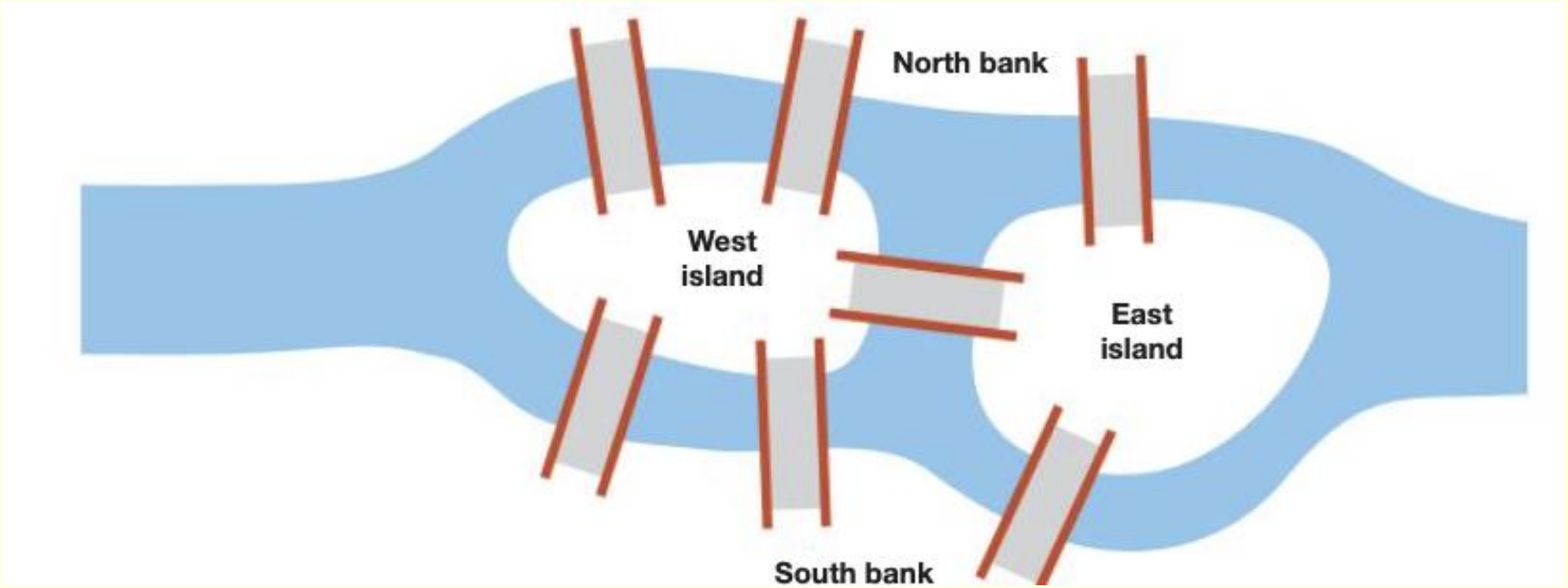
There is a famous problem dating back more than 200 years to the old Prussian city of Königsberg.

This beautiful city had seven bridges, and the inhabitants liked to stroll around the city on a Sunday afternoon, making sure to cross every bridge at least once.

Nobody could figure out how to cross each bridge once and once only, or alternatively prove that this was impossible, and eventually the Mayor turned to the local mathematical genius **Leonhard Euler.**

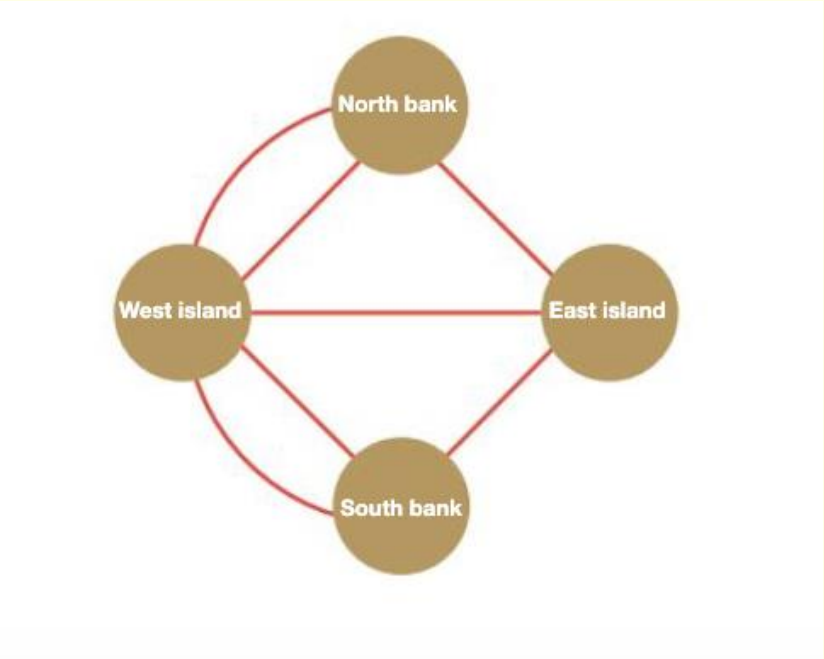
Solving the seven bridges

- Euler's first step was to remove all irrelevant details from the map, and come up with an abstraction:



Solving the seven bridges

- To really simplify it, Euler represented each piece of land as a circle and each bridge as a line between them.



What he now had was a graph, with nodes representing land masses and edges (lines connecting the nodes) representing the bridges.

Now that Euler had his graph, how could he solve the problem?

