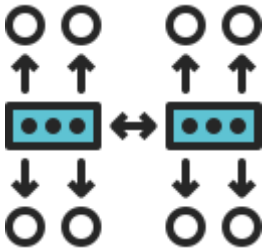
Topic 4 Theory of Computation

Algorithms



An algorithm is a series of defined steps which when followed complete a specific task. Algorithms do not contain infinite loops and must always terminate.

Pseudocode is a way of writing algorithms or other code which is not linked to a programming language. It allows programmers to communicate and share ideas without needing to all understand the same programming language.

Assignment is giving a value to a variable or constant. In pseudocode this is written as an arrow pointing to the variable or constant.

```
Count ← 77
```

Sequence is when several instructions are executed one after the other. In pseudocode instructions are written one per line, and lines are always executed in order.

Selection allows different sections of code to be executed depending on the result of a comparison. Pseudocode can use IF, ELSE IF, ELSE and END IF for selection.

Iteration, also known as loops, is the process of repeating a section of code more than once. Pseudocode can use FOR and WHILE to execute iteration. Code within the loop should be indented to make it easier to read.

Abstraction

Abstraction involves removing any details from a problem which are unnecessary and not relevant to finding a solution. It is the process of simplifying the problem down to its key details, making it easier to find a solution.

Representational abstraction focuses on removing unnecessary detail from the problem to simplify it.

Abstraction by generalisation / categorisation groups parts of the problem by common characteristics to arrive at a hierarchical relationship.

Information Hiding

Information hiding involves hiding any details or other information about an object which do not contribute to its essential characteristics. As an example, if writing an algorithm to determine how many people can fit on a bus then the people's names or hometown can be disregarded, whilst information about the height or weight of the people should be retained.

Procedural Abstraction

Procedural abstraction breaks down a complicated model into smaller parts, each of which is a reusable procedure. Abstraction is used to remove the actual values, allowing the code to be reused and a computational model formed.

Functional Abstraction

Abstracting this further can disregard the details of the procedure altogether, resulting in only a function call to provide the necessary data without needing access to or an understanding of how the function works.

Data Abstraction

Data abstraction removes details of how data is actually represented and stored, allowing new kinds of data structure to be created.

Problem Abstraction (Reduction)

Problem abstraction removes details from the problem one at a time until it is shown in a way which can be easily solved. When the problem is simplified, it often appears similar to another problem which has already been solved.

Decomposition

Decomposition divides a problem into a series of smaller subproblems, which can then be further divided. This takes a large problem and breaks it down into smaller parts which are easier to solve and manage.

Composition

Composition combines already written procedures to form a larger system when dealing with complex problems. The technique is used with abstract data types, which are formed from smaller simpler data types.

Automation

Automation used models, which are abstractions of real world problems, to solve problems. It used algorithms to form part of the code which together with these models can be executed against data structures to solve complex problems.

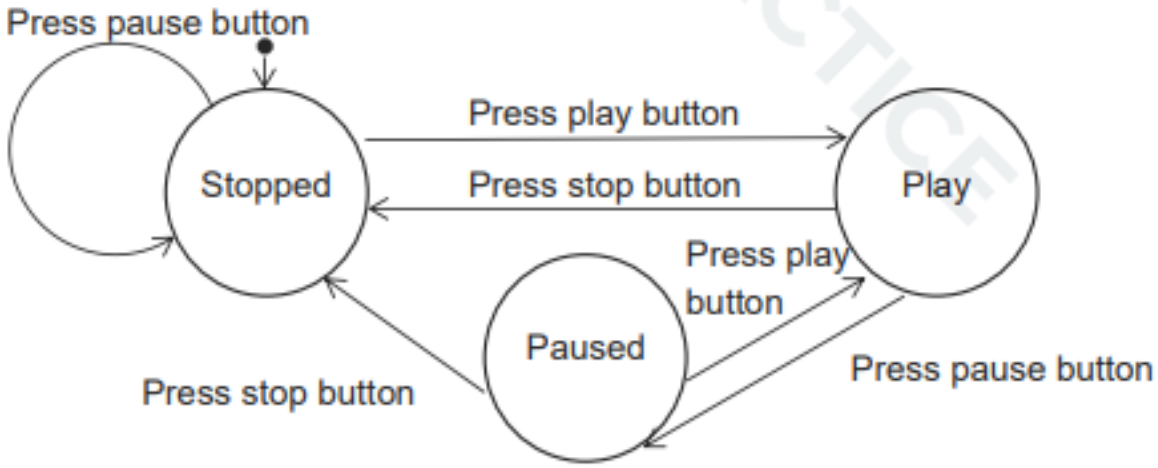
Finite State Machines (FSM)

A finite state machine is a computational model used to describe a machine which can only ever be in one of a number of finite states. The state of the machine can change based on its current state and the input data fed to it. If the input data is valid, the machine will process it and terminate in an accepting state. Transition rules define how these changes take place and how the machine should change state given certain criteria.

