**Computational Thinking**

**Overview**

Computational thinking is an approach to solving problems using concepts and ideas from computer science and expressing solutions to those problems so that they can be run on a computer. As computing becomes more and more prevalent in all aspects of modern society -- not just in software development and engineering, but in business, the humanities, and even everyday life -- understanding how to use computational thinking to solve real-world problems is a key skill in the 21st century. Computational thinking is built on four pillars: decomposition, pattern recognition, abstraction, and algorithm design. This course introduces you to the four pillars of computational thinking and shows how they can be applied as part of the problem-solving process.

Before computers can be used to solve a problem, the problem itself and the ways in which it could be resolved must be understood. Computational thinking techniques help with these tasks.

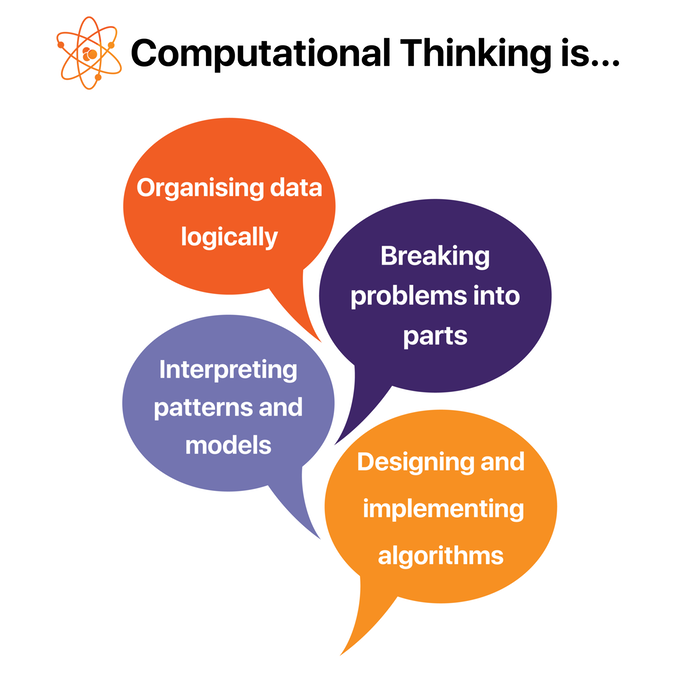
**What is computational thinking? -1**

Answering this question is actually quite challenging. Proponents of computational thinking (CT) have until very recently spent a lot of time debating over how to define it.

CT is also an idea that’s both new and old. It’s new in the sense that the subject suddenly became a hotly debated topic in 2006 after **Wing’s** talk (Wing, 2006). However, many of its core ideas have already been discussed for several decades, and along the way, people have packaged them up in different ways. For example, as far back as 1980, **Seymour Papert** of the Massachusetts Institute of Technology pioneered a technique he called 'procedural thinking’ (Papert, 1980). It shared many ideas with what we now think of as CT. Using procedural thinking, Papert aimed to give students a method for solving problems using computers as tools. The idea was that students would learn how to create algorithmic solutions that a computer could then carry out; for this, he used the Logo programming language. Papert’s writings have inspired much in CT, although CT has diverged from this original idea in some respects.

Nevertheless, during the 10 years following Wing’s talk, a number of succinct definitions were attempted. Below you will see some of them.

* 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. *(Wing, 2014)*
* The mental activity for abstracting problems and formulating solutions that can be automated. *(Yadav et al., 2012)*
* The process of recognizing 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. *(Furber, 2012)*
* A mental orientation to formulating problems as conversions of some input to output and looking for algorithms to perform the conversions. Today the term has been expanded to include thinking with many levels of abstractions, use of mathematics to develop algorithms, and examining how well a solution scales across different sizes of problems. *(Denning, 2009)*
* [Teaching CT is teaching] how to think like an economist, a physicist, an artist, and to understand how to use computation to solve their problems, to create, and to discover new questions that can fruitfully be explored. *(Hemmendinger, 2010)*



***What is computational thinking? -2***

* Computational Thinking is the must step between having a problem and having a solution to that problem. Computational Thinking teaches an approach to problem-solving where the ultimate aim is to provide a solution whose form means it is ready to be programmed into a computer.



* Programming is much like bricklaying: you need to know a little bit about it to be able to build something, and master artisans can have long and detailed conversations about pointing and the relative advantages of variations on herringbone layouts, but fundamentally a wall is a wall.



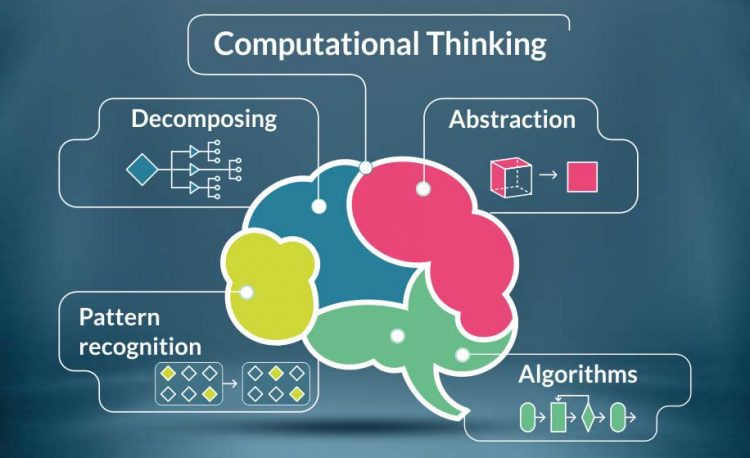
* Bricklaying isn't that interesting: architecture is interesting. Just as architecture is about understanding people's requirements and seeing how a particularly shaped pile of bricks could address them, computational thinking is about understanding a problem and seeing how a particularly shaped pile of program statements could address it.



**Pillars of Computational Thinking**

Computational Thinking is made up of four parts:

1. **Decomposition:** Breaking down data, processes, or problems into smaller, manageable parts
2. **Pattern Recognition:** Observing patterns, trends, and regularities in data
3. **Abstraction:** Focusing on important parts only, ignoring irrelevant details
4. **Algorithm Design:** Developing the step by step instructions for solving this and similar problems



You will learn the details of each step in the following lessons.

**Computational thinking in practice**

A complex problem is one that, at first glance, we don't know how to solve easily or to understand.

Computational thinking involves taking that complex problem and breaking it down into a series of small, more manageable problems (decomposition). Each of these smaller problems can then be looked at individually, considering how similar problems have been solved previously (pattern recognition) and focusing only on the important details while ignoring irrelevant information (abstraction). Next, simple steps or rules to solve each of the smaller problems can be designed (algorithm design).

Finally, these simple steps or rules are used to program a computer to help solve the complex problem in the best way.

**Thinking computationally**

Thinking computationally is not programming. It is not even thinking like a computer, as computers do not, and cannot, think.

Simply put, programming tells a computer what to do and how to do it. Computational thinking enables you to work out exactly what to tell the computer to do.

For example, if you agree to meet your friends somewhere you have never been before, you would probably plan your route before you step out of your house. You might consider the routes available and which route is ‘best’ - this might be the route that is the shortest, the quickest, or the one which goes past your favorite shop on the way. You'd then follow the step-by-step directions to get there. In this case, the planning part is like computational thinking, and following the directions is like programming.

