Computational Thinking

Core Concepts

What is Computational Thinking?

Computational thinking is a way of understanding complex problems in order to form a solution that can be implemented by either a computer, a human or a combination of both.

There are four key elements to computational thinking:

Decomposition - breaking a problem down into smaller pieces

Abstraction - removing details that are not relevant to solving a problem

Algorithmic thinking - identifying logical steps/a set of instructions that can be used to solve a problem

Pattern Recognition - Finding patterns or similarities

New Words!

Decompose

Say it with me: De-com-pose

Break a problem down into smaller pieces

Pattern Matching

Say it with me: Pat-ern Mat-ching

Finding similarities between things

Abstraction

Say it with me: Ab-strac-shun

Pulling out specific differences to make one solution work for multiple problems

Algorithm

Say it with me: Al-go-ri-thm

A list of steps that you can follow to finish a task

Abstraction

Abstraction is a computational thinking technique that **simplifies a problem** by **removing unnecessary detail** so that you can **focus on the important parts** that are relevant to the problem.

For example, a school might want a computer system to store details about pupils and staff. Each pupil or staff member would be represented as a row in a database with fields for the most important information like their name, address, date of birth and so on (rather than what they had for breakfast or who their favourite actor is). Each entry in the database is an **abstraction** of the real-life person. It is a simplified form that represents the relevant details about them for this particular "problem".

Examples of abstraction

The London Underground map is a classic example of abstraction.

It focuses on the important details such as the station names and the routes from one to another along the various train lines.

The lines are **not** realistically representative of the actual paths taken by each train.