State Transition Diagrams

State transition diagrams are pictures which show how a finite state machine works. They are made of circles with arrows joining them together. They must always have a start state which is shown by an arrow leading into the diagram and accepting states are shown with a double circle.

The example below shows a media player, which can be in three states, stopped, play and paused. Transition functions are shown by the arrows.



State Transition Tables

Finite state machines can also be shown using state transition tables, which list the machine's current state, and how that state will change based on an input.

Current State	Input	Next State
Stopped	Play Button	Play
Play	Stop Button	Stopped
Play	Pause Button	Paused
Paused	Play Button	Play
Paused	Stop Button	Stopped

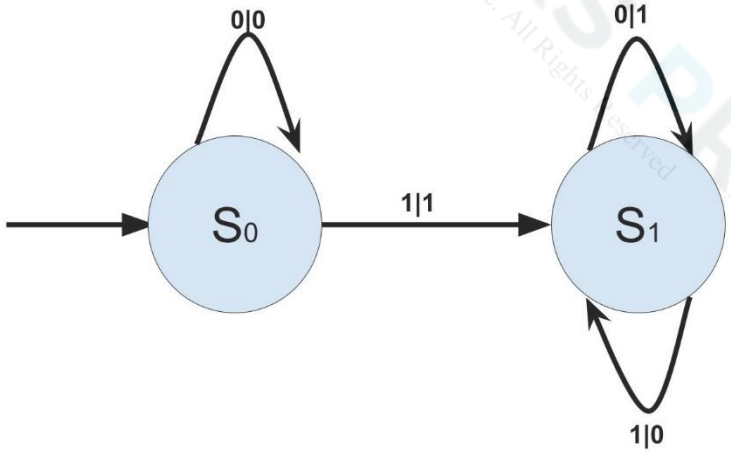
Finite State Machines With Outputs (Mealy Machines)

A Mealy Machine is a special type of finite state machine which can produce an output is called a Mealy Machine. The output is based on the input, and inputs can generate different outputs depending on the state it is applied to. These might be real systems or models of a logical system.

State transitions in these types of machines are labelled with both the input and output separated by a vertical bar.

The Mealy machine below converts binary numbers to two's complement representation with the number being read from the least significant bit to most significant bit. When the machine finds the first 1 it is left unchanged and all subsequent digits are flipped. As an example, the input 010100 would produce an output of 101100.

A state transition table can also be used to represent a Mealy Machine as shown below.



Current State	Input	Output	Next State
S0	0	0	S0
S0	1	1	S1
S1	0	1	S1
S1	1	0	S1

Sets

A set is an abstract data type containing unique unordered values, a set can also contain one or more other sets. It is also possible for a set to contain no elements, these are called empty sets and referred to using the symbol {} or Ø. Sets are written using the common set notation as shown below.

A set containing furniture in a dining room

```
D: {"Table", "Chair", "Sofa"}
```

A set containing furniture in a bedroom

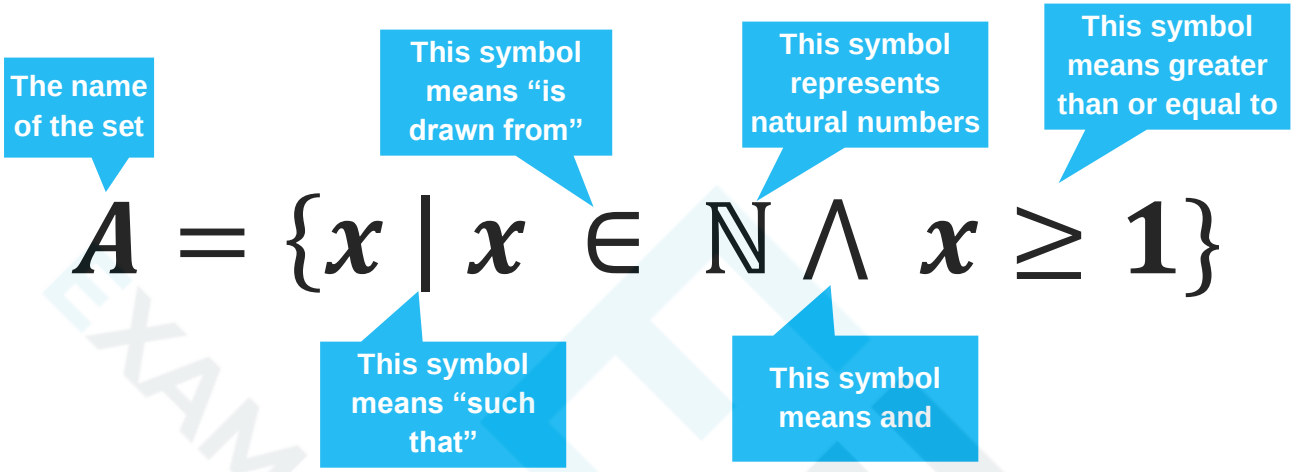
```
L: {"Bed", "Wardrobe", "Drawers"}
```