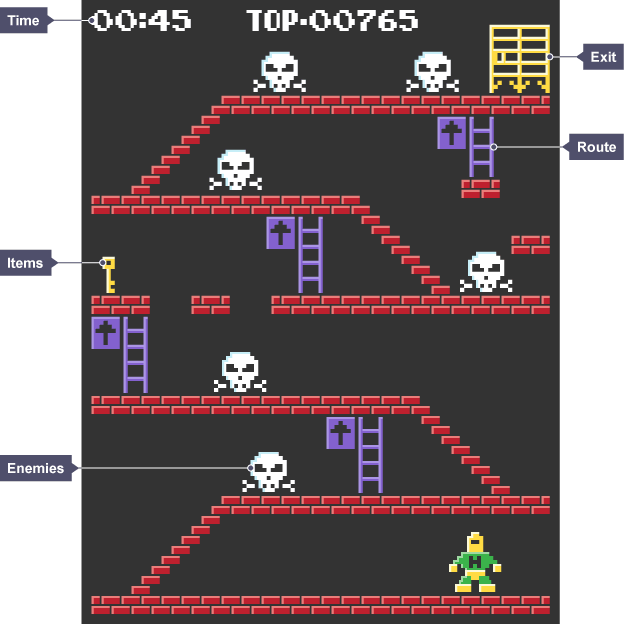
Being able to turn a complex problem into one we can easily understand is a skill that is extremely useful. In fact, it's a skill you already have and probably use every day.

Another example might occur when playing a videogame. Depending on the game, in order to complete a level you would need to know:

* what items you need to collect, how you can collect them, and how long you have in which to collect them
* where the exit is and the best route to reach it in the quickest time possible
* what kinds of enemies there are and their weak points

From these details, you can work out a strategy for completing the level in the most efficient way.

If you were to create your own computer game, these are exactly the types of questions you would need to think about and answer before you were able to program your game.



**Examples for Using Computational Thinking in Real Life**

* **Pipelining a graduation ceremony**

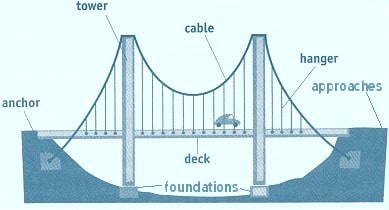
Dean Randy Bryant was pondering how to make the diploma ceremony at commencement go faster. By careful placement of where individuals stood, he designed an efficient pipeline so that upon the reading of each graduate’s name and honors by Assistant Dean Mark Stehlik, each person could receive his or her diploma, then get a handshake or hug from Mark, and then get his or her picture taken. This pipeline allowed a steady stream of students to march across the stage (though a pipeline stall occurred whenever the graduate’s cap would topple while getting a hug from Mark)



* **Designing earthquake-resistant bridges.**

In a Science class, students applied computational thinking, physics, and engineering design to build earthquake-resistant bridges. The unit started with understanding the function of bridges and the different types. Students then moved to study earthquakes and the impact of their forces.

In order to design an earthquake-resistant bridge, students applied both design thinking and computational thinking. Computational thinking enabled students to analyze a variety of bridge models to find patterns in their structure (pattern recognition) and abstract from these the important elements needed in a functional design (abstraction). As they tested the different prototypes, computational thinking allowed them to collect data and find opportunities to improve the structure.



* **The leaky pipe**

Here is another example using the computational thinking approach in a real-life problem.

Anna Shipman kept having a flood in her flat. Various plumbers came and went, and did something — until the next flood. But then someone came along who might have been a software engineer or a programmer in a previous life. He systematically and methodically tested each element in the piping and guttering that could be the source of the leak. In other words, he was applying the principle of decomposition.

The conclusion that Anna comes to is interesting: “Everything would be better if it was a bit more like software development. Craftsmen should be more like software developers.”



Formun Üstü

Formun Altı

**Practice: Finding The Shortest Path**

**Traveling**

You and your family are planning a road trip through all of the capitals in the region and you have the job of figuring out the shortest route possible to save on the cost of fuel.

Now that you have had a chance to think about a few options, now think about how would you explain the route you suggest your family to take? Will you provide a list of cities or is there a guiding principle like start at the west and follow it to the east?

Here are some questions to consider:

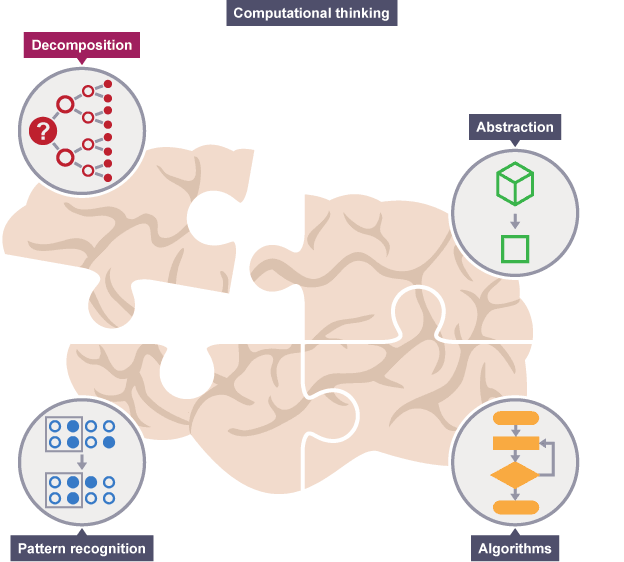
1. Does it matter which city you start from?
2. Would the strategy you used to develop your route (longitudinal, latitudinal, etc) apply to any country or does it depend? Try switching to another country and see if it works as well there.
3. What are some of the reasons this problem is difficult?

This type of problem is traditionally known as the traveling salesperson problem, where the challenge is to go through each city only once and return to the city of origin. Even though this type of problem has not been optimally solved by the community, the act of investigating the problem and developing algorithms to try and solve it (even if in a suboptimal way) has been used to improve the efficiency of many other related problems. Route planning and searching through data, like DNA and web pages, are two similar types of problems.

**First step in computational thinking: Decomposition**

**What is decomposition?**

Decomposition is one of the four pillars of Computer Science. It involves 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.



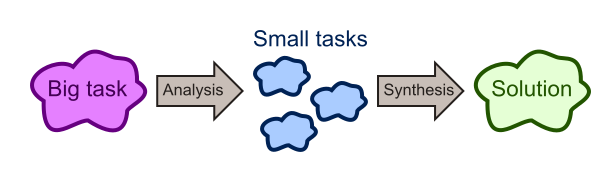
For example, if we are developing a game, different people can design and create the different levels independently provided key aspects are agreed in advance. Through decomposition of the original task, each part can be developed and integrated later in the process. A simple arcade level might also be decomposed into several parts, such as the life-like motion of a character, scrolling the background and setting the rules about how characters interact.

**Why is decomposition important?**

Decomposition is used for more than just project planning. It’s handy in understanding complex concepts, ones that are too big or cumbersome to pin down. When a concept is decomposed, the study of its constituent parts can offer novel ways to approach the subject. You can optimize processes, streamline designs or even come up with totally new ideas.

Think of a cake. It’s a single food item comprised of multiple ingredients: eggs, flour, sugar, butter. The ingredient list is a sort of decomposition. Understanding the parts of a cake allows you to tweak the recipe once you understand the fundamentals of baking.

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.

**Decomposing creating an app**

Imagine that you want to create your first app. This is a complex problem - there are lots of things to consider.

***Question***

How would you decompose the task of creating an app?

A : To decompose this task, you would need to know the answer to a series of smaller problems:

* what kind of app you want to create
* what your app will look like
* who the target audience for your app is
* what your graphics will look like
* what audio you will include
* what software you will use to build your app
* how the user will navigate your app
* how you will test your app
* where you will sell your app

This list has broken down the complex problem of creating an app into much simpler problems that can now be worked out. You may also be able to get other people to help you with different individual parts of the app. For example, you may have a friend who can create the graphics, while another will be your tester.

**An Example to Decomposition: Divide and Conquer**

This video introduces the idea of "divide and conquer" using a fictitious but serious problem - a pair of dirty socks have accidentally been wrapped in one of 1024 presents that Santa is about to deliver, and he needs to figure out which one to avoid a child getting a nasty surprise. Before you play the video below, think for a few minutes and try to put your solution to this problem.

