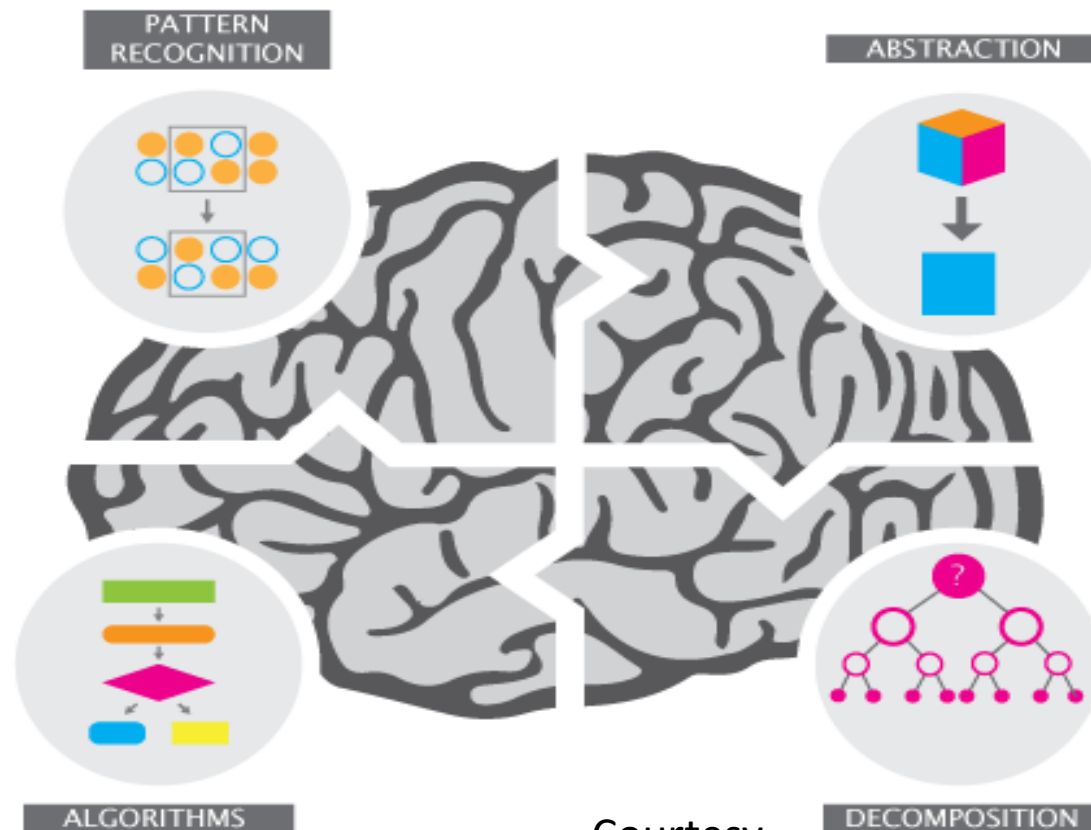


Computational Thinking

The four techniques to computational thinking:

- **decomposition** – dividing a complex problem or system into smaller, more manageable parts
- **pattern recognition** – looking for similarities between and within each part
- **abstraction** – focusing on the important information only, ignoring irrelevant detail and focussing on solving each part in turn so that the combined parts solve the problem
- **algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem



Courtesy

<https://www.bbc.co.uk/bitesize/guides/zp92mp3/revision/1>

Decompose: 1.1 Understand the Problem

- Underline words that describe what is to be done eg, find the avg, calculate tax/gst
- Try rephrase the problem in your own words
- List all the facts
- [Thinking computationally - Introduction to computational thinking - KS3 Computer Science Revision - BBC Bitesize](#)