A set containing furniture in different rooms of a house

```
H: {("Bed", "Wardrobe", "Drawers"), ("Table", "Chair", "Sofa"})
```

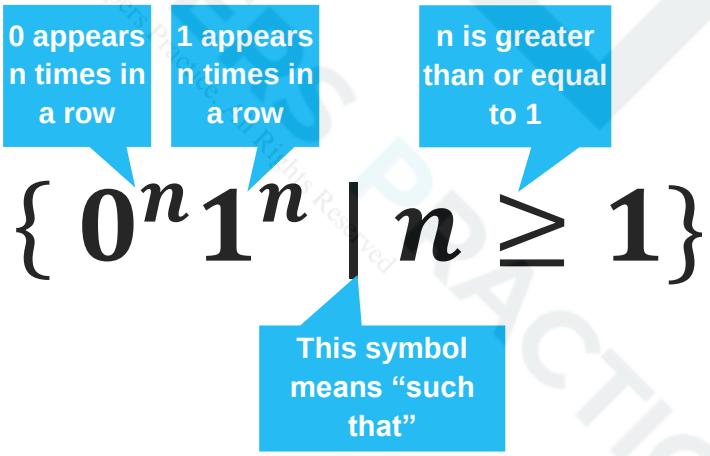
Set Comprehension

Set comprehension allows a set to be created by selecting items from a broader set rather than individually specifying all the items one at a time. The example below shows how a set might be constructed using this method to include all positive integers which are greater than 0. To put the example in plain English, the instruction is to create a set named A which includes numbers which are natural numbers and greater than or equal to 1.



Compact Set Representation

Compact set representation is a more space efficient way to describe a set by using shorthand methods to describe multiple instances of a number. The set below would contain all strings which have an equal number of 0s and 1s.



Common Set Types

There are a number of commonly used set types it is important to be aware of.

Finite Sets

Finite sets contain a set number of items, in other words we could count how many items were in the set. The cardinality of a finite set means the number of elements in a set. For example, a set containing the numbers 1 to 10 would be a finite set with a cardinality of 10. A set containing the names of 200 people would also be a finite set, but with a cardinality of 200

Infinite Sets

Infinite sets contain an infinite number of items, and can be thought of as the opposite of finite sets.

A countably infinite set contains elements which could be counted in such a way that we would eventually get to the end element, but in a very long time. An example would be the set of all integers, clearly this would be an incredibly large set but would have a finishing point.

A non-countable set contains elements which could not be counted. For example, the set of all real numbers would not be countable, since it contains every possible decimal point of every number.

Subsets

A subset is a set within another set and are written using the symbol $\subseteq$. For example, if every element of set A belongs to set B we could write $A \subseteq B$. If both A and B contain exactly the same elements, both are subsets of each other, and so we could write $A \subseteq B$ or $B \subseteq A$.

Propper Subsets

A proper subset contains only items from another set, but not all of them, and is written as $A \subset B$. This also means that A cannot be the same as B.

Set Membership

The symbol $\in$ is used to show that an item is within a set, for example if set R contained the number 4 we could write $4 \in R$.

The symbol $\notin$ is used to show that an item is not within a set. For example, if the set P did not contain the number 5 we could write $5 \notin P$.

Set Operators

Sets can be constructed from other sets using three different operations.