The solution in the story points out that when there are 1024 boxes to test, instead of having to open all of them until the socks are found, one half can be eliminated at a time, and repeatedly halving the problem very quickly narrows it down to one box (the size of the problem starts at 1024, then with one weighing there are 512 boxes, then 256, 128, 64, 32, 16, 8, 4, 2 and 1.) This idea comes up frequently in the design of fast computer algorithms.

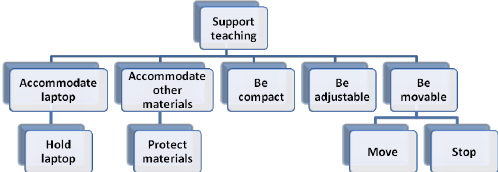
Here is a question for you: Is there any other solution that is faster than the young elf's solution?

*(Answer: Yes there is. Try to figure it out.)*

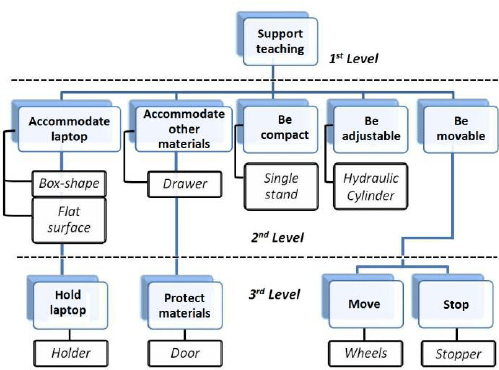
Since divide and conquer approach makes problems easier (and faster) to solve why not divide more. Divide 1024 boxes into three groups (341-341-342) instead of two. Weigh the two groups that have the same number of boxes. Then divide the heavier group into three, and if both are the same weight then divide the third group into three and so on until you find the socks.

**Decomposition: Tree Structures**

The structure of the problem can be represented by the relationships between functions and subfunctions in a functional tree, which consists of blocks connected by branches (as shown below). A sub-function is at a level lower than its parent function.

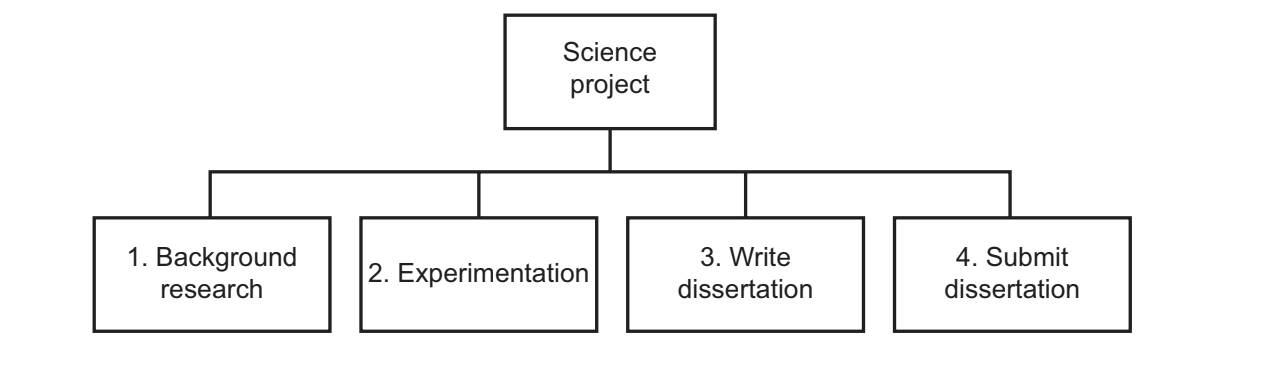


The tree-structured function decomposition in the below figure contains the features of decomposability and abstraction hierarchies of the structure of the problem. First, the main function is decomposed into several sub-functions. Second, the main function, sub-functions, and solutions are connected in a hierarchical structure. The type of information (e.g., behavior, functions, and structure) and the phases of design processes (from problem structuring to concept generation) are included in the abstraction hierarchy.

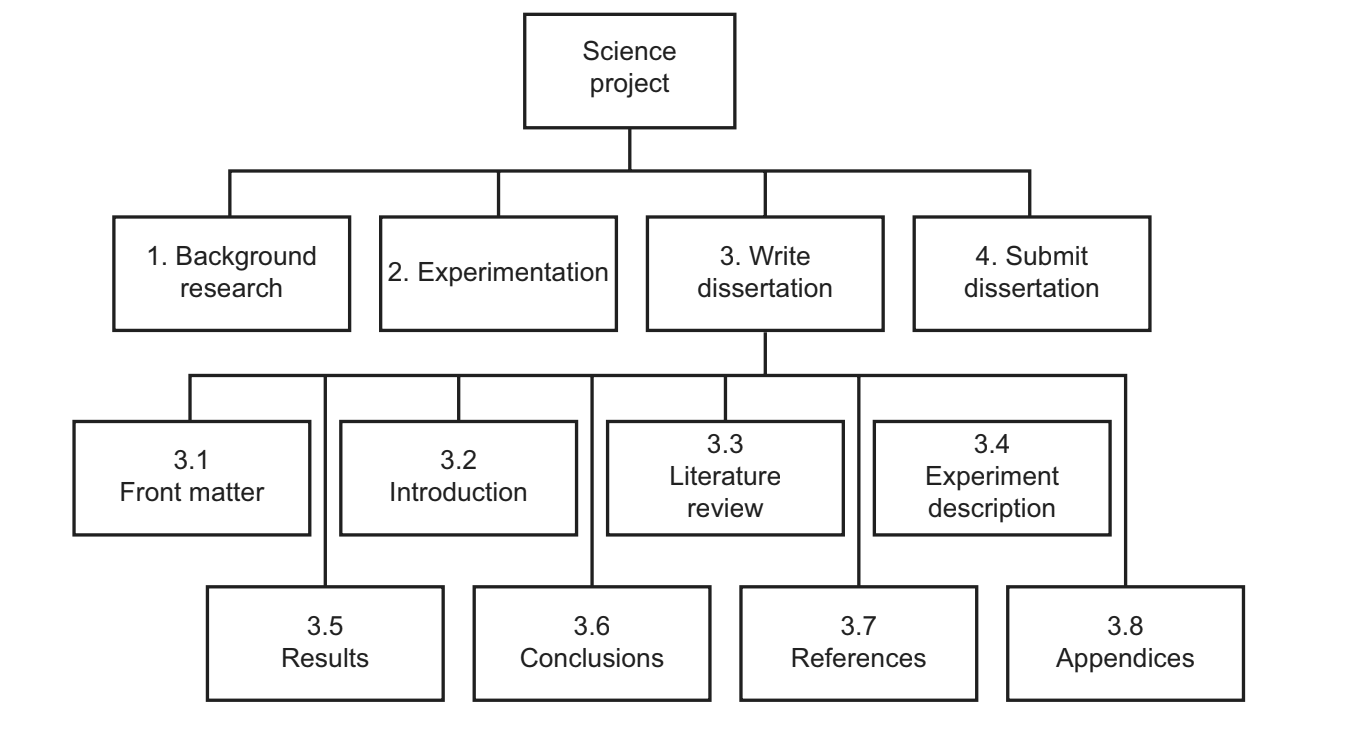


By applying decomposition, you aim to end up with a number of sub-problems that can be understood and solved individually. This may require you to apply the process recursively (see above). That is to say, the problem is re-formed as a series of smaller problems that, while simpler, might be still too complex, in which case they too need breaking down, and so on. Visually this gives the problem definition a tree structure.

Let’s look at an example.



One task that many of us encounter during our education is the production of a dissertation. Writing an academic work some tens of thousands of words in length can inspire fear, panic or even despair. By itself, one monolithic task entitled ‘write a dissertation’ is unmanageable, but by decomposing the problem you reduce it into smaller parts, each of which possesses far fewer details. You can then focus your attention on each part and deal with it much more effectively. By solving each sub-problem, you end up solving the overall problem.