2.2 Decompose the problem

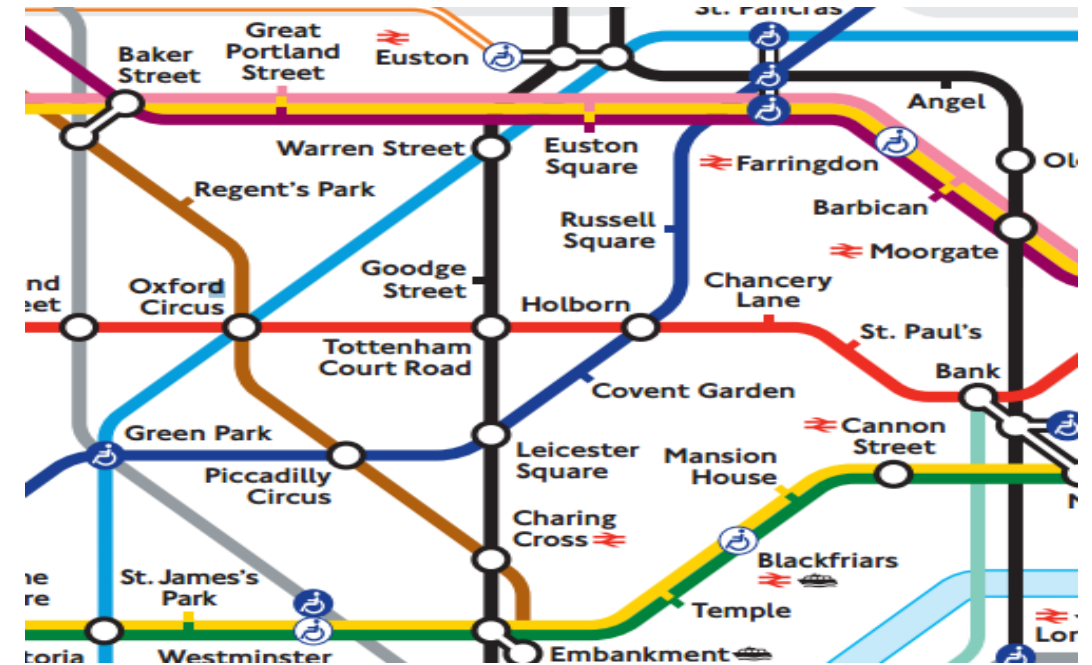
- [Decomposition \(schoolcoders.com\)](https://schoolcoders.com)
- IPO

2. Abstraction

It doesn't show you anything about the world above ground. The stations aren't even in the correct places, geographically. The distance between two stations on the map does not reflect the real distance between them, most lines are straight (even though they might have bends), and in many cases the tracks aren't going in the exact direction shown on the map.

But that doesn't matter, because when you are underground all you really need to know is which line you need to take, and in which direction. The map tells you that very clearly, with minimum distractions.

[Abstraction \(schoolcoders.com\)](http://schoolcoders.com)



Another example:
Computer case Manufacturer vs Motherboard
Manufacturer.

3. Pattern Recognition

- Whenever you tackle any kind of programming problem, you will probably notice patterns. You might see certain similarities between different cases, or maybe you will notice certain cases which never occur. You might see patterns in data, or you might notice a similarity between the problem you are trying to solve and another, totally different problem which already has a solution. Sometimes the pattern will be obvious, sometimes you need to look at things in a slightly different way before you see it.
- Pattern recognitions is the most creative, and arguably the most important, part of computational thinking. It is often the first step in discovering a new algorithm




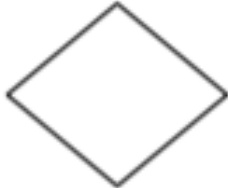

4. Algorithm

- An algorithm is a step-by-step procedure for solving a particular problem.
- We often use [decomposition](#) and [abstraction](#) to analyse the problem. We may try to [recognise patterns](#) to find existing techniques that might be applied to the problem.
- **Computer algorithms**
- When we design an algorithm for a computer to use, we must specify each step very precisely, so that it can be implemented as a computer program.
- To define an algorithm we usually need to specify:
 - The inputs
 - The outputs
 - The steps of the algorithm, usually in pseudocode or as a flowchart.

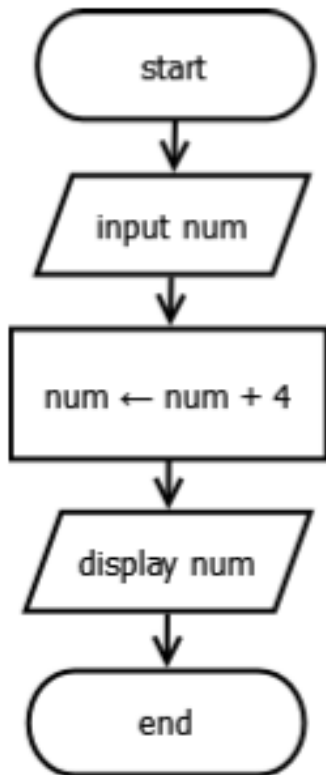
4.1 Data structure

- Main variables used to store data for IPO and are related to algorithms. The sometimes the algorithm determine the variables and visa versa. See slide 20 on Steps of Computational Thinking in Practice example 2
- Temp variables will not store data throughout program