Image by Transport for London

Tube map abstraction vs real paths and distances

Tube map abstraction

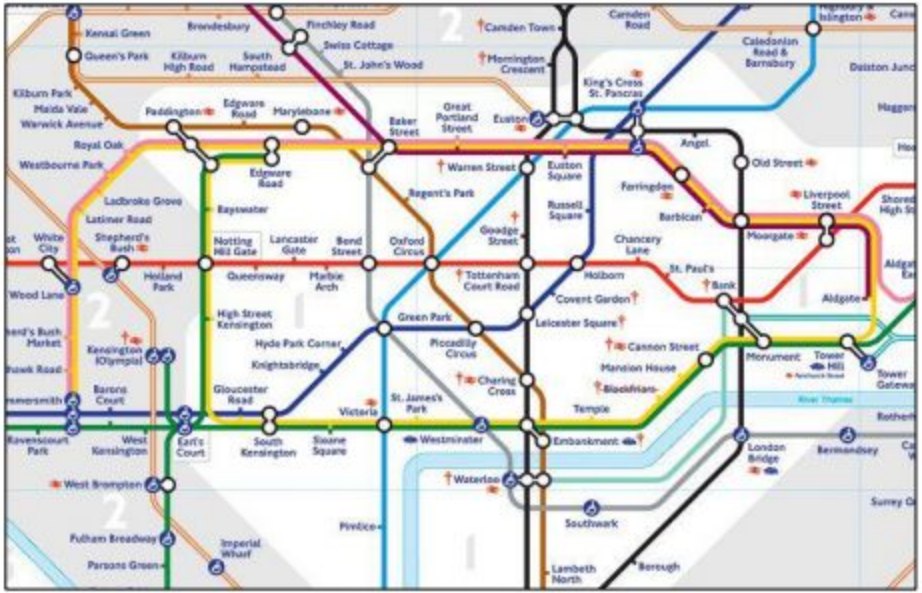


Image by Transport for London

Realistic distances and locations

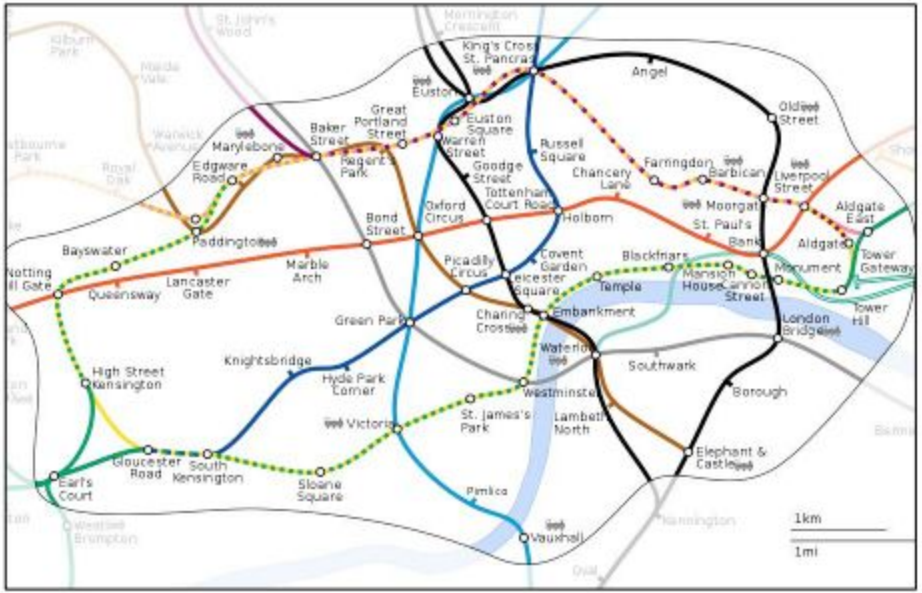


Image by Jc86035

All maps are examples of abstraction

- A map **represents something from the real world**
- It **removes unnecessary details** (plants, trees, colour of houses)
- It **represents** the **relevant information** that you need with symbols or shapes (such as buildings and roads)
- The purpose of the map will determine what information is left out. For example a map for climbers and hill walkers might include information about the relative height of the landscape, whereas a driving map may focus on showing roads more clearly.
- When planning a route from one location to another, a mapping app needs to know which roads connect those locations but it doesn't need to know every detail of the landscape between those locations.

Other examples of abstraction

In programming

- **Variables** in programs are used to represent real entities such as the name of a player or their score.
- A piece in a board game might be represented in a computerised version of that game as a **grid coordinate** whilst the board itself might be represented as a **2D array**.

In computer systems

- **Tables** in a database represent real-life entities (people, products, etc)

In the real world

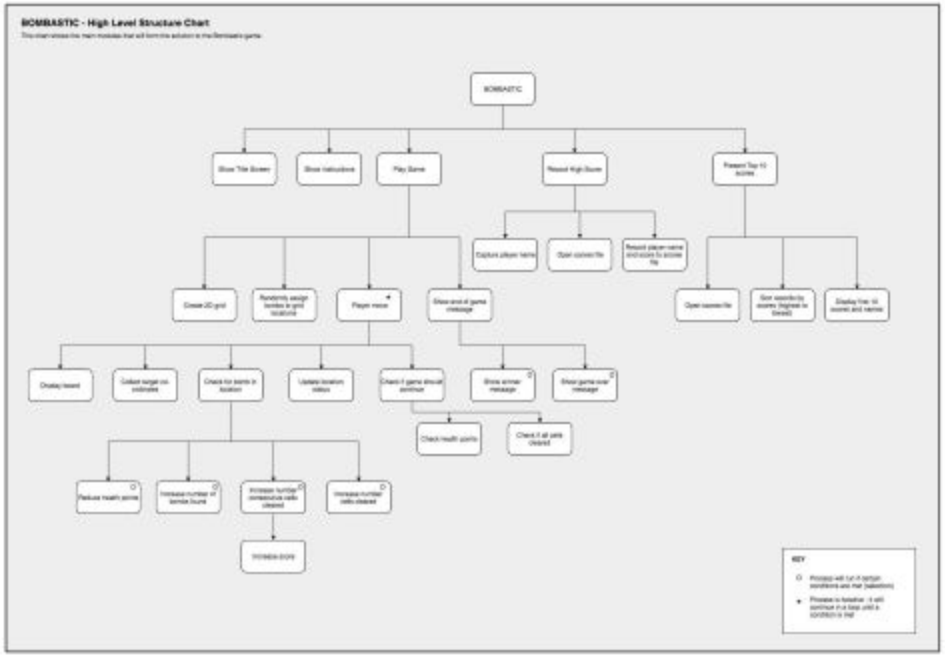
- **Money** has no real value (it's just paper, or maybe not even that!), but it represents the value of goods and services

Decomposition

- Decomposition means **breaking down** a **complex problem** into **smaller, more manageable parts**.
- Each smaller part can then be **solved individually**.
- Decomposition allows large teams to each take a part of a problem and work on it.

Structure charts

- Structure charts are used to break a large program down into the smaller parts that make it up.
- Each box represents a smaller problem to be solved. The lines show which bigger problem each box is a part of.
- Structure charts are a very useful tool when designing a solution to a big problem, such as a creating a computer game.



An example of a structure chart showing the parts that make up a computer game.