This has decomposed the task somewhat, but some sub-tasks maybe still too big. Those tasks should be broken down further by applying the same decomposition process. For example, the front matter of a dissertation is made up of several parts, such as:

3.1.1 write a title page;

3.1.2 write copyright section;

3.1.3 write abstract;

3.1.4 write contents section;

3.1.5 write a list of figures section.

This time, all the resulting tasks are now manageable. None of them will contain more than a few hundred words, and some can even be done automatically by the word processor.

At this point, you can go back up the tree and find a different task that requires further decomposition. This process needs to be repeated until all the leaves of the resulting tree are problems you can solve individually.

**Decomposition: Smiley Face Example**

Instructing a computer to draw a complex image is a multi-step process, so it can’t be solved in one go. However, by decomposing an image into a series of simple shapes, you break the problem up into a series of simple sub-problems that can be solved individually.

As a simple example: a smiley face. It’s one image, but it actually contains several component shapes:

* A large circle for the face.
* A medium circle for the outer eye.
* A small, filled circle for the inner eye.
* An arc for the smile.

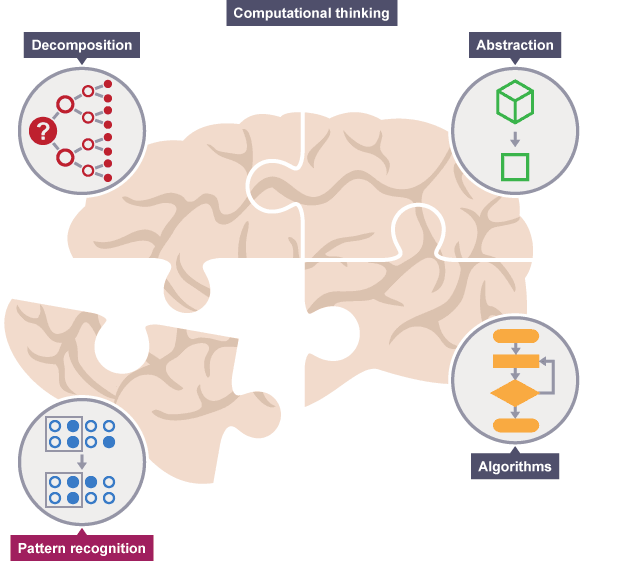


**Pattern Recognition**

**What is Pattern Recognition?**

When we decompose a complex problem we often find patterns among the smaller problems we create. The patterns are similarities or characteristics that some of the problems share.

Pattern recognition is one of the four pillars of Computational Thinking. It involves finding the similarities or patterns among small, decomposed problems that can help us solve more complex problems more efficiently.



**What are patterns?**

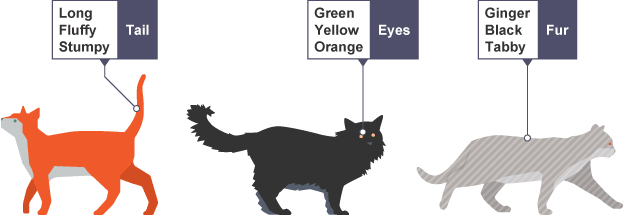
Imagine that we want to draw a series of cats.

All cats share common characteristics. Among other things, they all have eyes, tails, and fur. They also like to eat fish and make meowing sounds.

Because we know that all cats have eyes, tails, and fur, we can make a good attempt at drawing a cat, simply by including these common characteristics.

In computational thinking, these characteristics are known as patterns. Once we know how to describe one cat we can describe others, simply by following this pattern. The only things that are different are the specifics:

* one cat may have green eyes, a long tail, and black fur
* another cat may have yellow eyes, a short tail, and striped fur



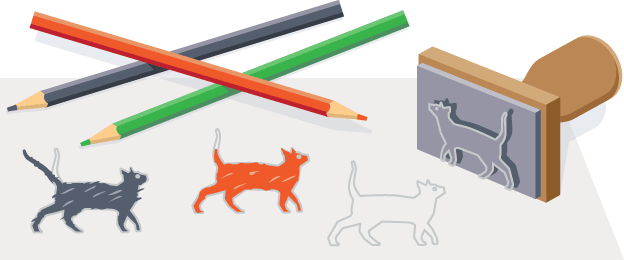
**Why do we need to look for patterns?**

Finding patterns is extremely important. Patterns make our tasks 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.

If we want to draw a number of cats, finding a pattern to describe cats in general, eg they all have eyes, tails, and fur, makes this task quicker and easier.

We know that all cats follow this pattern, so we don’t have to stop each time we start to draw a new cat to work this out. From the patterns we know cats follow, we can quickly draw several cats.



**What happens when we don’t look for patterns?**

Suppose we hadn’t looked for patterns in cats. Each time we wanted to draw a cat, we would have to stop and work out what a cat looked like. This would slow us down.

We could still draw our cats - and they would look like cats - but each cat would take far longer to draw. This would be very inefficient, and a poor way to go about solving the cat-drawing task.

In addition, if we don’t look for patterns we might not realize that all cats have eyes, tails, and fur. When drawn, our cats might not even look like cats. In this case, because we didn’t recognize the pattern, we would be solving the problem incorrectly.

**Recognising patterns**

To find patterns in problems we look for things that are the same (or very similar) in each problem. It may turn out that no common characteristics exist among problems, but we should still look.

Patterns exist among different problems and within individual problems. We need to look for both.

**Patterns among different problems**

To find patterns among problems we look for things that are the same (or very similar) for each problem.

For example, decomposing the task of baking a cake would highlight the need for us to know the solutions to a series of smaller problems:

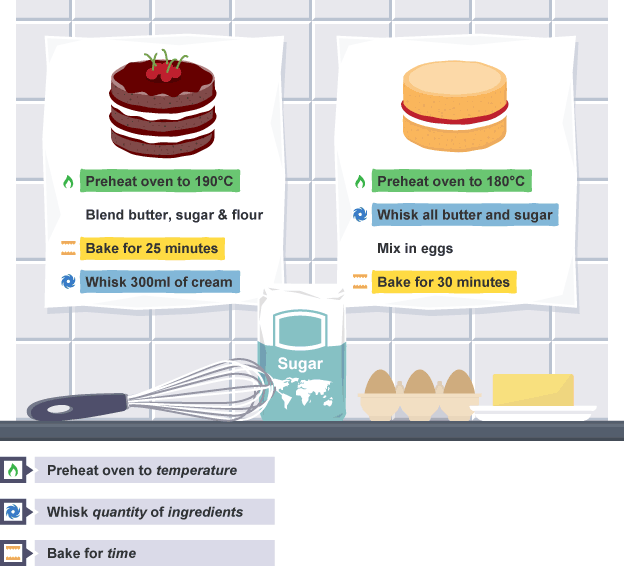
* what kind of cake we want to bake
* what ingredients we need and how much of each
* how many people we want to bake the cake for
* how long we need to bake the cake for
* when we need to add each ingredient
* what equipment we need

Once we know how to bake one particular type of cake, we can see that baking another type of cake is not that different - because patterns exist.

For example:

* each cake will need a precise quantity of specific ingredients
* ingredients will get added at a specific time
* each cake will bake for a specific period of time

Once we have the patterns identified, we can work on common solutions between the problems.



**Patterns within problems**

Patterns may also exist within the smaller problems we have decomposed to.

If we look at baking a cake, we can find patterns within the smaller problems, too. For example, for ‘each cake will need a precise quantity of specific ingredients’, each ingredient needs:

* identifying (naming)
* a specific measurement

Once we know how to identify each ingredient and its amount, we can apply that pattern to all ingredients. Again, all that changes is the specifics.