4.2 Flow Chart

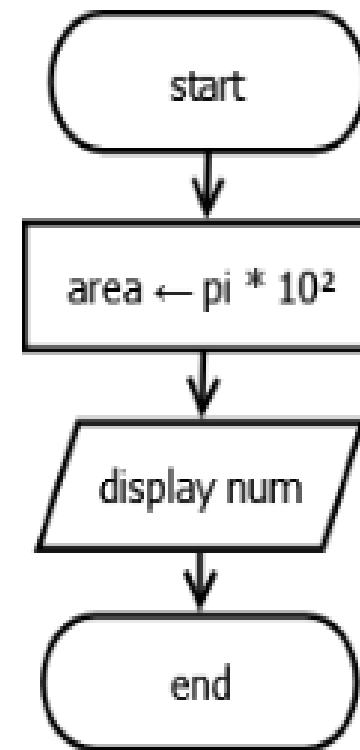
Symbol	Purpose
	Start or End
	Statement
	Input or Output
	Decision (to be covered later)
	Connecting arrow between processes

4.2 Flowchart Examples



Code:

```
int num;
Console.WriteLine("please
enter a num:");
num = int.Parse(Console.ReadLine());
num = num + 4;
// num+=4;
Console.WriteLine(num);
```



Code:

```
area= Math.PI*Math.Pow(10,
2);
Console.WriteLine(num);
```

4.3 Pseudocode

The following rules can be used to write an algorithm in pseudocode.

- start the code with the word "begin" and finish with the word "end". Indent all instructions between "begin" and "end".
- one line of pseudocode code is similar to a line of Java code
- an assignment statement is an arrow \leftarrow pointing left, not an equals = sign
- no semi colons at the end of a statement
- variable types vary from one language to another, so do not include any variable declarations
- do not use any method names that only belong to one language. For example, an algorithm will not use `bug.move()` but rather say something like move bug 1 block.
- mathematical symbols such as $<$, $>$, $=$, \neq , \geq and \leq are universal in any language and can be used
- use the word **input** when entering data from the keyboard
- use **display** when displaying data on the screen

	Code:
begin	<code>int num;</code>
input num	<code>Console.WriteLine("please enter a num:");</code>
$\text{num} \leftarrow \text{num} + 4$	<code>num = int.Parse(Console.ReadLine());</code>
display num	<code>num = num + 4;</code>
end	<code>Console.WriteLine(num);</code>

	Code:
begin	
$\text{area} \leftarrow \text{pi} * 10^2$	<code>area= Math.PI*Math.Pow(10, 2);</code>
display area	<code>Console.WriteLine(num);</code>
end	

Best Practice for coding

- User Friendly: does not mean has to have nice greetings etc but must be:
 - easy to use
 - with clear prompts of what data to input. Less chance of user typing in the wrong datatype and avoid runtime errors
 - Lined up outputs with meaningful messages of what is being displayed especially error messages .
- Simple GUI
- Readability: for another coder to understand not same as User-friendly

Readability

1. Use descriptive variable names

If a variable is going to store a sum call it **sum** (or **total**). It can be misleading when a variable determines the average but is called sum.

2. Place one statement on a line

Code is easier to read if there is only one statement on a line including brackets.

1. Indent your code correctly

Any code between an opening and closing curly bracket must be indented. This will be particularly helpful to identify errors when you use loops and selection statements later on. Your IDE may provide a function to indent code.

2. Include comments

If the variable name does not make the code easy to understand, include comments. At this stage, the method names are fairly explanatory, but if you are doing something new that you doubt you will understand later, add in some explanations in the form of comments.

3. Separate code

Use blank lines to separate sections of code. Try to group code that performs a similar function together. For example, use sections to: declare variables; perform calculations and produce output. Later on, we will write our own methods to break the code into sections.

4. Follow conventions

- By convention, identifiers in Java start with lowercase letters.
- The identifier should reflect the contents of the variable.