Algorithmic thinking

- Algorithmic thinking is a way of solving a problem by producing algorithms.
- An algorithm is a **reusable set of instructions** (or a series of steps or logical processes) **to solve a problem.**
- Algorithms can be written as a set of numbered steps to follow such as a cooking recipe or assembly instructions, however in computing they are presented as pseudocode or flow diagrams.

Algorithm building blocks

There are three main building blocks of all algorithms:

- **Sequence** - a series of steps that are to be completed one-by-one.
- **Selection** - steps that only occur if a certain condition is met.
- **Iteration** - steps that repeat, either for a fixed number of times or until a condition is met.

We use each of these algorithm structures when programming.

Applying Computational Thinking to solve problems

Computational Thinking is all about applying techniques from the world of Computer Science to solve problems, either in computing or in real life.

There are four main stages to solving any problem and different computational thinking techniques can be applied at each one:

- **Analyse** the problem to fully understand it through decomposition and abstraction of the problem.
- **Design** a solution to the problem using algorithmic thinking and abstraction.
- **Implement** the solution either by writing a program based on the algorithms or, if a real-world situation, follow the steps in the algorithm.
- **Test and evaluate the solution** to tests each of the outcomes from the initial computation thinking. For example, has each part identified through decomposition worked? Are the abstractions accurate enough? Do the algorithms work effectively and efficiently?

What is pseudocode?

```
INPUT num1
INPUT num2
IF num1 > num2 THEN
    OUTPUT "Yes"
ELSE
    OUTPUT "No"
```

This algorithm is written in **pseudocode**:

- Pseudocode is a way to write out algorithms using **code-like** statements
- It is **not** an actual programming language
- There is **no "correct" way** to write it
- It is used to **plan algorithms**, focussing on the **logic** and **steps** rather than language-specific **syntax**

Worked example: Writing an algorithm to solve a problem

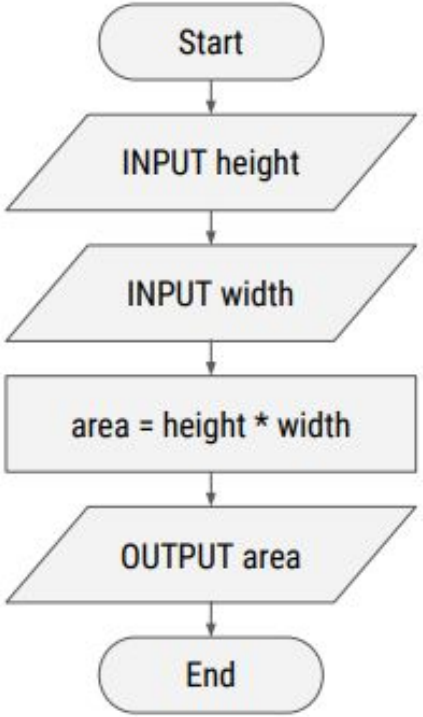
Implement the solution as an algorithm using pseudocode.

```
total_entered = 0
cost_of_drink = 65
while total_entered < cost_of_drink
        INPUT money
        total_entered = total_entered + money
        remaining_amount = cost_of_drink - total_entered
        OUTPUT "Amount remaining: " + remaining_amount
dispense_drink()
if total_entered > cost_of_drink then
        change_amount = cost_of_drink - total_entered
        give_change(change_amount)
```

This algorithm is written in **pseudocode** - "code-like" English statements that represent each step of the process.

Using flow diagrams to represent algorithms

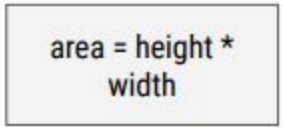
- Flow diagrams **visually represent** the steps that make an algorithm.
- A **standard set of shapes** are used to represent different types of step, such as running a sub-process or taking one path or another depending on a condition being met.
- The arrows in a flow diagram represent the **flow of control** through the algorithm.



Standard flow diagram shapes

You should be familiar with the following standard shapes used in flow diagrams.

Terminator - is used at the start and end of the flow diagram. You should include "Start", "End" or if the process is a function that returns a value then the shape should include "Return" followed by the value (or variable) being returned.

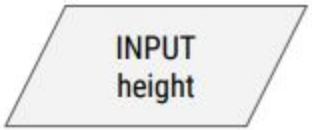


Process - this is used to represent a process. A short but clear description of the process should be written in the box. If you are describing several steps then you should use several process rectangles instead of one.