Example: The Pattern in Drawing a Smiley Face.

Below is a pattern to draw a smiley face.

1. Draw circle with radius 30 at position 50,50 with line thickness 2.

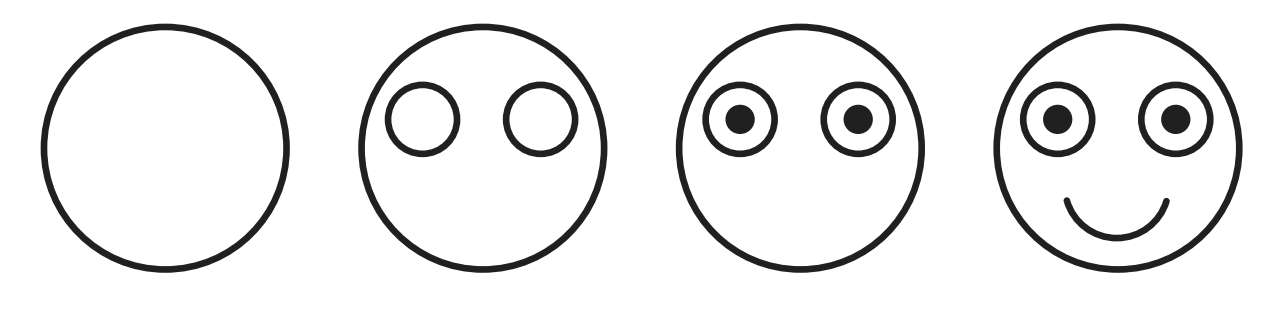
2. Draw circle with radius 6 at position 40,40 with line thickness 1.

3. Draw circle with radius 3 at position 40,40 filled black.

4. Draw circle with radius 6 at position 60,40 with line thickness 1.

5. Draw circle with radius 3 at position 60,40 filled black.

6. Draw a red line from position 30,70 to position 70,70 with line thickness 1.



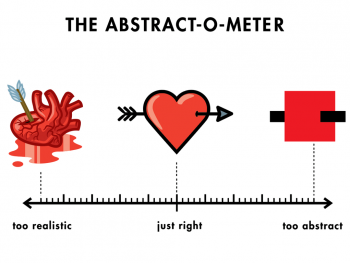
In this example, the patterns across instructions should be fairly simple to spot. The emergence of a pattern provides an opportunity for improvement. Instead of them being disjointed and separated, those parts making up the pattern can usually be brought together and solved using a common approach. In other words, you can take some separate, but similar concepts, and generalize them into a single concept. As a result, your solution becomes simpler because it contains fewer distinct concepts, and it becomes more powerful because you can reuse it in other situations and solutions.

**Abstraction**

A Brief Explanation of Abstraction

Abstraction is a way of expressing an idea in a specific context while at the same time suppressing details irrelevant in that context. Abstraction is a key feature of both computer science and computational thinking. Some have gone so far as to describe computer science as ‘the automation of abstraction’ (Wing, 2014).

The reasoning behind this goes right to the core of what programmers and computer scientists are trying to do; that is, solve real-world problems using computers. They can’t magically transport the real world into the computer; instead, they have to describe the real world to the computer. But the real world is messy, filled with lots of noise and endless details. We can’t describe the world in its entirety. There’s too much information and sometimes we don’t even fully understand the way it works. Instead, we create models of the real world and then reason about the problem via these models. Once we achieve a sufficient level of understanding, we teach the computer how to use these models (i.e. program it).



In computational thinking, when we **decompose** problems, we then **look for patterns** among and within the smaller problems that make up the complex problem.

Abstraction is the process of filtering out – ignoring - the characteristics of patterns that we don't need in order to concentrate on those that we do. It is also the filtering out of specific details. From this, we create a representation (idea) of what we are trying to solve.

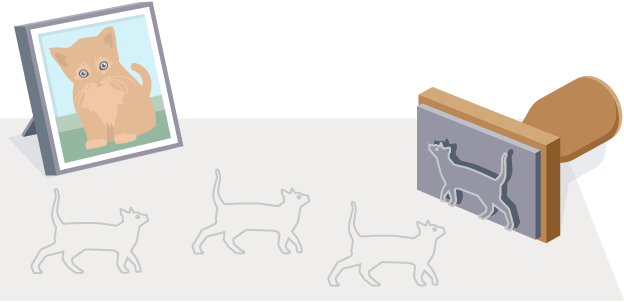
**What are specific details or characteristics?**

In **pattern recognition** we looked at the problem of having to draw a series of cats.

We noted that all cats have general characteristics, which are common to all cats, eg eyes, a tail, fur, a liking for fish and the ability to make meowing sounds. In addition, each cat has specific characteristics, such as black fur, a long tail, green eyes, a love of salmon, and a loud meow. These details are known as specifics.

In order to draw a basic cat, we do need to know that it has a tail, fur, and eyes. We don't need to know what sound a cat makes or that it likes fish. These characteristics are irrelevant and can be filtered out. We do need to know that a cat has a tail, fur, and eyes, but we don't need to know what size and color these are. These specifics can be filtered out.

From the general characteristics we have (tail, fur, eyes) we can build a basic idea of a cat, ie what a cat basically looks like. Once we know what a cat looks like we can describe how to draw a basic cat.



**Why is abstraction important?**

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 form our idea of the problem. This idea is known as a ‘model’.

If we don’t abstract we may end up with the wrong solution to the problem we are trying to solve. With our cat example, if we didn’t abstract we might think that all cats have long tails and short fur. Having abstracted, we know that although cats have tails and fur, not all tails are long and not all fur is short. In this case, abstraction has helped us to form a clearer model of a cat.

**How to abstract**

Abstraction is the gathering of the general characteristics we need and the filtering out of the details and characteristics that we do not need.

When baking a cake, there are some general characteristics between cakes. For example:

* a cake needs ingredients
* each ingredient needs a specified quantity
* a cake needs timings

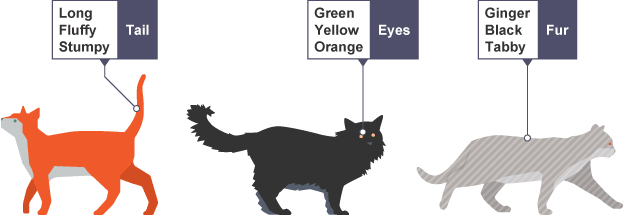
When abstracting, we remove specific details and keep the general relevant patterns.

| **General patterns** | **Specific details** |
| --- | --- |
| We need to know that a cake has ingredients | We don't need to know what those ingredients are |
| We need to know that each ingredient has a specified quantity | We don’t need to know what that quantity is |
| We need to know that each cake needs a specified time to bake | We don't need to know how long the time is |

**Creating a model**

A model is a general idea of the problem we are trying to solve.

For example, a model cat would be any cat. Not a specific cat with a long tail and short fur - the model represents all cats. From our model of cats, we can learn what any cat looks like, using the patterns all cats share.



Similarly, when baking a cake, a model cake wouldn’t be a specific cake, like a sponge cake or a fruit cake. Instead, the model would represent all cakes. From this model we can learn how to bake any cake, using the patterns that apply to all cakes.