Your teacher may have other conventions that should be followed. Use them as it makes marking quicker for your teacher and it's easier to understand your friends' programs.

Errors

- a **syntax error** is an error picked up by the compiler, such as a semi colon missing. It is the lowest level of errors and is the easiest to find as your IDE should give some kind of error message.
- a **run time error** is when a program compiles but does not run, for example trying to determine the square root of a negative number. Run time errors are one level up from syntax errors. When the program unexpectedly stops, your IDE should indicate where in the code the program halted and give an error message.
- **logical errors** occur when there is an error in the logic of a program statement or a sequence of statements are in the incorrect order. The program compiles and runs but does not do what you wanted. There is no error message making logical errors more difficult to find.

Write down the type of error (syntax/run time/logical) for each scenario below:

- 1.1. The program produces an error "missing semi colon".
- 1.2. The user enters 5.3 for an integer value.
- 1.3. The program determines the sum and not the average of four numbers.

```
public class Subtract
{
    public static void main ()
    {
        String n1 = 4;
        String n2 = 0;
        n3 = n1 / n2;
        System.out.println(n3);
    }
}
```

- 2.1.** Identify the syntax error(s).
- 2.2.** Identify the run time error(s).
- 2.3.** Explain why the program is not readable.
- 2.4.** How could the program be altered to be more user-friendly?

Steps of Computational Thinking in Practice

Problem Statement Given: Calculate Sales Price

A shopkeeper needs to determine the selling price of a T-shirt after 5% is deducted and VAT is included. We will call this problem **CalcPrice**.

Step 1. Abstraction

We need to determine what is relevant to the problem. The problem we need to solve is to take a price and work out the sales price after discount and VAT have been calculated.

The shopkeeper and T-shirt are not relevant. The problem can be generalised to work out the sales price for any item. We can assume that the shopkeeper will run the program, but it can be run by any user.

Step 2.1 Decomposition

Input	price
Processing	discount is 5% of the price VAT is 15% of the price after the discount has been taken off
Output	result

- Step 2.2 Data Structure:

Once you have analysed a problem into IPO, you need to select the main and temporary variables and choose an appropriate type for each variable. As data structures become more complex, the choice of data structure will have consequences and implications for how the problem can be solved.

In the above example, we will need to store the **price**, **discount**, **VAT** and the final **result**. Since the problem deals with money, double will be a suitable type.

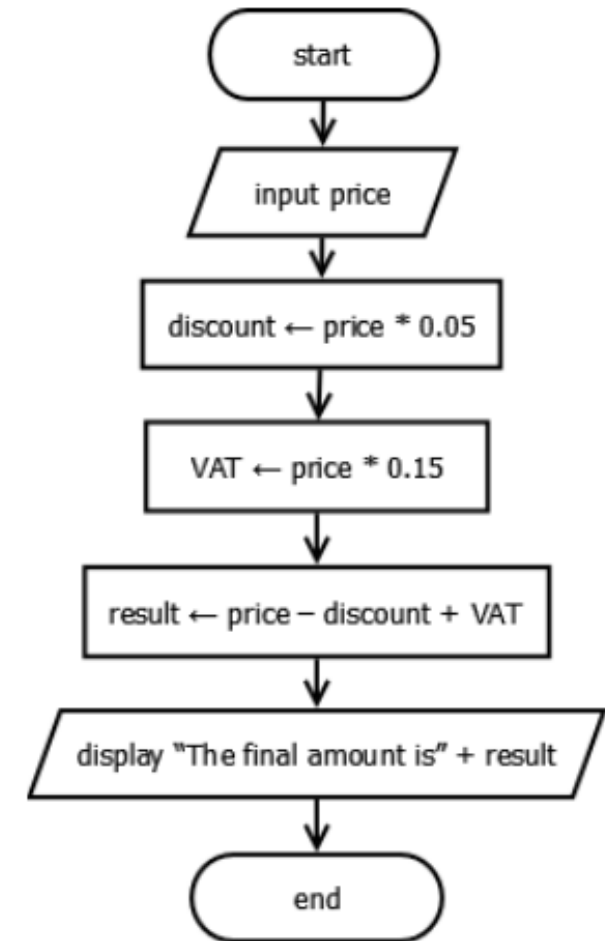
The flow chart for the **CalcPrice** problem using Input, Processing and Output:

- Step 3 Write Algorithm:

Represent the solution in either pseudocode or a flow chart.