Union

A new set can be constructed by combining the items in two other sets, this process is called union and shown using the symbol $\cup$. If an item appears in both sets, it will appear only once in the new set as shown in the example below:

Set A: {1,4,5,7,2}
Set B: {9,10,11,2,1}
Set A ∪ Set B = {1,4,5,7,2,9,10,11}

Intersection

A new set can be constructed by selecting only the items which appear in both sets, this is called intersection and uses the symbol $\cap$. The example below shows this in action.

Set A: {1,4,5,7,2}
Set B: {9,10,11,2,1}
Set A ∩ Set B = {1,2}

Difference

A new set can be created using only the items which appear in one set but not another set, this is called difference and uses the symbol \ or -. The example below shows this in action

```
Set A: {1,4,5,7,2}
Set B: {9,10,11,2,1}
Set A \ Set B = {4,5,7}
```

Regular Expressions

Regular expressions use different metacharacters to describe sets. There are many metacharacters used in regular expressions, however, only the five below will be used in the exam:

Metacharacter	Description	Example
*	0 or more repetitions	Set AB* would be {A, AB, ABB, ABBB, etc.)
+	1 or more repetitions	Set PQ+ would be {PQ, PQQ, PQQQ, etc.
?	The previous character is optional	Set Book?s would be {Books,Boos}
|	Or	Set J|K would be {J,K}
()	Groups regular expressions	Set (NM)|(OP)Q would be {NMQ, OPQ}

Finite State Machines and Regular Expressions

Finite state machines can be used to show regular expressions, every regular expression has a corresponding finite state machine.

Context Free Languages

Context free languages are sets of strings and symbols which follow context free grammar rules. Production rules, which replace one character with another, describe which strings are and are not possible.

Backus-Naur Form

Backus-Naur form is a notation method for context free languages which uses statements where the right hand side of the statement defines the left hand side.

Non-Terminals

Text placed within angled brackets is called a non-terminal, although can also be called a meta component or syntactic variable. The example below shows how a non-terminal could be used to describe the makeup of a name:

<FullName> ::= <Title><Forename><Surname>

Terminals

Text without brackets is called a terminal and is always taken as the written value without any breaking down or interpretation. The example below shows how a child's age could be defined using the numbers 1-9. The straight line character is called a pipe symbol, and means or:

<Age> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

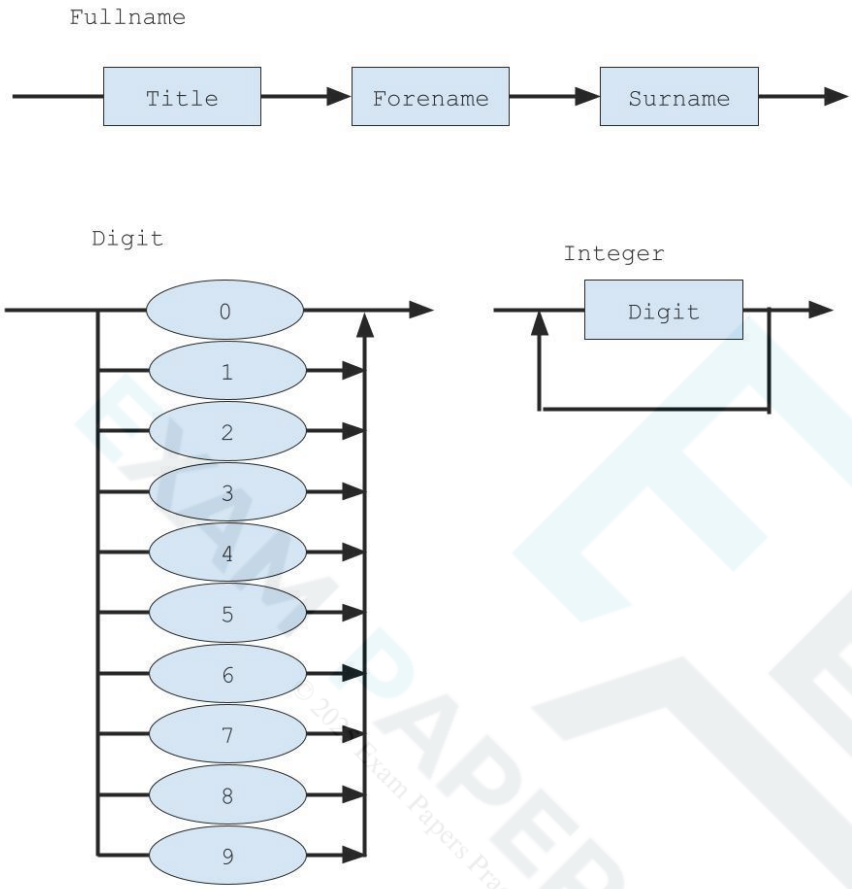
Recursions

Backus-Naur form uses recursion to make more complex definitions, defining a non-terminal in terms of itself to allow this. This allows the form to represent languages which cannot be represented using regular expressions, which do not allow recursion.