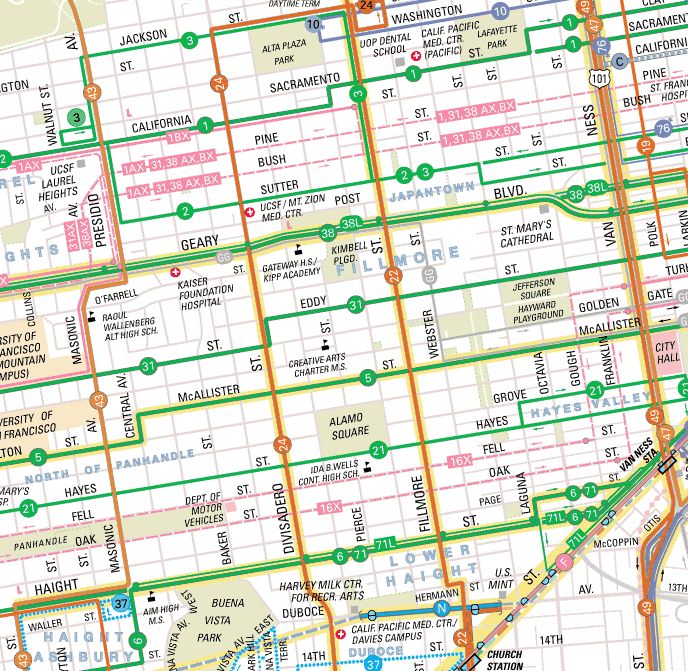
Once we have a model of our problem, we can then **design an algorithm** to solve it.

**Converting a map into a more abstract version of it (Unrealistic but easier to understand)**

**An example of abstraction.**

**Should transit maps be geographical or abstract?**

It goes without saying that transit maps should be geographically accurate. Many agencies follow San Francisco Muni in superimposing transit lines on a detailed map of the city:



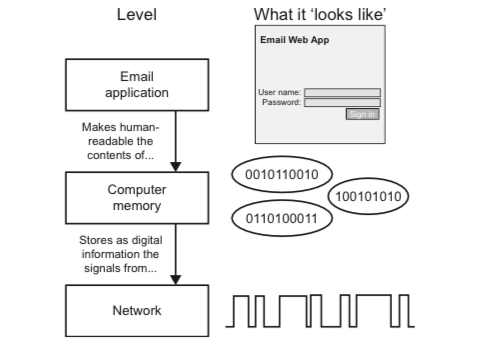
But researches suggest that we really need to see network structure, and that requires a degree of abstraction:

By putting alternate versions of the New York and Boston subway maps through the computer model, the researchers showed that abstract versions of the maps (as opposed to geographically accurate versions) were more likely to be easily understood in a single, passing glance.

**Another Example to Abstraction: E-mails**

If you ask a computer scientist what’s so great about abstractions, they may well give you a more technical example, say, an email. Most of the world knows what an email is: a message written on a computer and transmitted via a network to another user. However, this is an abstraction in the same way that ‘letter’ is an abstraction for a piece of paper with intelligible ink markings on it. There’s a lot more underlying detail to an email. If you imagine an email, you probably think of the text in your mail client.

But an email exists as digital information, organized as packets of ones and zeroes in your computer’s memory. What’s more, when in transit, it exists as a series of electrical impulses traveling through a cable, or as electromagnetic waves hurtling through the atmosphere. If you operated at this level of abstraction, you’d be saying to your friend, ‘I’ll send you an encoded series of ones and zeroes via electrical impulses over the Internet for you to decode into the original human-readable message.’ To which your confused friend might unwittingly reply, ‘Why not just send me an email instead?’



**Algorithm Design**

**Brief Explanation of Algorithm.**

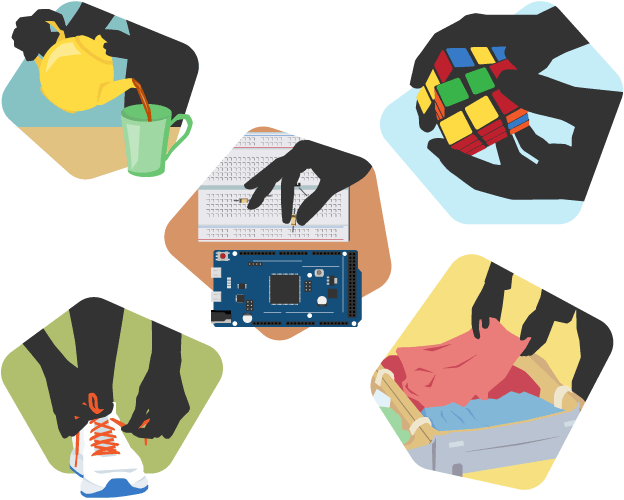
An algorithm is a sequence of clearly defined steps that describe a process to follow a finite set of unambiguous instructions with **clear start and endpoints.** Algorithms are a way of specifying a multi-step task, and are especially useful when we wish to explain to a third party (be it human or machine) 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.

In an algorithm, each instruction is identified and the order in which they should be carried out is planned. Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a flowchart or in pseudocode.

If we want to tell a computer to do something, we have to write a computer program that will tell the computer, step-by-step, exactly what we want it to do and how we want it to do it. This step-by-step program will need planning, and to do this we use an algorithm.

Computers are only as good as the algorithms they are given. If you give a computer a poor algorithm, you will get a poor result – hence the phrase: ‘Garbage in, garbage out.’

Algorithms are used for many different things including calculations, data processing and automation.



* Making a plan

It is important to plan out the solution to a problem to make sure that it will be correct. Using computational thinking and decomposition we can break down the problem into smaller parts and then we can plan out how they fit back together in a suitable order to solve the problem.

This order can be represented as an algorithm. An algorithm must be clear. It must have a starting point, a finishing point and a set of clear instructions in between.

**Pseudocode**

Representing an Algorithm: Pseudocode

* **What is Pseudocode?**

There are two main ways to represent an algorithm: **Pseudocode** and **flowchart**. Pseudocode is an informal high-level description of a computer program or algorithm. It is written in symbolic code which must be translated into a programming language before it can be executed. Using Pseudocode is similar to “writing in a programming language” and might look something like this:

OUTPUT 'What is your name?'

INPUT user inputs his/her name

STORE the user's input in the name variable

OUTPUT 'Hello' + name

OUTPUT 'How old are you?'

INPUT user inputs his/her age

STORE the user's input in the age variable

IF age >= 70 THEN

OUTPUT 'You are aged to perfection!'

ELSE

OUTPUT 'You are a spring chicken'

***INPUT*** asks a question. ***OUTPUT*** prints a message on the screen.

Pseudocode makes creating programs easier. Programs can be complex and long; preparation is the key. For years, flowcharts were used to map out programs before writing one line of code in a language. However, they were difficult to modify and with the advancement of programming languages, it was difficult to display all parts of a program with a flowchart. It is challenging to find a mistake without understanding the complete flow of a program. That is where pseudocode becomes more appealing.

To use pseudocode, all you do is write what you want your program to say in English. Pseudocode allows you to translate your statements into any language because there are no special commands and it is not standardized. Writing out programs before you code can enable you to better organize and see where you may have left out needed parts in your programs. All you have to do is write it out in your own words in short statements. Let's look at some examples.

Begin

INPUT hours

INPUT rate

pay = hours \* rate

OUTPUT pay

End

And this, slightly more complex example might calculate pay with overtime:

Begin

INPUT hours, rate

IF hours < 40

THEN

pay = hours \* rate

ELSE

pay = 40 \* rate + (hours - 40) \* rate \*1.5

OUTPUT pay

End

Most programs are developed using programming languages. These languages have a specific syntax that must be used so that the program will run properly. **Pseudocode is not a programming language**, it is a simple way of describing a set of instructions that does not have to use specific syntax.

Writing in a pseudocode is similar to writing in a programming language. Each step of the algorithm is written on a line of its own in sequence. Usually, instructions are written in uppercase, variables in lowercase and messages in sentence case.

* **Why use Pseudocode?**

A prototype is an early sample, model or release of a product created with the intention of concept testing and for learning purposes. They help us to learn without fully implementing our solutions. When developing user interfaces for our applications, we have several prototypes before the final interface. Some examples of these are wire-frames, graphical designs, and mock-ups. The same applies to writing technical code. Directly writing code for complex purposes might result in time wastage. The causes of this range from improper algorithms to ambiguous program flow. To prevent this, we can use Pseudocode.