Sub-process - a rectangle with extra vertical lines on each side represents a whole other process that is to be run at this stage. This is very useful for a complex algorithms that are made up of several smaller processes, each of which is described in its own flow diagram. Include the name of the sub-process within the rectangle.

Standard flow diagram shapes

You should be familiar with the following standard shapes used in flow diagrams.



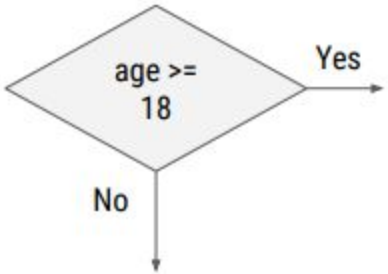
Input/Output - a parallelogram is used to represent a stage in the algorithm where data is either input by a user or output to a display. You should write either "INPUT" or "OUTPUT" within the shape to make clear which process is being represented.

If data is being input, you should also state the name of a variable that will be used to store the value.

If data is being output you should include the name of the variable that contains the output value or the message that is to be displayed.

Standard flow diagram shapes

You should be familiar with the following standard shapes used in flow diagrams.

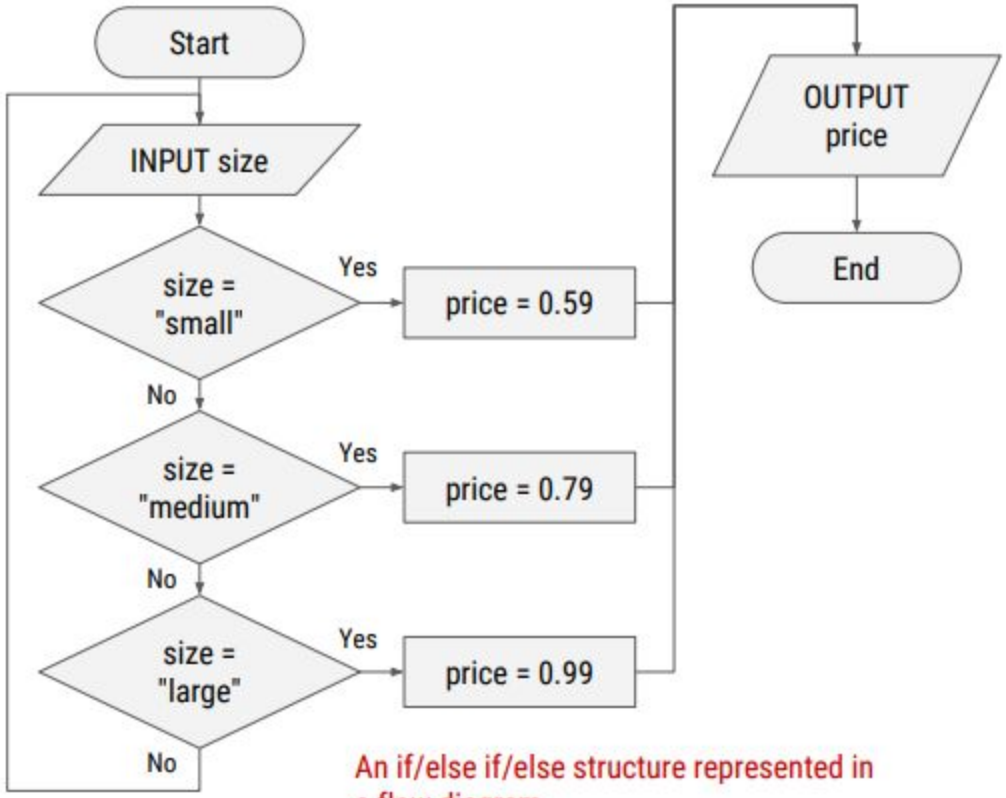


Decision - a diamond shape is used to represent a point of decision (or selection). You should include the condition that the decision depends on within the shape. Again, try to keep this short but specific.

A decision diamond must have **two labelled lines** coming out of it to show what sequence of steps should follow whether the condition has been met or not. These lines should be labelled with either "Yes" and "No" or "True" and "False".

It doesn't matter whether the "yes/true" or "no/false" labels go on the line coming down or to the side of the diamond.

Representing if/else if/else in a flow diagram



An if/else if/else structure represented in a flow diagram

An if/else if/else selection structure is used when multiple conditions are to be tested resulting in a different thing happening each time.

if/else if/else structures can be represented in flow diagrams using multiple decision diamonds flowing into each other from their "No" branches.