Syntax Diagrams

Syntax diagrams give a visual representation of a regular language, using rectangles to show non-terminals and ellipses to show terminals. Arrows join the shapes to show how strings can be formed.

The example syntax diagrams below show how this works. The first shows how Fullname can be formed from three components. Whilst the second shows first how Digit can be formed of any digit 0 to 9, and Integer can be formed by several digits.



Comparing Algorithms

There are often multiple different approaches to solving a problem, resulting in several different algorithms which accomplish the same task. It is important to be able to compare these algorithms to find out which is the most efficient and least complex. An algorithm's complexity can be measured either in terms of space or in terms of time. Ideally, an algorithm should run as quickly as possible and take up as little space as possible.

Big-O Notation

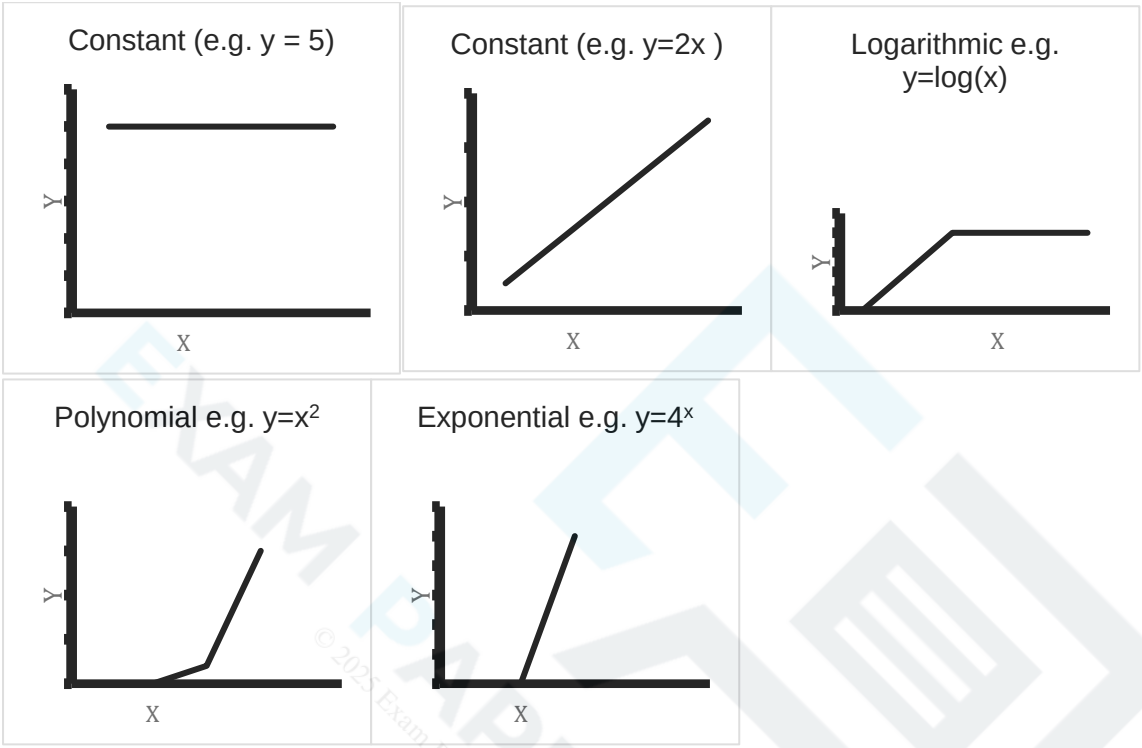
Big O notation is used to describe the complexity of an algorithm. This notation always assumes the worst case scenario and uses the letter n to describe the input. For example, an algorithm with a liner time complexity could be written as O(n).

There are a number of standard functions in Big O notation it is important to be familiar with, they are listed below in order from least to most complex.

Function	Big O	How to Spot
Constant	O(C)	The time is the same regardless of the input.
Logarithmic	$O(\log_2(n))$	The number of items is halved in each iteration.
Linear	O(n)	Each item must be processed once in a worst case scenario.
Linear Logarithmic	O(nlog(n))	
Polynomial	$O(n^2)$	Appears as a loop within a loop.
Polynomial	$O(n^3)$	Appears as a loop within a loop within a loop.