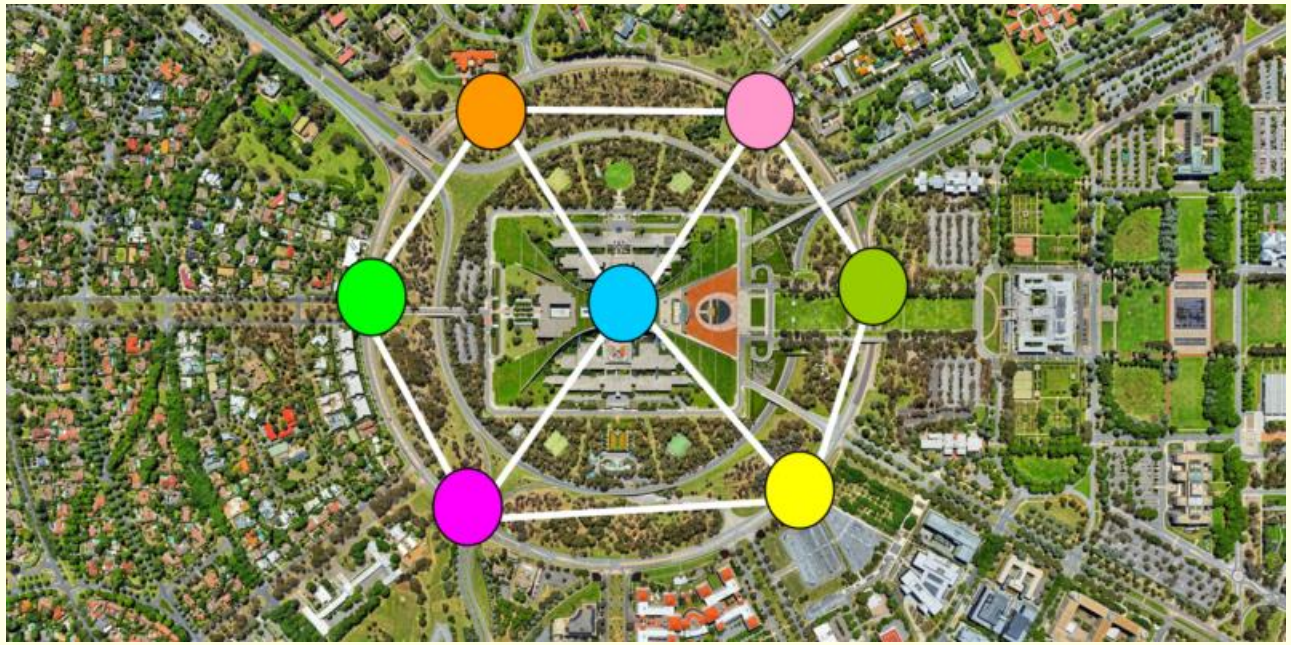
Solving the seven bridges

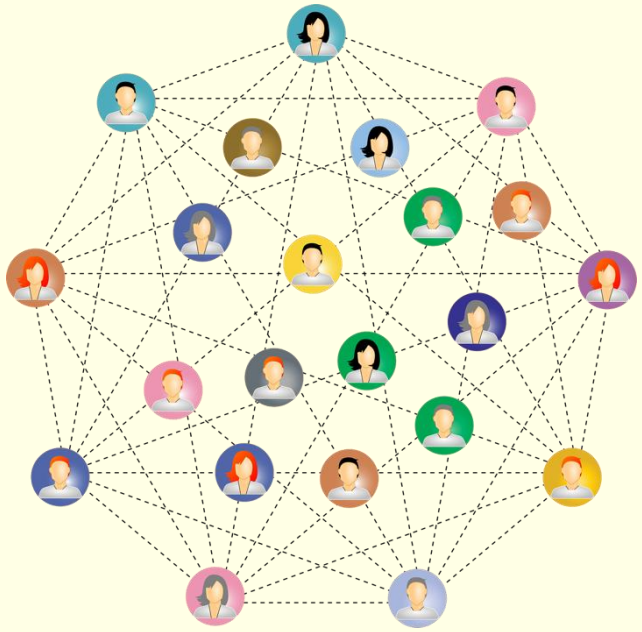
- Euler realised he didn't need to try every solution; instead, he focused on finding a general solution for similar problems.
- He noticed that to cross each bridge only once, each point (node) must have an even number of connections, except for the start and end points.
- Since all the points in the puzzle had an odd number of connections, **solving it was impossible!**
- Euler's work formed the basis of graph theory, allowing similar problems to be solved through abstraction.
- **By generalizing the problem**, Euler showed how it applies not just to cities with bridges but to **many other problems** with similar rules.

Graph data structure is used with maps but can also be used on **social media** sites.

Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them.

Facebook's Friend suggestion algorithm uses graph theory.





Abstraction by generalisation

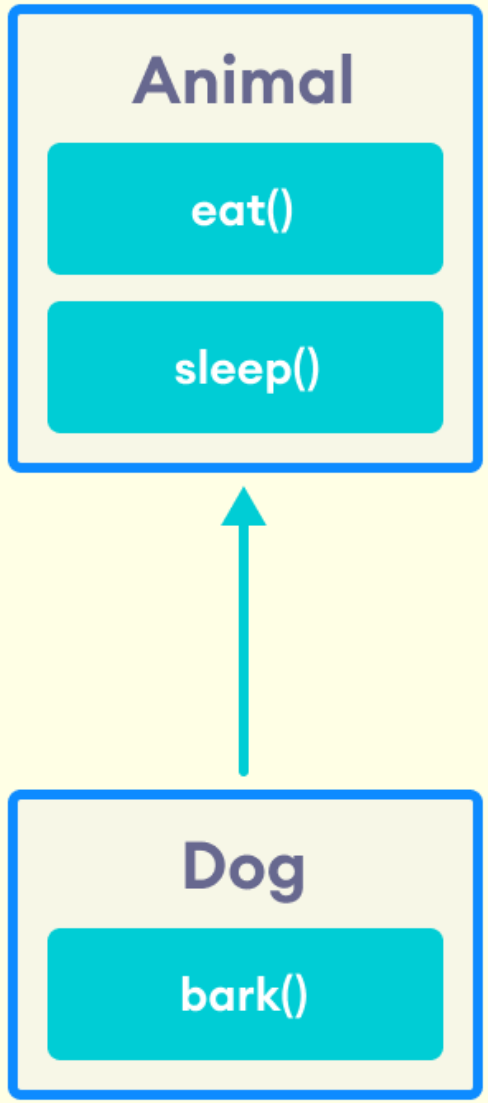
- Also means **grouping similar things into a more general category** by identifying what they have in common.

- Instead of focusing on specific details, we create a broad category that applies to all similar objects.

Example:

dog, cat, and bird → Animal

- A **dog, cat, and bird** are all animals.
- Instead of treating them separately, we group them under **"Animal"**, since all animals **eat, sleep, and move**.

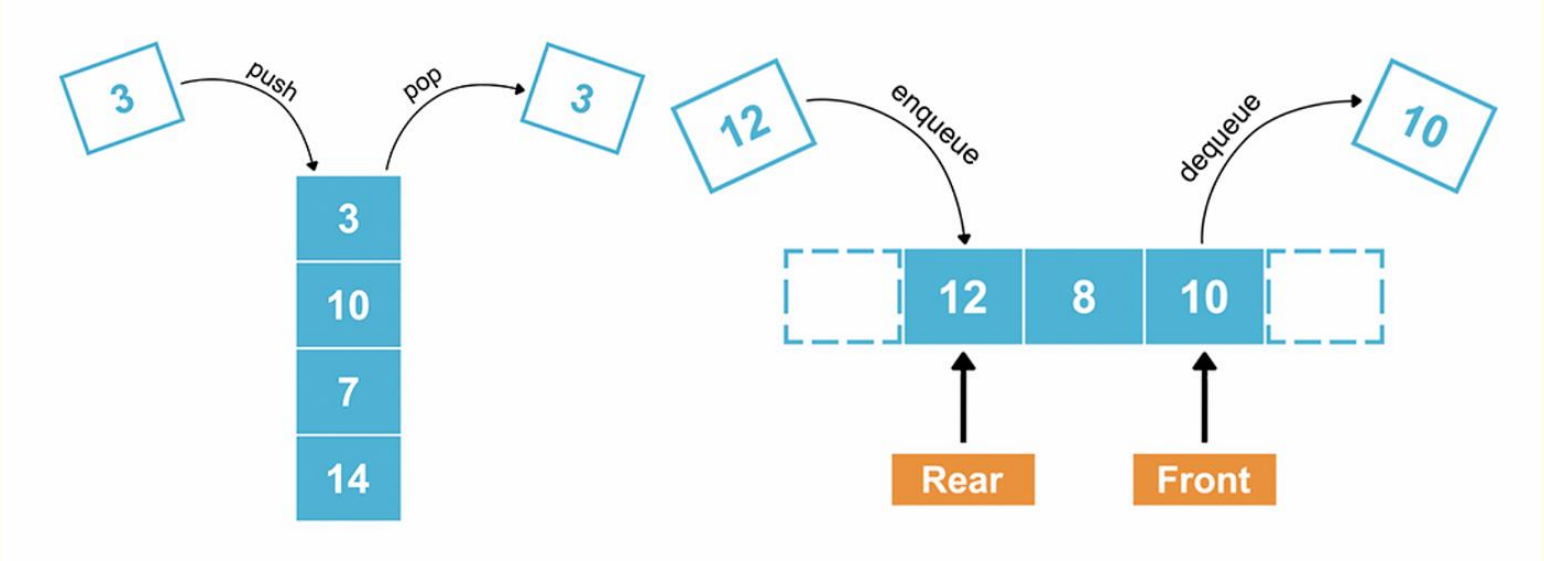


Data abstraction

- Hides the way **data is stored** and manipulated, providing only necessary details.
- Uses **abstract data types** such as stacks, queues, and lists.

Example:

- Using a stack.push(x) and stack.pop() without knowing the internal array or linked list implementation.



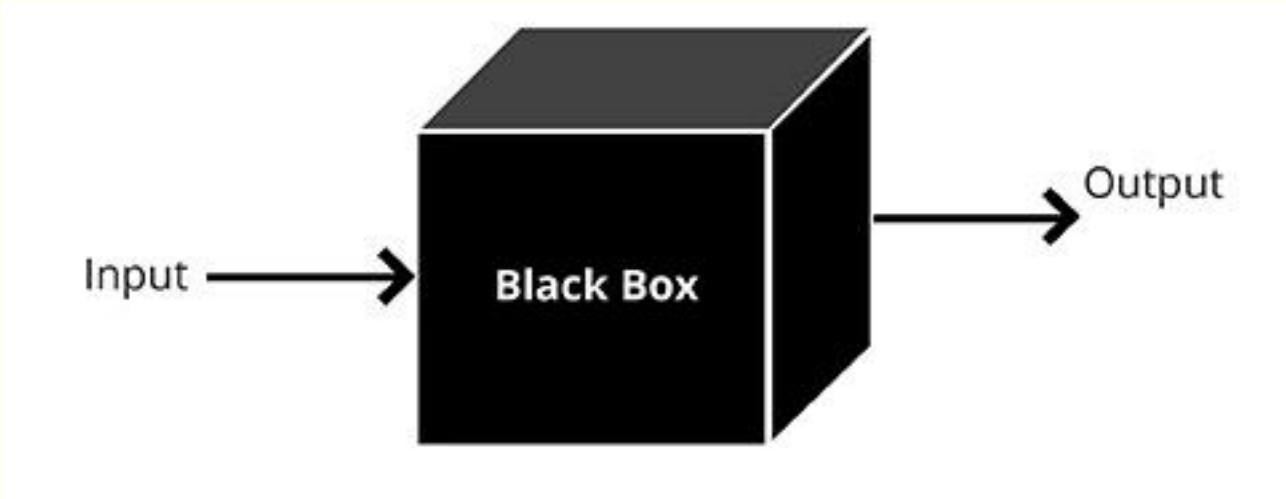
This is an abstract model for explaining how to stack and queue work. How stacks and queues are actually stored in memory and implemented is hidden.

Procedural abstraction

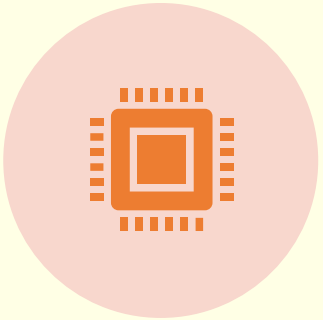
- Focuses on **breaking down** a problem into smaller procedures (subroutines or functions).
- Hides **implementation details**, allowing users to call a procedure without knowing how it works internally.

Example:

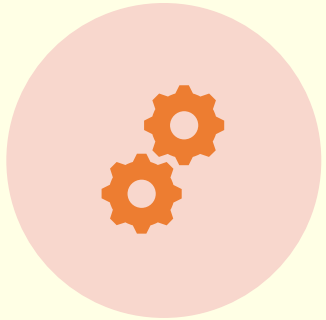
- A function calculateInterest(amount, rate, years) computes interest without revealing its internal formula.



Why abstraction ...



Reduces programming time and factors that can detract from the program - programmers can focus on core elements rather than unnecessary data



Reduces complexity of programming

Focus is on the core aspects of the program rather than the extras

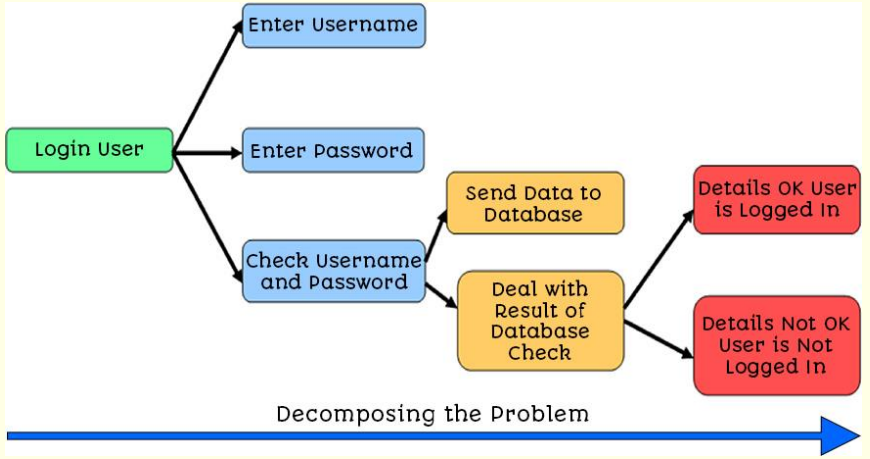
Too much abstraction can detract from the appeal of the application I.e. a game, may be too simplistic/not realistic enough, may not have enough scope to engage users

Computational Thinking	**Compatibility** – whether or not a problem can be solved using an algorithm. **Speed and memory** - limiting factor. As larger amounts of power become available we can tackle more problems using a computer. Solving problems is joint between humans and computers. Some problems will never be solvable by computers.

Decomposition

What is Decomposition?
Decomposition is the process of breaking down a complex problem into smaller, manageable parts.
Why Use Decomposition?

- Smaller parts are easier to solve.
- Makes it easier to assign tasks to a team.
- Helps in creating subroutines/modules for better organisation.
- Allows parallel development, where different teams work on different sections.



Problem Decomposition Process

1. Break down the problem into smaller parts.
2. Keep dividing until each part is a single task.
3. Each task becomes a subroutine that can be developed and tested separately.
4. Subroutines are combined to create the full solution.
5. Existing modules/libraries may be used to simplify tasks.

Key Technique: Top-Down Design (Stepwise Refinement)

- Problems are divided into multiple levels of complexity.
- Development starts with the simplest tasks and builds up to the full solution.

Learning Aims

Thinking Ahead

- Identify the inputs and outputs for a given situation
- Determine the preconditions for devising a solution to a problem
- Understand the need for reusable program components
- Understand the nature, benefits and drawbacks of caching

Computational problems

At its most abstract level, a computational problem can be represented by a simple diagram:

Input is the information relevant to the problem, which could for example be passed as parameters to a subroutine.

Output is the solution to the problem, which could be passed back from a subroutine.

A clear statement of exactly what the inputs and outputs of a problem are is a necessary first step in constructing a solution.

Example 1: Determine whether a given item is present in a list

On the face of it, this is a simple problem.

But do we know exactly what the inputs are? For example:

Is the list sorted? Are the items numeric or alphabetic?

What about the output – are we expecting it to be simply True or False, or should the output give the position in the list of the item if it is found?

The problem needs to be formally defined, stating the inputs and outputs. This can be done as follows:

Name:	SearchList
Inputs:	A list of strings S = (s1, s2,s3,sn) A target string t
Outputs:	A Boolean variable b

Now we can write pseudocode for the function SearchList:

```
function SearchList(s, t)
    found = False
    n = 0
    while found == False AND n < len(s)
        if t == s[n] then
            found = True
        else
            n = n + 1
        endif
    endwhile
    return found
endfunction
```

Write a pseudocode algorithm which initialises a list of string items, asks the user to enter an item to search for, calls the above function SearchList and prints an appropriate message depending on whether the function returns True or False.

```
aList = [‘banana, ‘pear’, ‘blackberry’, ‘rhubarb’]
searchItem = input(“Please enter item: “)
inList = SearchList(aList)
if (inlist == True)
  print(“Item in list”)
else
  print(“item not in list”
endif
```

Identifying inputs, processes and outputs

"Write a program that asks the user for the number of students in their class and prompts them to enter each student's test score within the range of 0 – 100. It should then output the highest, lowest and average score."

Inputs

NumOfStudents: Integer
CurrentScore: Integer

Processes

TotalScore = TotalScore + CurrentScore
AverageScore = TotalScore / NumOfStudents
- Loop through the array and return the lowest score
- Loop through the array and return the highest score

Outputs

MinScore: Integer
MaxScore: Integer
AveScore: Real/Float

Example: Identifying the inputs & outputs

You must be able to identify the inputs and outputs that would be required to form a program given a scenario.

Take, for example, a program designed for an ATM.

Inputs	Outputs
Transaction type: Deposit? Balance check? Withdrawal?	If deposit selected: Display total amount entered on screen
Card details, captured using a card reader	If balance check selected: Display total account balance on screen
PIN, entered via keypad	If withdrawal selected: Dispense correct amount of cash.
	Print receipt to confirm transaction
	Speaker provides verbal feedback throughput.

Specifying preconditions

Suppose that a pseudocode algorithm has been written to find the maximum of a list of numbers.

```
function maxInt(listInt)
    maxNumber = listInt[0]
    for i = 1 to len(listInt) - 1
        if listInt[i] > maxNumber then
            maxNumber = listInt[i]
        endif
    next i
    return maxNumber
endfunction
```

In order to make sure the function never crashes, either the function must test for an empty list, or a precondition must be specified with the documentation for the function.
Name: maxInt

Inputs: A list of integers listInt = (k_1,K_2,K_3…K_n)

Outputs: An integer maxInt

Precondition: **length of listInt > 0**

If the function is called with an empty list, it will crash on the statement

maxNumber = listInt[0]

Specifying preconditions

Specify the input, output and any preconditions for a function sqrt(n) which finds the square root of an integer or floating point number.

Name: sqrt

Input: integer or floating point number *n*

Output: square root of n

Precondition: n >= 0

Advantages of specifying preconditions

- Specifying preconditions as part of the documentation of a subroutine ensures that the user knows what checks, if any, must be carried out before calling the subroutine.

- If there are no preconditions, then the user can be confident that necessary checks will be carried out in the subroutine itself, thus saving unnecessary coding. The shorter the program, the easier it will be to debug and maintain.

- Clear documentation of inputs, outputs and preconditions helps to make the subroutine reusable. This means that it can be put into a library of subroutines and called from any program with access to that library.

How Programmers Use Reusable Components in Large Programs

Programmers use **reusable components** to save time, reduce errors, and make development more efficient. These components include **functions, modules, libraries, and classes** that can be used multiple times in a program or across different projects.

Libraries & Modules
- Pre-built libraries (e.g., Python's math or random) are imported instead of rewriting common functions.
- This improves efficiency and reliability.

Functions & Subroutines
- Frequently used code (e.g., file handling, validation) is written as functions.
- This reduces repetition and makes debugging easier.

Modular Programming
- Large programs are split into **small, reusable modules** that can be used in different projects.
- Makes development more manageable.

Object-Oriented Programming (OOP)
- Classes are used to create reusable objects with properties and methods.
- Example: A Car class with attributes (speed, colour) and methods (accelerate(), brake()).

Version Control & Code Repositories
- Tools like **GitHub** store reusable code, making it easy for teams to share and reuse components.

Reusability

Benefits	Drawbacks
Benefits: **Saves Time** – No need to rewrite common code. **Easier Maintenance** – Updates apply across all uses of the component. **Fewer Errors** – Well-tested components reduce bugs. **Better Collaboration** – Teams can work on separate modules and combine them. Using reusable components makes software **faster to develop, more reliable, and easier to maintain**.	Third-party components may not always be compatible with existing software. They may need modification, which can be **costly** and **time-consuming**, sometimes making in-house development a better option.

Nature and benefits of caching

Caching **temporarily stores** program instructions or data that were recently used and may be needed again soon.

For example:

The last few instructions of a program may be kept in **cache memory** for **quick access.**

Web caching is another example, where recently viewed **HTML pages and images** are stored. This allows **faster access** to previously visited pages and reduces unnecessary bandwidth usage by avoiding repeated downloads.

Benefits and Drawbacks of caching

Benefits	**Drawbacks**
Caching allows for **faster response times** and overall better device performance.	**Caching** depends on how well a caching algorithm is able to manage the cache. Larger caches still take a long time to search and so cache size limits how much data can be stored
Prefetching can be difficult to implement but can significantly improve performance if implemented effectively.	**Prefetching** - The biggest limitation is the **accuracy** of the algorithms used in prefetching, as they can only provide an informed **prediction** as to the instructions which are likely to be used and there is no guarantee that this will be right.

Worked Example

A user working on a PC at home has done several searches on the Internet and leaves the browser windows opens when she shuts down. The next day, her mother needs to use the PC. On opening the browser, all the windows that were previously open, appear automatically.

How does this happen? What are the advantages? What are the drawbacks? How can this situation be prevented?

The pages are cached by the operating system and reloaded when the browser is loaded again.

Advantage

Often very convenient for the user, who can continue work from where they left off, without having to search for pages again

Disadvantage

Possibly the user who loaded the pages may not want another user to see the pages they were looking at.

The situation can be avoided if each user has their own login password.

Learning Aims

- Identify the components of a problem
- Identify the components of a solution to the problem
- Determine the order of steps needed to solve a problem
- Identify sub-procedures necessary to solve a problem

Procedural abstraction

Procedural abstraction means using a **procedure** to carry out a sequence of steps for achieving some task such as calculating a student's grade from her marks in three exam papers, buying groceries online or drawing a house on a computer screen.

Consider, for example, how you could code a program to create the plan for an estate of 100 new houses.

You could use a procedure which will draw a triangle of certain dimensions and colour. The colour and dimensions are passed as arguments to the procedure, for example:

procedure drawTriangle(colour, base, height)

This procedure may be called using the statement:

drawTriangle("red", 4.5,2.0)

The programmer does not need to know the details of how this procedure works. They simply needs to know how the procedure is called and what arguments are required, what data type each one is and what order they must be written in. This is called the **procedure interface.**

Problem decomposition

- Thinking procedurally makes the task of writing a program a lot s**impler by breaking a problem down into smaller parts** which are **easier to understand** and consequently, **easier to design.**

- The first stage of thinking procedurally in software development involves taking the problem defined by the user and breaking it down into its component parts, in a process called problem **decomposition.**

Test and maintain

- Simpler to **test and maintain.**

- When a change has to be made, if each module is **self-contained** and **well documented** with **inputs, outputs and preconditions** specified, it should be relatively easy to find the modules which need to be changed, knowing that this will not affect the rest of the program.

- By separating the problem into sections, it becomes more feasible to manage and can be **divided between a group of people** according to the skill sets of different individuals.

Advantages of using modular design

- Each module focuses on a **small sub-task** and so it is easy to solve and **test/debug**
- Makes it **easy to maintain** and **update** part of the system without affecting the rest
- The program will be well-structured
- Development can be **shared** between a **team of programmers** so developed **faster**, allocated according to expertise, improving the quality of the final product
- Reduces the amount of code as it can be **reused**, standard library modules can be used and reduces development time.

Top-down design

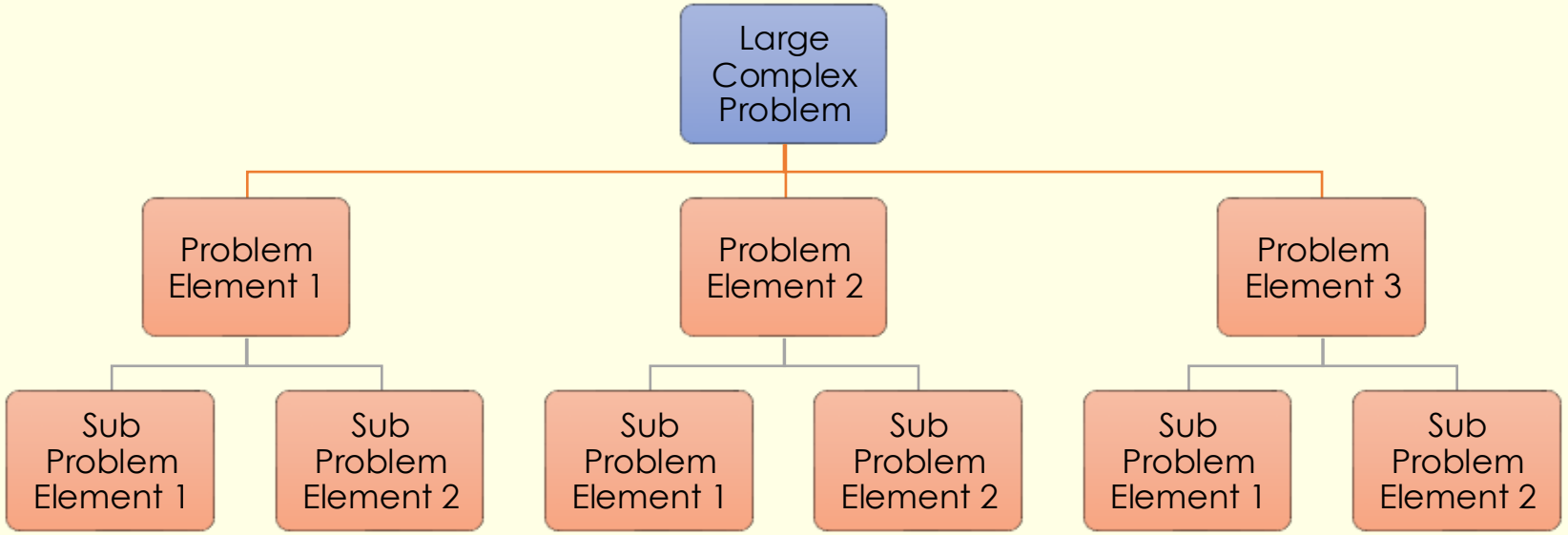
- Top-down design is the technique of breaking down a problem into the major tasks to be performed

- Each of these tasks is then further broken down into separate subtasks, and so on until each subtask is sufficiently simple to be written as a self-contained module or subroutine.