Advantages of pseudocode:

* Pseudocode is understood by the programmers of all types.
* It enables the programmer to concentrate only on the algorithm part of the code development.
* It cannot be compiled into an executable program.

**How to Write Pseudocode - 1**

**Statements**

A statement is defined as an instruction that directs the computer to perform a specific action. In writing pseudocode, we will refer to singular instructions as statements. When writing pseudocode, we assume that the order of execution of the statements is from top to bottom. This changes when using control structures, functions and exception handling.

* **Mathematical operations** Mathematical operations are integral to solution development. They allow us to manipulate the values we have stored. Here are common mathematical symbols:

Assignment: ← or :=

Example: c ← 3, c := 2

Comparison: =, ≠, <, >, ≤, ≥

Arithmetic: +, −, ×, /, mod

Logical: and, or

* **Keywords** A keyword is a word that is reserved by a program because the word has a special meaning. Keywords can be commands or parameters. Every programming language has its own keywords (reserved words). Keywords cannot be used as variable names. In Pseudocode, they are used to indicate common input-output and processing operations. They are written fully in uppercase.

START, BEGIN: This is the start of your pseudocode.

INPUT: This is data retrieved from the user through the input device.

READ, GET: This is used when reading data from a data file.

PRINT, DISPLAY, SHOW, OUTPUT: This will show your output to a screen.

COMPUTE, CALCULATE: To calculate the result of the expression.

SET, INIT: To initialize values

INCREMENT, BUMP: To increase the value of a variable

DECREMENT: To reduce the value of a variable

END: This is the end of your pseudocode

**Conditionals**

During algorithm development, we need statements that evaluate expressions and execute instructions depending on whether the expression evaluated to True or False. Here are some common conditions used in Pseudocode:

* **IF — ELSE IF — ELSE**

This is a conditional that is used to provide statements to be executed if a certain condition is met. This also applies to multiple conditions and different variables. Here is an if statement with one condition

IF you are happy

THEN smile

ENDIF

Here is an if statement with an else section. Else allows for some statements to be executed if the “if” condition is not met.

IF you are happy

THEN smile

ELSE

frown

ENDIF

We can add additional conditions to execute different statements if met.

IF you are happy

THEN smile

ELSE IF you are sad

THEN frown

ELSE

keep face plain

ENDIF

* **CASE**

Case structures are used if we want to compare a single variable against several conditions.

INPUT color

CASE color of

red: PRINT "red"

green: PRINT "green"

blue: PRINT "blue"

OTHERS

PRINT "Please enter a value color"

ENDCASE

The OTHERS clause with its statement is optional. Conditions are normally numbers or characters.

**Iteration**

To iterate is to repeat a set of instructions in order to generate a sequence of outcomes. We iterate so that we can achieve a certain goal.

* **FOR structure**

The FOR loop takes a group of elements and runs the code within the loop for each element.

FOR every month in a year

Compute number of days

ENDFOR

* **WHILE structure**

Similar to the FOR loop, the while loop is a way to repeat a block of code as long as a predefined condition remains true. Unlike the FOR loop, the while loop evaluates based on how long the condition will remain true.

To avoid a scenario where our while loop runs infinitely, we add an operation to manipulate the value within each iteration. This can be through an increment, decrement, et cetera.

PRECONDITION: variable X is equal to 1

WHILE Population < Limit

Compute Population as Population + Births — Deaths

ENDWHILE

**Functions**

When solving advanced tasks it is necessary to break down the concepts in a block of statements in different locations. This is especially true when the statements in question serve a particular purpose. To reuse this code, we create functions. We can then call these functions every-time we need them to run.

Function clear monitor

Pass In: nothing

Direct the operating system to clear the monitor

Pass Out: nothing

Endfunction

To emulate a function call in pseudocode, we can use the Call keyword

call: clear monitör

**Program Wrapping**

After writing several functions in our pseudocode, we find the need to wrap everything into one container. This is to improve readability and make the execution flow easier to understand.

To do this, we wrap our code as a program. A program can be defined as a set of instructions that performs a specific task when executed.

PROGRAM makeacupoftea

END

**Exception Handling**

An exception is an event which occurs during program execution that disrupts the normal flow of the instructions. These are events that are non-desirable.

We need to observe such events and execute code-blocks in response to them. This is called exception handling.

BEGIN

statements

EXCEPTION

WHEN exception type

statements to handle exception

WHEN another exception type

statements to handle exception

END

**Conclusion**

There are no technical rules for Pseudocode. It is meant to be human-readable and still convey meaning and flow.

There are different guide and tutorials which lean more towards language-specific pseudocode, examples of such are Fortran style pseudocode, Pascal style pseudocode, C style pseudocode and Structured Basic style pseudocode.

**How to Write Pseudocode - 2**

Now that you know what pseudocode looks like, it’s time to learn key aspects of how to write it.

We’re going to break down the process of writing pseudocode step by step so that you know exactly how to write it and how to use it effectively.

**1. Understand the Uses**

It’s difficult to use pseudocode if you don’t actually understand its many uses.

For starters, pseudocode simply makes the task of creating a new computer program more simple and straightforward.

Writing out the code in English enables you to create a verbal outline to follow during the programming stages of the project.

Pseudocode gives you the tools needed to ensure that everything you need will be included during programming. It lets you catch mistakes before they become mistakes.

It’s also highly beneficial to use pseudocode for group projects. It breaks the program down in a simple manner so that all programmers are on the same page.

**2. Pseudocode is Subjective**

The trickiest thing about pseudocode is that it is subjective.

There is no standard way to write pseudocode. The goal is simply to properly outline everything in your mind.

That said, there are certain structures and standard procedures you should use if you’re working with others. Follow these rules to ensure that everyone else on the team is on the same page.

Perhaps the most important rule is to place clarity first. Make your pseudocode as clear and concise as possible, so there is no question as to what you mean.

**3. Algorithms and Basic Constructs**

Two of the most important things you must understand when it comes to writing pseudocode are algorithms and basic algorithm constructs.

* **Understand Algorithms –** An algorithm is the steps you must take to achieve a specific goal.
* **Know Algorithm Flow –** The most basic algorithm construct or flow is “sequence,” “selection,” and “iteration.” These lay out the proper way to write the code.
* **Combine the Pieces –** Take the information you want to relay and use algorithm flow to create a straightforward outline.

**4. Standard Procedure**

As mentioned above, there is no standard procedure for writing pseudocode. However, that doesn’t mean there aren’t certain rules you should follow.

Follow these basic rules to ensure that everyone you’re collaborating with understands your pseudocode.

* **One Statement Per Line –** Express each statement or action on its own line.
* **Capitalize Directions –** Capitalize directions to highlight their importance (for example, “READ”).
* **Focus on Meaning –** Write what the program will do. Don’t write how to program it.
* **Standard Programming Structures –** Follow the algorithm flow discussed above to create easy-to-follow structures.
* **Utilize Blocks –** Group similar actions together into blocks to separate the pseudocode into separate steps.

**5. Important Tips**

Once again, while pseudocode doesn’t have any hard and fast rules, there are certainly some things you should do to make understanding the pseudocode easier for all involved.

* **Keep It Simple –** Simplicity and clarity are key. Write down what the actions will be, not how to program them.
* **Explain Everything –** Don’t include information without explaining it. Add comments to explain your steps and reasoning if needed.
* **Practice Makes Perfect –** Just like learning a new programming language, learning how to write pseudocode takes time. Practice writing and reviewing it now.
* **Review the Pseudocode –** The biggest reason to write pseudocode is to catch any mistakes before programming. So, review the finished product thoroughly to nip errors in the bud.
* **Translate into Programming Language –** Implement the pseudocode by tracing it with your computer language. Compare the finished product to the pseudocode.