The pseudocode for the **CalcPrice** problem using the decomposed Input, Processing and Output:

Input	input price
Processing	$\text{discount} \leftarrow \text{price} * 0.05$ $\text{VAT} \leftarrow \text{price} * 0.15$ $\text{result} \leftarrow \text{price} - \text{discount} + \text{VAT}$
Output	display "The final amount is" + result



Step 4:

- Code:

Steps of Computational Thinking in Practice

example 2

Problem Statement Given: **Swop two numbers**

Zinzi wants to write a program to swop two numbers around. The user will input the two numbers called **num1** and **num2**. The program will place **num1**'s value in **num2** and vice versa, then output the results.

Step 1. Abstraction

Make sure you understand what is required and extract the relevant details. The program is required to input two numbers, swop them and display both. What is not relevant is the person who needs the program, Zinzi.

Step 2.1 Decomposition

Input	Two numbers num1 and num2
Processing	Swop the numbers around
Output	num1 and num2 with swopped values

- Step 3 Write Algorithm:

To swop two numbers, we cannot assign **num1** to **num2** and **num2** to **num1**.

$\text{num1} \leftarrow \text{num2}$

$\text{num2} \leftarrow \text{num1}$

We can show this by use of a small trace table assuming **num1** is 5 and **num2** is 10.

Algorithm	num1	num2
$\text{num1} \leftarrow \text{num2}$	5 10	10
$\text{num2} \leftarrow \text{num1}$		10

When **num1** is assigned **num2**'s value **num1**'s value is **overwritten** and lost. We need to keep a copy of **num1**'s value in a variable called **temp** before we perform $\text{num1} \leftarrow \text{num2}$. Then **num2** can get its value from temp. The code will be:

$\text{temp} \leftarrow \text{num1}$

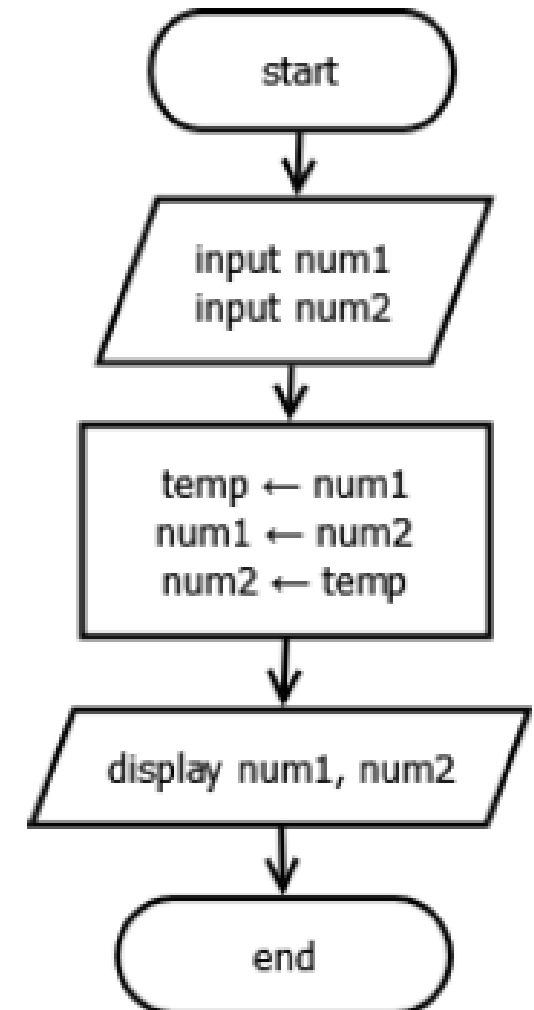
$\text{num1} \leftarrow \text{num2}$

$\text{num2} \leftarrow \text{temp}$

The IPO table will become:

Input	num1, num2
Processing	$\text{temp} \leftarrow \text{num1}$ $\text{num1} \leftarrow \text{num2}$ $\text{num2} \leftarrow \text{temp}$
Output	A message with num1, num2

Data Structure: We Need to use temp variable, num1 and num2. As mentioned earlier sometimes the Algorithm highlights other data structures that we might need



Step 4:

- Code: