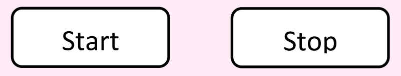
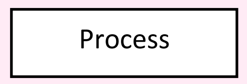
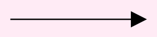
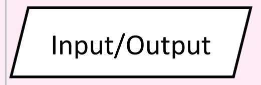
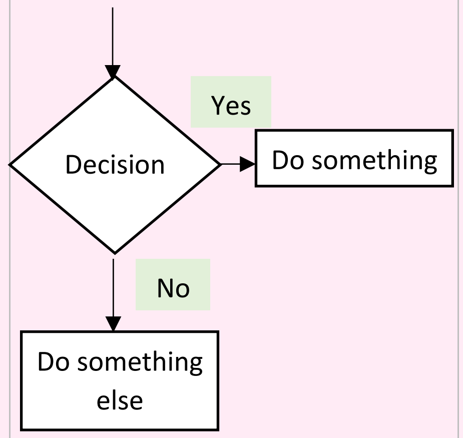
Flowchart Symbols

We can represent algorithms using flowcharts

Start and Stop 	**Process – An operation that the algorithm performs** 
Connector – Links all the other symbols together 	**Input and Output of data that is read in and written out** 

Decision is the same as a selection (if then … else)

	`IF answer is "yes" THEN`
	      `do something`
	`ELSE IF answer is "no"`
	      `do something else`
	`ENDIF`

Pseudocode

We can represent algorithms using pseudocode

	Example	Python equivalent
Variable assignment	a ← 10	`a = 10`
Constant assignment	constant PI ← 3.142	`PI = 3.142`
Input	a ← USERINPUT	`a = input()`
Output	OUTPUT "Bye"	`print("Bye")`
Arithmetic Operators Add Multiply Divide Subtract Integer division Modulus (remainder)	 + * / - a ← 7 DIV 2 a ← 7 MOD 2	 + * / - a= 7 // 2 a = 7 % 2
Relational Operators Less than Greater than Equal to Not equal to Less than or equal to Greater than or equal to	 < > = ≠ or <> ≤ ≥	 < > == != <= >=
Boolean Operators AND OR NOT	 AND OR NOT	 AND OR NOT

Selection
 	
if ..
 	
if .. else …
 	
if … else if … else
	IF i > 2 THEN
	  j ← 10
	ENDIF
	
	IF i > 2 THEN
	  j ← 10
	ELSE
	  j ← 3
	ENDIF
	
	IF i ==2 THEN
	  j ← 10
	ELSE IF i==3 THEN
	  j ← 3
	ELSE
	  j ← 1
	ENDIF
		if i > 2:
		  j=10
		
		if i > 2:
		    j=10
		else:
		    j=3
		
		if i ==2:
		  j=10
		elif i==3:
		  j=3
		else:
		  j=1

Iteration
 	
While loops
 	
For loops
 	
Repeat loops
	
	a ← 1
	WHILE a < 4
	    OUTPUT a
	    a ← a + 1
	ENDWHILE
	
	FOR a ← 0 TO 3
	    OUTPUT a
	ENDFOR
	a ← 1
	
	REPEAT
	    OUTPUT a
	    a ← a + 1
	UNTIL a←4
		
		while a<4:
		  print(a)
		  a=a+1
		
		for a in range(3):
		  print(a)

Arrays

	Example	Python equivalent
Set up array	a ← [1,2,3,4,5]	a=[1,2,3,4,5]
Access element	a[0]	a[0]
Update element	a[0] ← 4	a[0] = 4
Set up 2D array	a ← [[1,2],[3,4]]	a = [[1,2],[3,4]]
Access 2D element	a[0][1]	a[0][1]
Update 2D element	a[0][1] ← 4	a[0][1] = 4

Subroutines
 	
procedure
 	
Function (with paramerters and return)
	
	SUB hello()
	  OUTPUT "hello"
	ENDSUB
	
	SUB add(n)
	  a ← 0
	  FOR a ← 0 TO n
	    a ← a + n
	  ENDFOR
	  RETURN a
	ENDSUB
		
		def hello():
		  print("hello")
		
		def add(n):
		  a=0
		  for a in range(n+1):
		    a=a+n
		  return a

Built-in functions
 	
Length of array
 	
Random integer
	
	LEN(a)
	
	RANDOM_INT(0, 9)
		
		len(a)
		
		import random
		random.randint(0,9)

Abstraction

Representational abstraction
Abstraction allows us to remove unnecessary detail from a problem leaving only the essential features thereby making it easier to solve. Maps are examples of representational abstraction.

Abstract generalisation
With abstract generalisation we identify common (general) characteristics thereby enabling us to group similar constructs together into a hierarchy

Abstract generalisation is also the ability to see patterns so that we can recognise problems or parts of problems that we may have solved before. Lots of programming is concerned with reusing pieces of code that were originally developed for other solutions. Even within a piece of code we are writing we may notice that quite a lot of our code is repeated. It is our ability to notice those repetitions that help us write more succinct and generalisable code using functions perhaps.

Procedural abstraction
Abstract away the actual values used in a computational method. In that sense algebra and formulas are abstractions. The following expressions have the same form:

```
(1+2) x 3
(7+9) x 2
(5.5 + 12.3) x 18.1
```

We can abstract them away algebraically as:

```
(a+b) x c
```

Functional abstraction
- A functional abstraction maps an input to an output. The function returns a value given a certain input.
- Functional abstraction is an extension of procedural abstraction. A procedural abstraction might form part of a functional abstraction.

```
def calc(a,b,c):
    return (a+b)*c
print(calc(1,2,3))
```

- In programming we abstract details using functions. We do not need to know how functions work to use them. The functions themselves can be a black box to us. The details of how the function works have been abstracted away.

Data abstraction
The details of how the data are represented are hidden. We do not need to worry how ASCII characters, real numbers and integers are represented. Real numbers can be represented using exponent and mantissa in binary, but we do not need to concern ourselves about this when we a writing programs. We can have more complex abstract data types. These include queues, stacks graphs, trees, hash tables, dictionaries and vectors.

Problem abstraction
Remove details of a problem until you are left with a problem that you already know how to solve. This allows us to use solutions that have been applied to analogous problems. For instance Euler solved the Konigsberg bridge problem (is it possible to cross all bridges only once) by reducing it down to a graph problem, that he already knew how to solve.

Information hiding
In OOP, this is where data that do not contribute to the essential characteristics of an object are hidden. These attributes and methods are private and not accessible from outside an object. Essential characteristics of the object can be accessed via an interface. Also we do not need to concern ourselves with local variables in functions.

Decomposition
Decomposition is the breaking down of a complex problem into smaller more manageable problems that are easier to solve. Each component of the program completes a specific task. This allows algorithms to be more modular and therefore more intuitive.

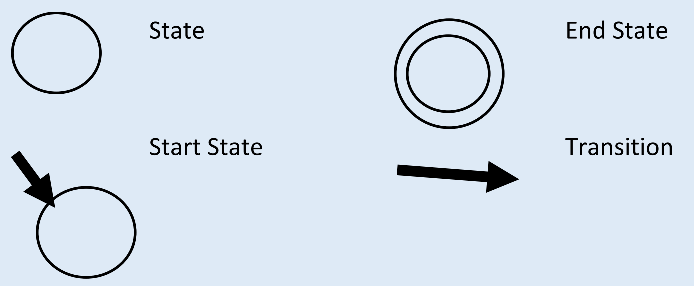
Composition
Composition is combining the procedures together to form compound procedures in order to solve a greater part of the problem that each of the procedures can solve separately. Specifying the interface between the components is important otherwise they would not fit together.

Automation
- Putting models into action using algorithms
- Putting the abstractions into algorithms and putting the algorithms into code.
- Developing computer models that concentrate on the essence of a problem. The models are a simplified representation of reality where assumptions are made.
- For instance, weather forecast models use mathematical models and physics to model the atmosphere as a fluid, which is a good way to run simulations and predict the weather up to a few days ahead.
- It is not possible to model the billions of variables, so simplifications are made to help solve the problem. As computers get more powerful and algorithms improve and we have more data we get better at predicting the weather.

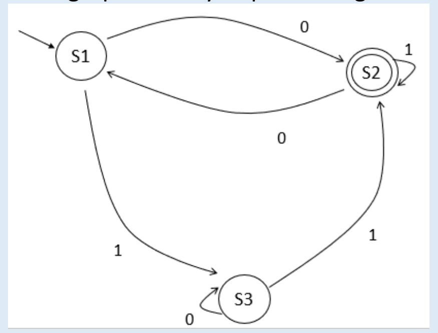
Finite State Machines (FSMs)

FSMs are a model of computation that allow us to understand how computers work. FSMs consist of a set number of states that allows the transition between states and are determined by a fixed set of inputs and have a set of outputs.

Notation for FSM



Finite state diagrams are a graphical way of presenting finite state machines.



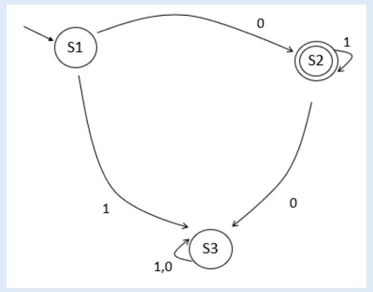
- S1, S2 and S3 are the states
- Each transition edge has an input value
- S1 is the Start State
- S2 is the Accept State)
- For the input sequence to be valid the sequence must end on the accept state (S2)

Example sequences
- 0 0 0 1 1 - Valid
- 1 0 0 1 - Valid
- 1 0 1 0 - Invalid
- 1 0 1 - Valid

State transition tables are another way of representing FSM.

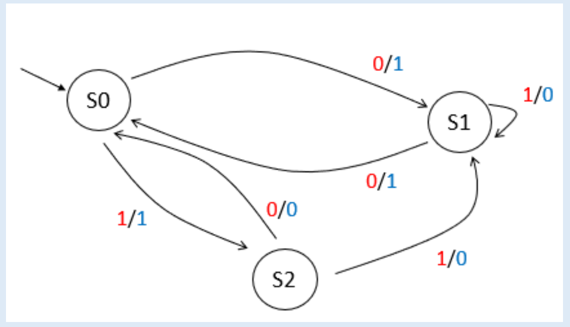
Start state	Input	New State
S1	0	S2
S1	1	S3
S2	0	S1
S2	1	S2
S3	0	S3
S3	1	S2

Trapping invalid input – In the following example S3 captures invalid input there is no what to transition to another state once S3 has been achieved.



Mealy Machines
- The FSM we have looked at so far have a valid and invalid state. The valid state is the accept state
- Mealy machines are a type of FSM that have outputs on each transition and have no end state

Example Mealy Machine



The red number is the input and the blue number is the output.

Input	Output
0 1 1 1 0 1 1 0 1	1 0 0 0 1 1 0 1 1
1 1 1 0 0 1 0 0 0	1 0 0 1 1 1 1 1 1
1 0 1 0 0 1 1 1 0	1 0 0 1 1 0 0 0 1

Corresponding state transition diagram

Start state	Input	New State	Output
S0	0	S1	1
S0	1	S2	1
S1	0	S0	1
S1	1	S1	0
S2	1	S1	0
S2	0	S0	0

Maths for Regular Expressions

A **set** is an unordered collection of values where each value occurs only once. Values can be numbers, symbols of letters. The contents of a set are represented using curly brackets. For instance, the set

$$A = \{1,\ 2,\ 3,\ 4,\ 5\}$$

defines all the integers between 1 and 5 inclusive, where the name of the set is A.

Notation of special sets

- N is the infinites set of natural numbers from 0 to infinity
- $N = \{0,1,2,3,4,\ \ldots.\}$ (infinite set has ellipses)
- $x \in N$ means x is a member of the set N
- R is the set of real numbers
- Empty sets are sets that contain no elements and are represented using {} or Ø

Set comprehension is a short hand for writing out sets. For instance they take the form of:

$$A = \{n \mid n \in N \wedge n < 7\}$$

This means the set is a set of natural numbers and the values are less than 7. The ^ character means Boolean AND and the | character means such that.
Therefore:

$$A = \{0,1,2,3,4,5,6\}$$

Examples of set comprehension

$A = \{\, x^2 \mid x \in N \wedge x < 4\}$	$A = \{0,1,4,9\}$
$A = \{2x \mid x \in N \wedge x < 4\}$	$A = \{0,2,4,6\}$
$A = \{\, 3x \mid x \in N \wedge x > 4 \wedge x < 10\}$	$A = \{15,18,21,24,27\}$

Compact representation of Sets

$$A = \{\ 1^n\, 0^n \mid n \geq 1\}$$

would produce the set:

$$A = \{10,1100,111000,11110000, \ldots..\}$$

A **finite** set can be counted up by natural numbers. It has a certain number of elements.

An **infinite** set has an infinite number of elements

The **cardinality** of a finite set is the number of members in a set

A **countable** set can be counted off by a finite subset of the natural numbers. A countable set has the same number of elements as a subset of the natural numbers

Index	0	1	2	3	4
Value	2	4	5	-2	-7

A **countably infinite** set can be counted off by the natural numbers.

Real numbers are not countable and you do not know which is the next value because they can be infinitesimal.

The **Cartesian product** of two sets A and B (A x B) is the set of all combinations of pairs of elements in A and B.

e.g. $A = \{2,4,6\}$ and $B = \{3,5,7\}$ C=A X B,

$$C = \{(2,3), (2,5), (2,7), (4,3), (4,5), (4,7), (6,3), (6,5), (6,7)\}$$

Membership

A **Proper subset** A contains everything in the set B, but there is at least one element in set B is not contained in subset A. A is a proper subset of B (or B is a superset of A)

$$A \subset B, B \supset A$$

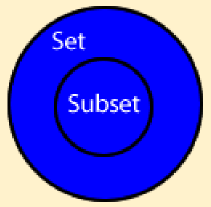
$$\text{e.g } A = \{0, 1, 2\}, B = \{0,\ 1,\ 2,\ 3\}$$

That is subset A will always have fewer elements than set B even if it only has one fewer element. In other words, All members of a subset will also be in a set. If there are n elements in a subset, a **proper** subset consists of a most n-1 elements

The difference between a subset and a proper subset is that subset A contains everything in the set B, and that all element in set B can be contained in subset A. A is a subset of B

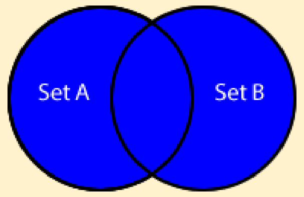
$$A \subseteq B, B \supseteq A$$

$$\text{e.g } A = \{0, 1, 2, 3\}, B = \{0,\ 1,\ 2,\ 3\}$$



Union

$A \cup B$ – A union B (Add together two sets)

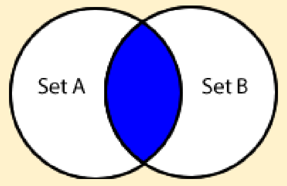


The Union of two sets consists of all elements from both sets. Where values are duplicated a value appears only once in the new set.

$$A = \{1,2,3,4,5,6\}$$
$$B = \{5,6,7,8,9,10\}$$
$$C = A \cup B$$
$$C = \{1,2,3,4,5,6,7,8,9,10\}$$

Intersection

$A \cap B$ – A intersects B – contains the members that both sets have in common

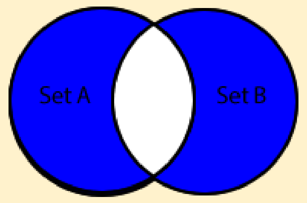


An intersection between 2 sets consists of elements that occur in both sets.

If A={1,2,3,4,5,6} and B={5,6,7,8,9,10}
The intersection between A and B is
{5,6}

Difference

A/B - Difference of set A and B



The difference between the 2 sets consists of elements that occur in one or other of the sets.