Exponential	$O(2^n)$	Intractable, meaning it cannot be solved within a practical amount of time.
Factorial	O(n!)	Intractable, meaning it cannot be solved within a practical amount of time.

Graphs for Big-O Notation



The graphs above show different results of the Big-O Notation which we can use to represent algorithm complexity. It is important to be able to recognise, different graphs increase at different rates. With the exception of the constant function, as the input increases the graph grows.

The factorial notation, written as x!, means all the positive integer values smaller than or equal to the number multiplied together, so 3! is 1x2x3, or 6. This is useful for working out permutations, for example, the numbers 1 to 5 could be ordered in 5!, or 96 ways.

Limits of Computation

Algorithms can be either traceable or intractable.

Tracible problems can be solved within a useful and practical amount of time and have a time solution which is polynomial or less.

Intractable problems are solvable in theory but limits on computational power mean that the algorithm would take so long to solve, sometimes millions of years, that it would not be worth waiting for the result. Sometimes a heuristic method can be used to produce an approximate solution, this will not provide an exact answer however.

It is also important to remember that not every problem can be solved using a problem. One example of this is the Halting problem, which states it is impossible to produce an algorithm to determine if a second algorithm will finish with a given input. This shows that there are some problems which simply cannot be solved by computers.

Turing Machines

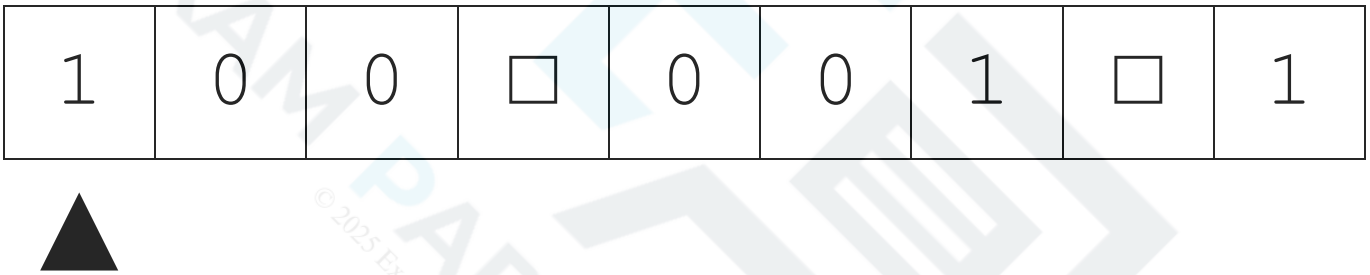
A Turing Machine is a computational model made of a finite state machine, a read/write head and a tape of infinite length.

The tape is divided into cells which may be blank or contain a symbol. These symbols are written to or removed from the tape by the read/write head. The set of available symbols is called the alphabet, and this must be finite and defined in advance.

We can look at a Turing Machine as a computer which runs a single program defined by a finite state machine with a start state, and several states from which there are no transitions (known as halting states). The machine will stop when it reaches a halting state, this can occur at any point in the machine's execution and indicates that all input data has been processed.

As shown below, a Turing Machine can be drawn as a series of cells each containing a single symbol or a square to signify the cell is empty. The black triangle indicates the position of the machine's read/write head.

Turing Machines can provide a more powerful computational model than finite state machines because they can utilise a greater range of languages and are infinitely long in one direction.



Transition functions lay out the rules a Turing Machine must follow and are written in this form:

$$\delta \text{ (current state, read) = (new state, write, move)}$$

The transition function below means that if the machine is in S_0 and reads an empty cell, it should write a 1, move to S_1, and finally move the read/write head to the right.

$$\delta \text{ (S0, □) = (S1, 1, R)}$$

Universal Turing Machines

A normal Turing Machine follows a single finite state machine meaning it is specific to a single computational problem it was designed to solve. A Universal Turing machine on the other hand is capable of representing any finite state machine. The universal Turing machine reads a description of the finite state machine which it will use from the same type as the input data then proceeds to process the input data as usual. This is an example of the stored programme concept.

Because of the way in which universal Turing machines read their instructions in sequence before executing operations on their input data, it can be said that they act as interpreters.

The importance of Turing Machines

In providing a formalised model of computing, Turing machines give a definition of what is and is not computable. This is incredibly important since a Turing machine can be used to prove whether a given problem can or cannot be solved by computers.