**Practice 1: Write a pseudocode that reads two numbers and multiplies them together and print out their product.**

Begin

INPUT number1 , number2

SET answer to num1 \* num2

OUTPUT answer

End

**Practice 2: Write a pseudocode that tells a user that the number they entered is not a 5 or a 6.**

Begin

INPUT number

IF (number = 5)

OUTPUT "your number is 5"

ELSE IF (number = 6)

OUTPUT "your number is 6"

ELSE

OUTPUT "your number is not 5 or 6"

End

**Practice 3: Write a pseudocode to print all multiples of 5 between 0 and 100 (including both 0 and 100).**

Begin

SET x to 0

WHILE (x <= 20)

OUTPUT x \* 5

x = x + 1

End

**Practice 4: Write a pseudocode that will count all the even numbers up to a user-defined stopping point.**

For example, say we want to see the first 5 even numbers starting from 0.

Well, we know that evens numbers are 0, 2, 4, etc.

The first 5 even numbers are 0, 2, 4, 6, 8.

The first 8 even numbers are 0, 2, 4, 6, 8 ,10 ,12, 14

Begin

INPUT stopping\_point

SET x to 0

SET even to 0

WHILE (x < stopping\_point)

OUTPUT even

x = x + 1

SET even to even + 2

End

**Practice 5: Write a pseudocode that will perform the following.**

Read in 5 separate numbers.

Calculate the average of the five numbers.

Find the smallest (minimum) and largest (maximum) of the five entered numbers.

Write out the results found from steps 2 and 3 with a message describing what they are

Begin

OUTPUT "please enter 5 seperate numbers"

INPUT n1, n2, n3, n4, n5

OUTPUT "The average is:"

SET avg to (n1 + n2 + n3 + n4 + n5) / 5

OUTPUT avg

IF (n1 < n2)

SET max to n2

ELSE

SET max to n1

IF (n3 > max)

SET max to n3

IF (n4 > max)

SET max to n4

IF (n5 > max)

SET max to n5

OUTPUT "The max is:"

OUTPUT max

IF (n1 > n2)

SET min to n2

ELSE

SET min to n1

IF (n3 < min)

SET min to n3

IF (n4 < min)

SET min to n4

IF (n5 < min)

SET min to n5

OUTPUT "The min is:"

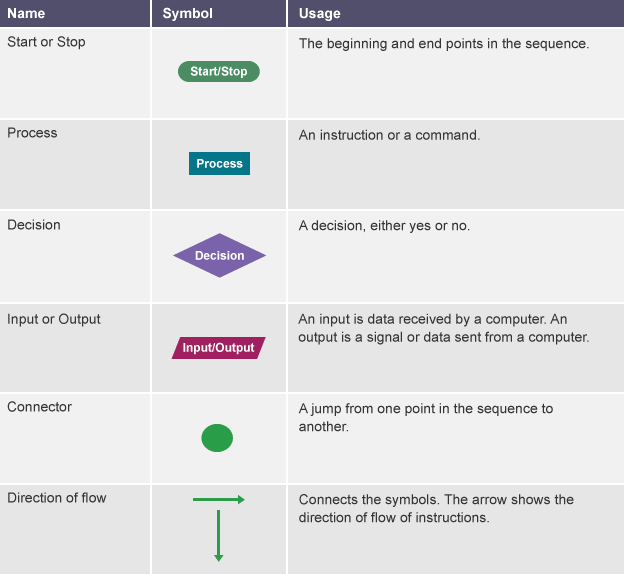
OUTPUT min

End

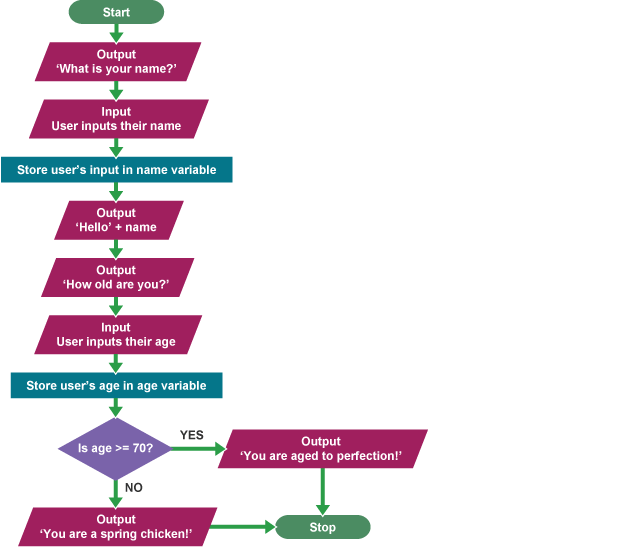
**Representing an Algorithm: Flowchart**

A flowchart is a diagram that represents a set of instructions. Flowcharts normally use standard symbols to represent different instructions. There are few real rules about the level of detail needed in a flowchart. Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening. Sometimes they are simplified so that a number of steps occur in just one step.

**Flowchart symbols**

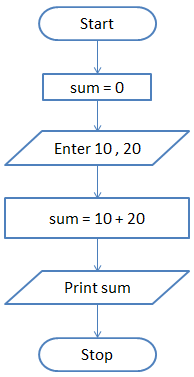


A **simple** program could be created to ask someone their name and age, and to make a comment based on these. This program represented as a flowchart would look like this:

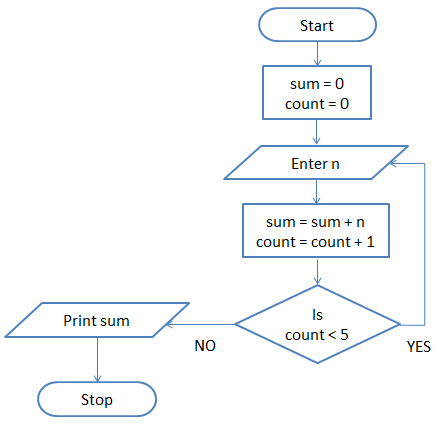


**Flowchart: Practice**

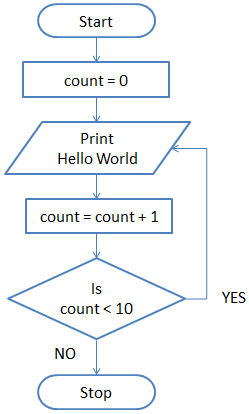
**Practice 1:** Add 10 and 20 then display the sum.

To solve this problem we will take a variable sum and set it to zero. Then we will take the two numbers 10 and 20 as input. Next, we will add both the numbers and save the result in the variable sum i.e., sum = 10 + 20. Finally, we will print the value stored in the variable sum.  


**Practice 2:** Take 5 numbers from the user and display the sum of the numbers.

In this question, we are asked to find the sum of 5 numbers. So, we will take two variables - sum and count and set both of them to zero. The sum variable will store the result while the count variable will keep track of how many numbers we have read. To solve this problem we will use the concept of a loop. In loop or iterative operation, we execute some steps repeatedly as long as the given condition is TRUE. In this case, we will keep reading the input until we have read 5 numbers. So, we first initialize sum and count to zero. Then we will take the input and store it in a variable n. Next, we will add the value stored in n to sum and save the answer in sum. i.e., sum = sum + n Then we will increment the count by 1 and check if the count is less than 5. If this condition is TRUE then we will take another input. If the condition is FALSE then we will print the value stored in the variable sum.  


**Practice 3:** Print Hello World 10 times.

This problem is also solved using the loop concept. We take a variable count and set it to zero. Then we print "Hello World" and increment count by 1. i.e., count = count + 1 Next, we check if the count is less than 10. If this is TRUE then we again print "Hello World" and increment the variable count. On the other hand, if the condition is FALSE then we will stop.  


Formun Üstü

Formun Altı

Formun Üstü

Formun Altı

Formun Üstü

Formun Altı