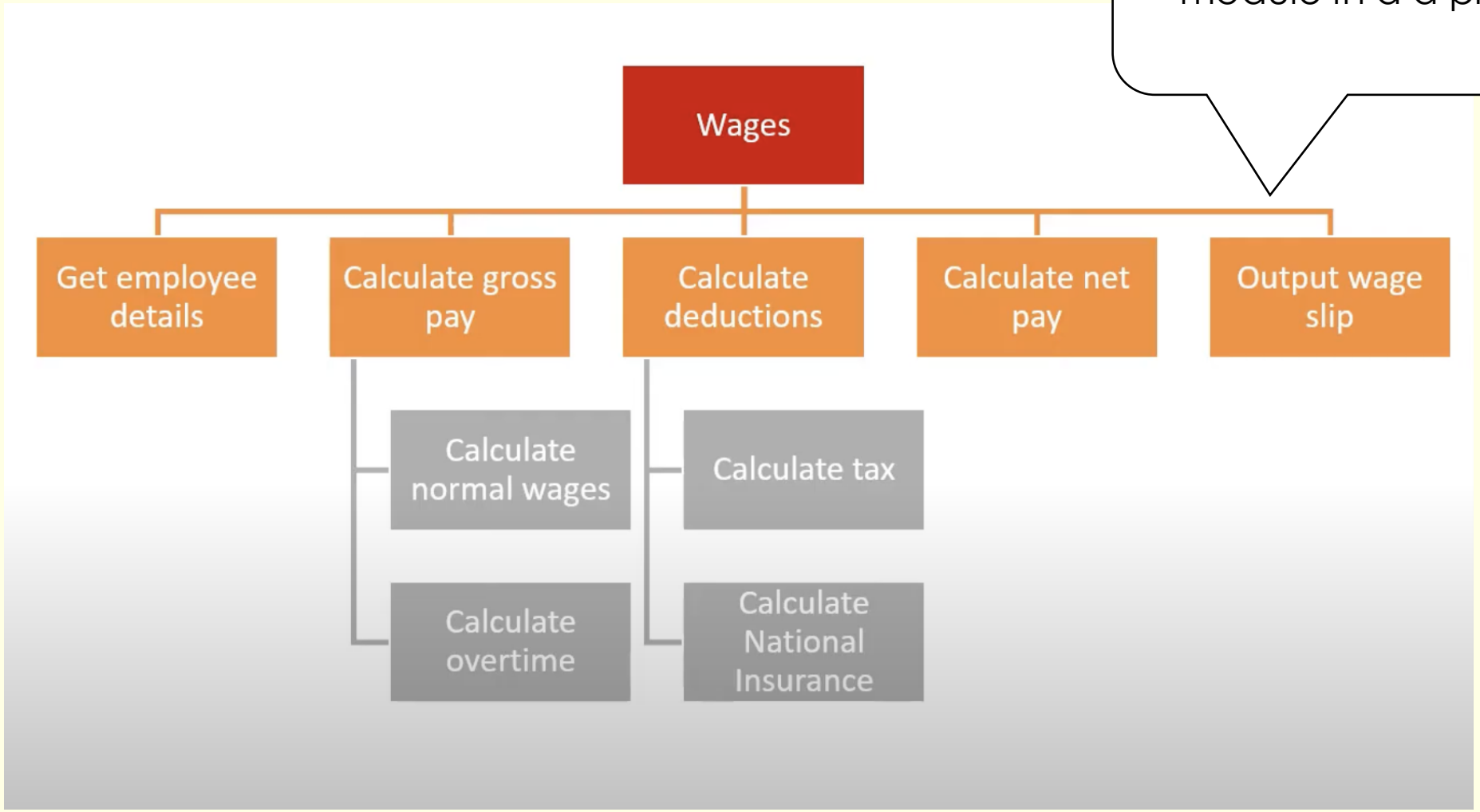
- Remember that some programs contain tens of thousands, or even millions, of lines of code, and a strategy for design is absolutely essential.

- Even for small programs, top-down design is a very useful method of breaking down the problem into small, manageable tasks.

As you can see in the ***'Top-Down'* diagram** below, a problem can be split up into its different elements. These elements can then be further split into sub problems which are more easily to solve.



Example



Each element could be a module in a a program

Summary of Decomposition

- Splits problem into sub-problems
- Splits these problems further until each problem can be solved
- Smaller problems are easier to solve
- This approach leads to the **Modular Design** approach for computer programming.
- Team of programmers can work on different elements of the design and speed up production
- Identify if some of the sub-problems can be solved using pre-existing modules and see if new modules can be re-used in different parts of the program.

Learning Aims

- Identify the points where a decision has to be taken
- Determine the logical conditions that affect the outcome of a decision
- Determine how decisions affect flow through a program

The structured approach

The structured programming approach aims to improve the clarity and maintainability of programs.

Using structured programming techniques, only three basic programming structures are used:

- **sequence** – one statement following another
- **selection –** if … then … else… endif and switch/case … endswitch statements
- **iteration** – while … endwhile, do… until and for … next loops

Languages such as Python block-structured languages which allow the use of just three control structures.

They may allow you to break out of a loop, but this is not recommended in structured programming.

Each block should have a single entry and exit point.

Thinking Logically

- It is important to identify where decisions need to be made within the program, and plan out the outcomes of the decision made.

This involves:

- Identifying decision points for branching or iteration.
- Determine the conditions of the decision.
- Determine the next steps depending on the outcome of the decision.
- A **flow chart and pseudocode** can be used to designing a solution to a problem.

Learning Aims

- Determine which parts of a program can be tackled at the same time
- Determine the benefits and trade-offs of concurrent processing

Thinking Concurrently

- ‘Thinking concurrently’ is the need to work out which parts of a program can be developed to be processed **at the same time**, and which parts are dependent on other parts

Thinking Concurrently

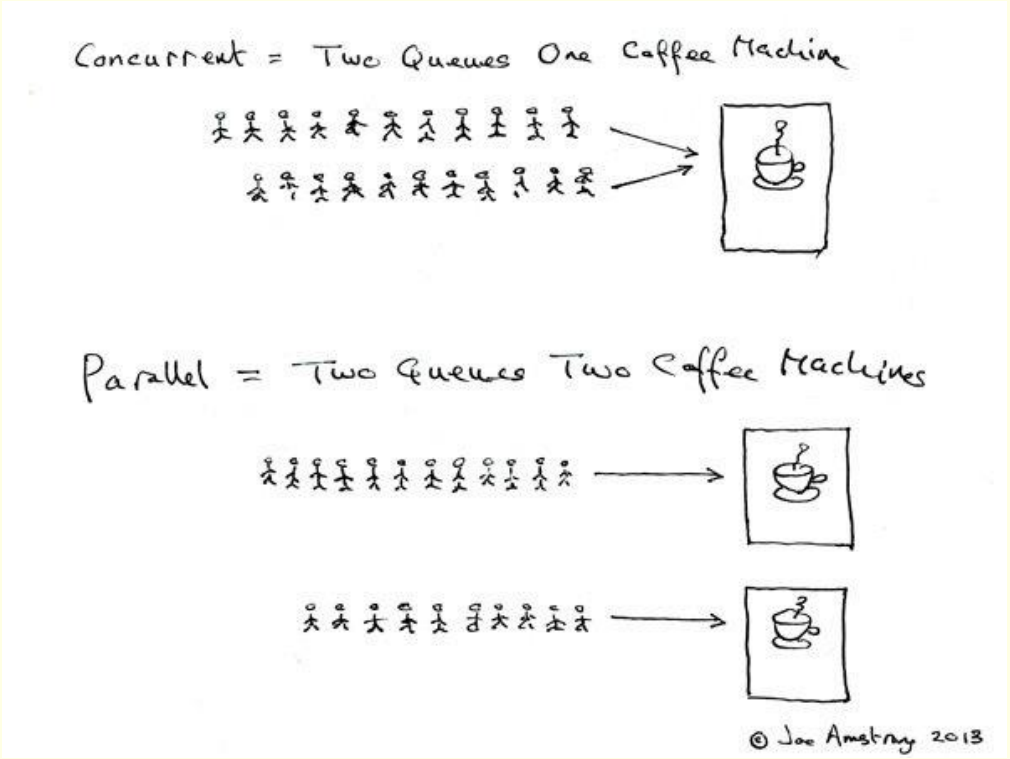
- Any situation in the design or programming of a system when you would want more than one thing happening at the same time.
- Means programs need to be specifically designed to take advantage of this.
- Modules processed at the same time should be independent.
- Well-designed programs can save a lot of processing time.
- Programs have to be written specifically to take advantage of parallel processing which can make them longer and more complex.
- However, there is the issue of sequential tasks, **not all tasks can run concurrently**, which means sometimes there is no time saving as all the tasks still need to be run in order.

Concurrent computing vs Parallel computing

- Generally, concurrent computing is defined as being related to but distinct from parallel computing.

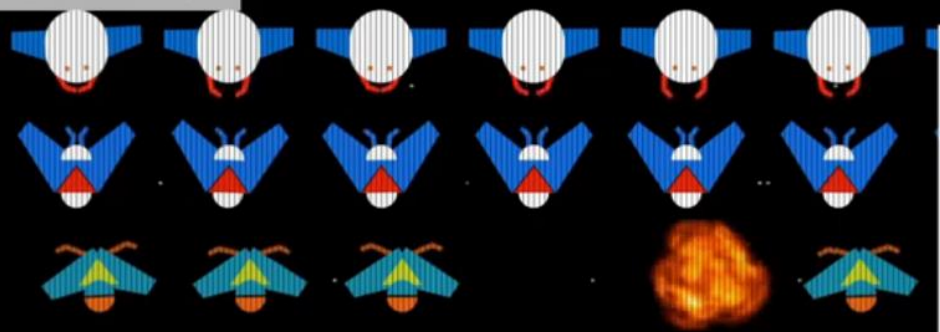
- **Parallel computing** requires **multiple processors** each executing different instructions **simultaneously**, with the goal of speeding up computations.
- It is impossible on a single processor.

- **Concurrent processing**, on the other hand, takes place when several processes are running, with each in turn being given a **slice of processor time**. This gives the appearance that several tasks are being performed simultaneously, even though only **one processor** is being used. (Processor scheduling algorithms)



Being able to think concurrently

- Moving the player ship.
- Firing bullets.
- Bullet movement.
- Enemy ship movement.
- Updating players score.
- Collision detection.

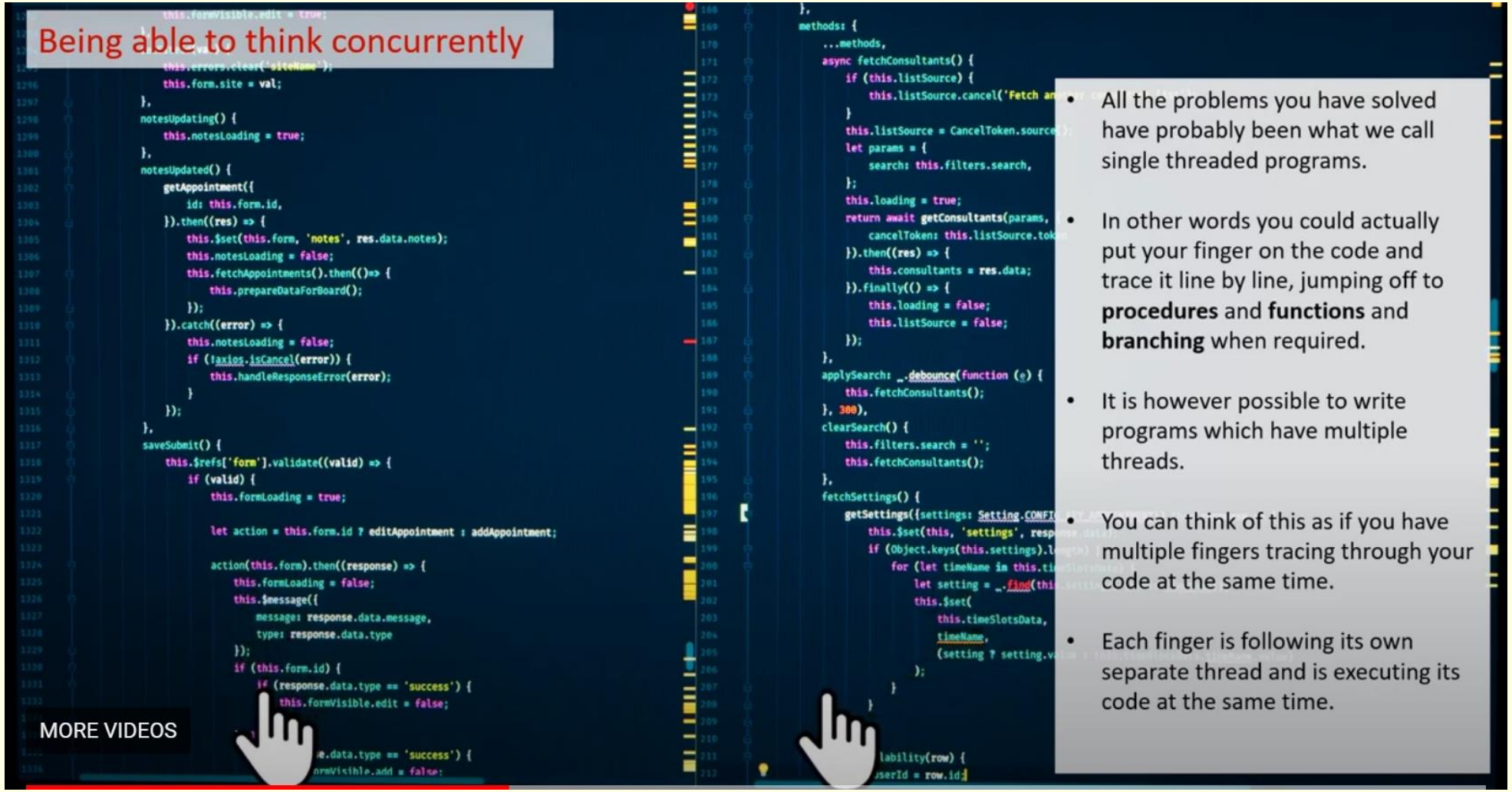


- Consider this simple space shooter game.
- **Thinking concurrently**, what aspects of this problem could be developed to take place (be processed) at the same time?

Concurrent computing and multithreading

- Parallel - Processes are happening at the same time only if multicore
- Concurrent - On a single core only 1 process can actually happen at a time, concurrent tries to **simulate multiple processes**.
- Individual processes are **threads,** each thread has a life line.

Multi-threading



- All the problems you have solved have probably been what we call single threaded programs.
- In other words you could actually put your finger on the code and trace it line by line, jumping off to **procedures** and **functions** and **branching** when required.
- It is however possible to write programs which have multiple threads.
- You can think of this as if you have multiple fingers tracing through your code at the same time.
- Each finger is following its own separate thread and is executing its code at the same time.

Benefits and trade-offs of concurrent processing

- Concurrent processing has benefits in many situations.
- Increased program **throughput** – the number of **tasks** completed **in a given time** is increased
- Time that would be wasted by the processor **waiting** for the user to input data or look at output is used on another task
- The drawback is that If a large number of users are all trying to run programs, and some of these involve a lot of computation, these programs will take longer to complete

Benefits and trade-offs of parallel processing

- Parallel processors enable several tasks to be performed simultaneously by different processors.
- It can speed up processing enormously when repetitive calculations need to be performed on large amounts of data
- Graphics processors can quickly render a 3-D object by working simultaneously on individual components of the graphic
- A browser can display several web pages in separate windows and one processor may be carrying out a lengthy search or query while processing continues in other windows

Parallel processing has limitations:

- There is an overhead in coordinating the processors and some tasks may run faster with a single processor than with multiple processors.

Learning Aims

Problem recognition

- Know what features of a problem make it soluble by computational methods
- Categorise different types of problem and solutions
- Explore different strategies for problem-solving
- Discuss problem recognition

Computable problems

- A problem is defined as being computable if there is an algorithm that can solve every instance of it in a finite number of steps.
- Some problems may be, in theory, computable, but if they take millions of years to solve, they are, in a practical sense, insolvable
- An example of such a problem is the cracking of a secure password.
- If you choose a password of 10 characters or more, comprising a mixture of random letters, numbers and special symbols, it will be impossible to crack. You can test the strength of your passwords on various websites.

Methods of problem solving

There are many ways of problem solving, including:

Problem-solving method	Brief description
Abstraction and decomposition	Simplifying a problem by removing unnecessary detail (abstraction) and breaking it into smaller, manageable parts (decomposition).
Enumeration (listing all cases)	Solving a problem by systematically listing all possible cases or solutions and checking each one. (Brute Force)
Simulation and automation	Using a model or computer program to imitate a real-world process, often automating repeated steps to test different scenarios efficiently.
Theory and mathematics	Applying mathematical formulas, rules, or theoretical principles to analyse the problem and reach a logical solution.

Enumeration - Brute force method

- Many problems and algorithmic puzzles can be solved by exhaustive search – trying all possible solutions until the correct one is found.

- Thousands of problems which were in the past insoluble have, thanks to the power of modern computers, become soluble.

For example, a database of fingerprints or DNA can within a reasonable time find the identity of an individual, if his or her fingerprints or DNA are on the database.

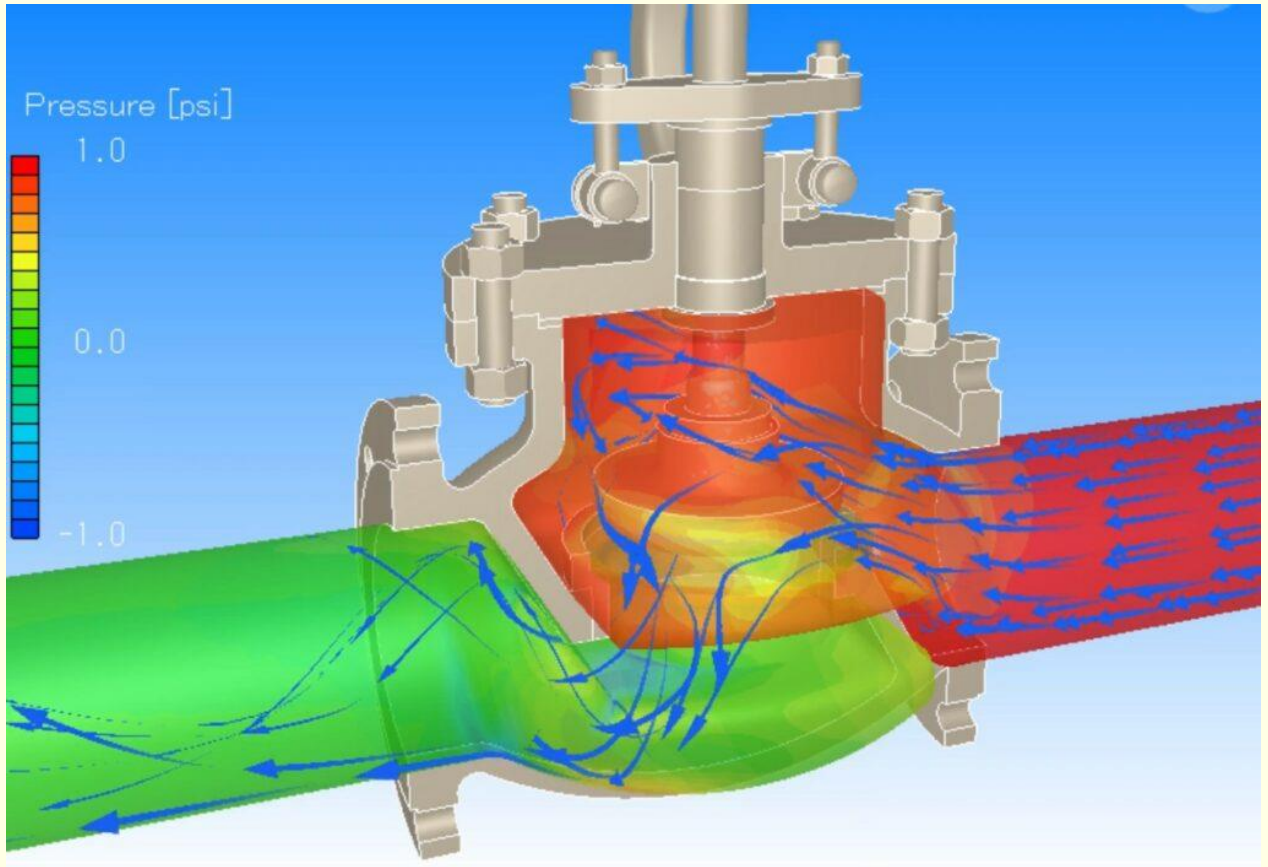
The main problem with the exhaustive search strategy is that it becomes very slow and inefficient as the problem gets bigger. This is because, as the size of the problem increases, the number of possible solutions grows very quickly, making it harder and more time-consuming to check every option.

Simulation

Simulation is the process of designing a model of a real system in order to understand the behaviour of the system, and to evaluate various strategies for its operation.

Such problems include:

- Financial risk analysis
- Population predictions
- Queueing problems
- Climate change predictions
- Engineering design problems



- Simulation can also involve building a physical model of, for example, a spacecraft, ship or wind turbine, so that its behaviour can be studied.

- This is obviously useful when it would be too expensive, dangerous or impractical to carry out tests on the real thing.

- A model can be used to evaluate performance or test outcomes.