If A={1,2,3,4,5,6} and B={5,6,7,8,9,10}
The difference between A and B will be
{1,2,3,4,7,8,9,10}

Regular Expressions

A regular expression is a shorthand way of representing a set. The following characters are applied to the preceding value:

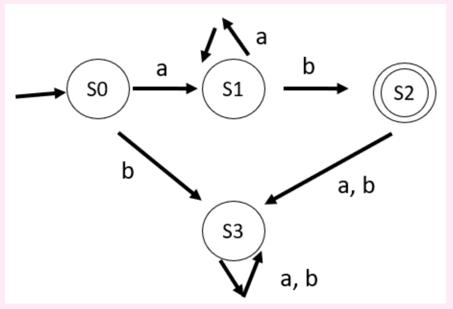
*	Zero or more repetitions
+	One or more repetitions
?	0 or 1
|	alternative
()	Group expression

Examples of regular expressions (The ellipses refer to infinite series)

ab	ab	*ab* only
a*b	b,ab,aab,aaab, ..	Any number of *a* followed by a single *b*
(ab)*	ab, abab, ababab, ..	Zero or more repetitions of *ab*
ab+	ab,abb,abbb,..	A single *a* followed by one or more *b*
a*b*	a,b,ab,aab,abb,..	Zero or more *a* followed by zero or more *b*
a?b	ab, b	Zero or one *a* followed by one *b*
a|b	a,b	*a* or *b*

Finite state machines can be used represent regular expressions. A regular language can be represented by a regular expression or FSM. A FSM recognises whether strings are valid for a language.

e.g. The finite state machine for *a+b* is given as:



Backus Naur Form (BNF)

A language is regular if it can be represented by a regular expression and an FSM.

However, some languages cannot be expressed using regular expressions.

A context-free language like Backus-Naur form (BNF) is necessary whenever an infinite number of elements need to be counted.

All regular languages can be represented by context-free languages.

Production Rules
BNF is expressed using production rules. For instance, a bit is defined with the following production rule.

```
<bit> ::= 0 | 1
```

This means that a bit can take on the value 0 or 1.

- < > is a non-terminal symbol. In this example `<bit>` is a non-terminal symbol and 0 and 1 are terminal symbols.
- ::= states that the right-hand side is defined by the left hand side
- | means a choice (OR) between two symbols

Non-Terminal Symbols
If there is a non-terminal symbol on the right hand side there should be another production rule with the non-terminal symbol on the left.

This is demonstrated in the following example of three production rules where the non-terminal symbols <month> and <year> appear on both the left and right-hand side.

```
<date>  ::= <month>/<year>
<year>  ::= 2018 | 2019 | 2020
<month> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 11 | 12
```

Valid expressions include 2/2020 or 12/2019

Recursion
Recursion allows us to have one or more of a symbol. Consider the following example:

```
<integer>::=<digit>|<digit><digit>|<digit><digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 |7 | 8 | 9
```

These pair of production rules only allow us to express a maximum value for an integer 999. Of course integers can have an infinite number of digits. We represent this using recursive BNF production rules.

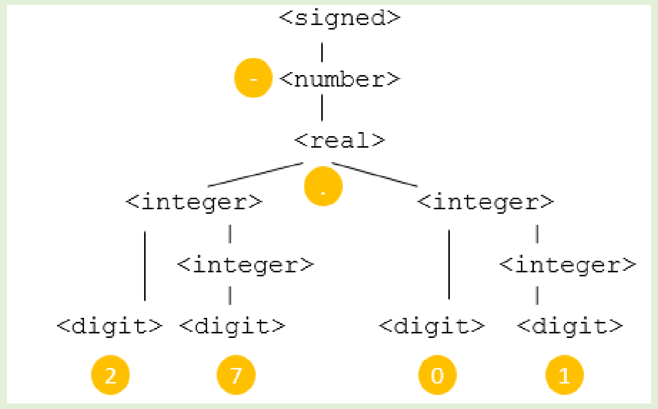
```
<integer> ::= <digit> | <digit><integer>
```

As we know with recursive functions we need a base case and general case. The base case is `<digit>` on its own and the general case is `<digit><integer>`

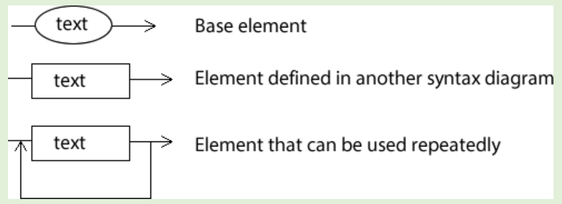
Parse Tree
We are going to extend our rules so that we can define real numbers and negative numbers.

```
<signed> ::= <number> | +<number> |-<number>
<number> ::= <real> | <integer>
<real> ::= <integer> . <integer>
<integer> ::= <digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 |7 | 8 | 9
```

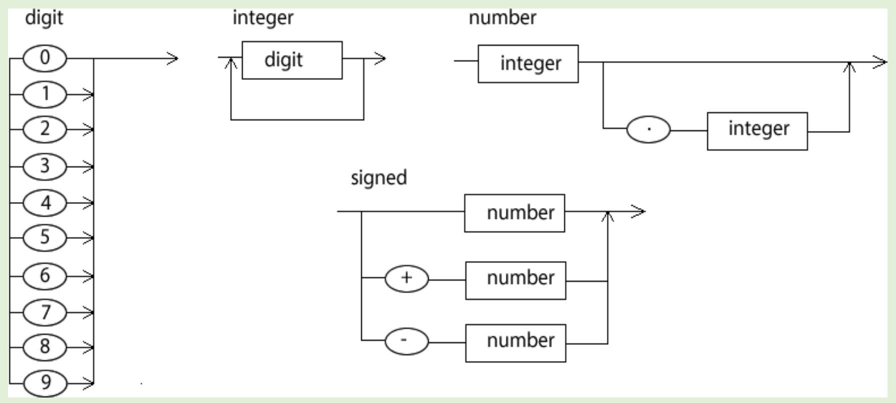
Let us check that -27.01 is a valid expression for these production rules by using a parse tree.



Syntax diagrams
Syntax diagrams are another way of expressing the syntax of a language. The symbols for the syntax diagram are:


text
Base element
text
Element defined in another syntax diagram
text
Element that can be used repeatedly

The syntax diagram for the BNF production rules that we have look at is given as:



Classification of Algorithms

Comparing Algorithms

- The time efficiency of algorithms refers how long an algorithm takes to run as a function of the size of the input.
- More than one algorithm can be used to solve the same proble.
- For instance to calculate the sum of a sequence of numbers we can use the following algorithm:

$$sum = (n + 1) * n / 2$$

where n is the number we wish to sum the values up to. Using this calculation the time remains constant regardless the value of n. In other words, regardless of how many numbers we wish to add up the time taken will always be same.

We could use an alternative algorithm to calculate the sum of a sequence of numbers

```
sum ← 0
FOR i ← 1 to n
      sum ← sum + i
ENDFOR
OUTPUT sum
```

Using this algorithm the number of operations increases in linear time with the size of the input. Therefore, the time taken for the algorithm to run will grow in linear time as in size of the input increases. Clearly this is more inefficient than the first algorithm even though it solves the same problem.

Another area where algorithms differ in their efficiency is in regard to the memory requirements of algorithms. For instance programs that read in huge data files into memory can end up taking up a large space in memory.

When developing algorithms it is important to consider the hardware constraints of the system you are developing for (eg mobile phone which has limited processing and space capability). If you have large memory then your algorithm can afford to be less space efficient. Likewise if you have access to tremendous processing power algorithm (eg supercomputer) your may not need to be time efficient although it is still desirable to make algorithms as efficient as possible.

Maths for Big-O Notation

A function allows us to map a set of input values to a set output values

$$y = f(x)$$

where x is a value from the domain and y a value from the codomain

domain -> codomain

A **linear function** takes the form $y = mx + c$, where m is the gradient and c the intercept on the y axis.

A **polynomial function** takes the form $y = ax^2 + bx + c$

An exponential function takes the form $y = a^x$

A **logarithmic function** takes the form $y = a \log_n x$

Permutations illustrate how the number of operations grows factorially when we add additional dimensions to some problems.

How many different combinations can sequence of digits have?

No. of digits	No of combinations
2	2
3	6
4	24
5	120

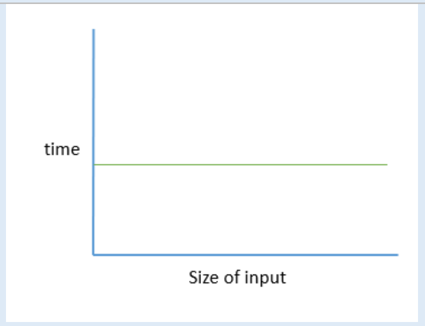
Big O notation gives us an idea of how long a program will run if we increase the size of the input. We need to consider how many operations will need to be carried out for a given size of input. This gives us the time complexity of the algorithm.

Constant Time O(1)

The time remains constant even when the number of input increases. E.g. calculating the sum of a sequence of numbers.

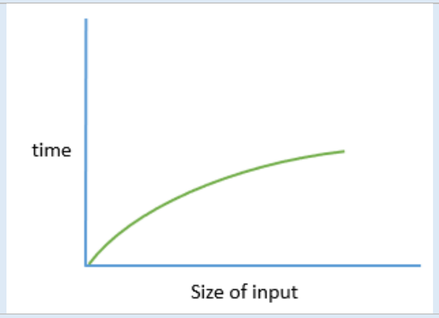
$$sum = (n + 1) * n / 2$$

Regardless of how many numbers we wish to add up the time taken will always be same.



Logarithmic Time O(log n)

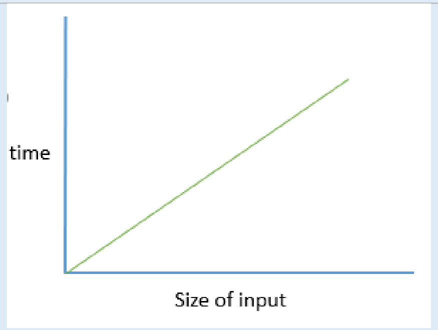
The time taken for the algorithm to run will grow slowly as in size of the input increases.



Linear Time O(n)

The time taken for the algorithm to run will grow in linear time as in size of the input increases.
Eg inefficient algorithm to calculate the sum of a sequence of numbers

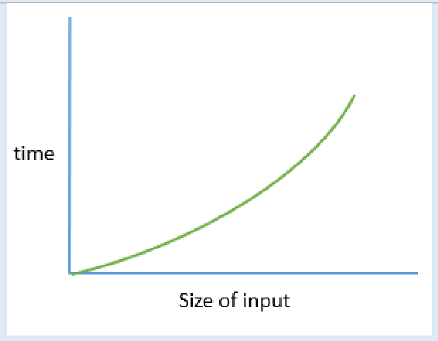
```
sum = 0
for i=0 to n
      sum = sum + i
output(sum)
```



Polynomial Time O(n^2)

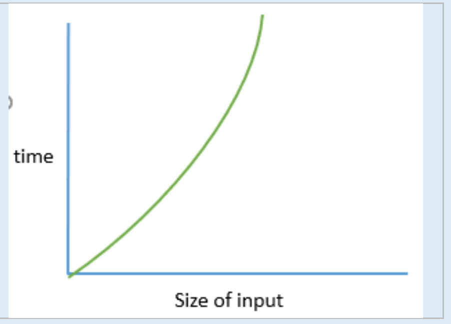
The time taken for the algorithm to run will grow proportionally to the square of the size of the data set. Normally when you have nested for loops this will have a polynomial time complexity.

```
for i=0 to n
 for j=0 to n
  Do something
```



Exponential Time O(2^n)

The time taken for the algorithm will grow as the power of the number of inputs, so the time taken for the algorithm to run will grow very quickly as more input data are added.



The time taken for an algorithm to run will depend on the hardware (eg processor clock speed, RAM size), even though the number of operations will be constant for a fixed input.

Tractable problems are problems that have a polynomial or less time solution eg O(1), O(n), O(log n), O(n^2)

Intractable problem are problems that can be theoretically solved but take longer than polynomial time e.g. O(n!), O(2^n)

Heuristic algorithms are used to provide approximate but not exact solutions to intractable problems

The travelling Salesman Problem
The idea is to find the shortest route to visit all cities. This is a permutation of the number of cities so has a factorial time complexity so quickly becomes an intractable problem with an unfeasibly huge number of permutations.

To solve this we use an heuristic algorithm. This provide an acceptable solution to the problem but it may not be the optimal or best solution. So for the travelling salesman problem we may find a short route but not necessarily the shortest route. Heuristic algorithms for the travelling salesman problem include the following:

- Greedy algorithm – take the shortest route to the next city
- Visit the cities in a circle
- Brute force algorithm – Apply to small but different subsets of cities and combine together. Apply the brute force algorithm to fewer manageable problems rather than a single intractable problem

Time Complexity of common algorithms

Linear Search	O(n)
Binary Search	O(log n)
Binary Tree Search	O(log n)
Bubble Sort	O(n^2)
Merge sort	O(n log n)
Travelling Salesman Problem	O(n!)
Brute force password cracker where n is the length of the password	O(A^n)

Unsolvable problems Some problems cannot be solved by a computer. The **Halting problem** is one such problem and show that some problems cannot be solved algorithmically.

The **halting problem** states that there is no computer program that exists that can determine whether another computer program will halt or will continue to run forever given some specified input.

The halting problem show that some problems cannot be solved by a computer

Turing Machines

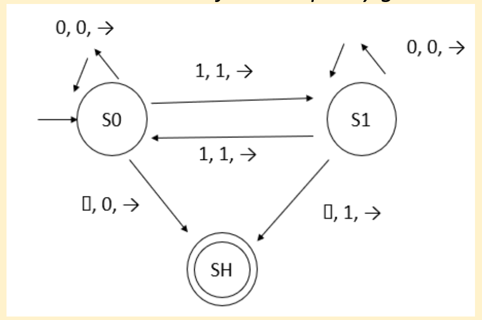
Purpose of Turing Machines

- Turing machines are a model of computation that help us understand how algorithms can be solved computationally.
- If a problem is computable then it can be solved by a Turing machine (Church-Turing thesis).
- Turing machines can be used determine whether an algorithm is computable.

How a Turing Machine Works

- A Turing machine is a finite state machine with a tape of infinite length that is divided into squares. This is the memory of the machine.
- Has a finite set of symbols, commonly 0 , 1 and □ which indicates no value. Each square on the tape takes on one of the values of the symbols.
- Has a head which can read and write to the tape and move along the tape in either direction.
- Has a finite set of states. It can have a start state and must have a halting state.
- Behave as interpreters because the deal with one instruction at a time.
- Turing machines can be expressed using:
    - Finite State Machines / diagrams
    - State transition tables
    - State transition functions

Finite state machine for even parity generator



First value is the symbol read, Second value is the symbol to write and third value is the direction in which to move the head

State transition table for even parity generator

State	Read	Write	Move	Next state
S0	0	0	→	S0
S0	1	1	→	S1
S0	□	0	→	SH
S1	0	0	→	S1
S1	1	1	→	S0
S1	□	1	→	SH

State transition function for even parity generator

∂(current state, input symbol) = (next state , output symbol ,direction)

∂(S0,0) = (S0,0,→)
∂(S0,1) = (S1,1,→)
∂(S0,□) = (SH,0,→)
∂(S1,0) = (S1,0,→)
∂(S1,1) = (S0,1,→)
∂(S1,□) = (SH,1,→)

Worked example:

Tape used for even parity generator	□	1	0	1	1	1	0	□	□	□

	The green arrow denotes the position of the read/ write head									
Step 1	□	1	0	1	1	1	0	□	□	□
		↑ S0								
Step 2	□	1	0	1	1	1	0	□	□	□
			↑ *S1*							
Step 3	□	1	0	1	1	1	0	□	□	□
				↑ *S1*						
Step 4	□	1	0	1	1	1	0	□	□	□
					↑ *S0*					
Step 5	□	1	0	1	1	1	0	□	□	□
						↑ *S1*				
Step 6	□	1	0	1	1	1	0	□	□	□
							↑ *S0*			
Step 7	□	1	0	1	1	1	0	□	□	□
								↑ *S0*		
Step 8	□	1	0	1	1	1	0	0	□	□
									↑ *SH*	

Universal Turing Machines

- For each operation a different Turing machine has to be created, so this is not ideal.
- For a Turing machine, the state transition diagram / function / FSM are the instructions so is separate from the tape
- A Universal Turing machine is a Turing machine that can execute another Turing machine. The instructions of the Turing machine are stored on the tape.
- This model of computing is what is used in modern computers today where both the program instructions and the data are stored in memory.