Specification Section

1 Fundamentals of Programming

1.1 Programming

4.1.1.1 A data type determines what sort of datum is being stored and how it will be handled by the program. There are several built in data types common to many programming languages:

- Integer: Any positive or negative whole number including zero.

  -2, -1, 0, 1, 2
- Real/Float: Any number with a decimal/fractional part.

  -1, $-\frac{1}{2}$, 0, 0.543543, 1
- Boolean: True or False.
- Character: An individual character (alphanumerical or symbol).

  "a", "1", "&"
- String: A sequence of characters.

  "Hello World"
- Date/Time: A date or time.

  28.05.2016 12:39:40:056
- Pointer/ Reference: The location of a value in memory

  pointer = 1023 would refer to the value in memory address 1023
- Records: A collection of items which be of different data types which are related.

  [{Name: "Nathan", age: 18} ,{Name: "Serena", age: 18}]
- Arrays: A collection of items of the same data type.

  ["Ryan", "Helen", "Luke","Giorgia"]

A user defined data type is a data type that is made of built-in data types (A data type that is provided within the programming language being used). In python you can write your own data types by writing type methods in C, or simply creating a class which does what you want.

4.1.1.2 Within a program there is usually a combination of the following statement types:

- Variable declaration: The process of defining a variable in terms of its data type and identifier (variable name).
- Constant declaration: The process of defining a constant in terms of its data type and identifier (constant name).
- Assignment: Giving a value to a variable or constant.
- Iteration: The principle of repeating processes.
- Selection: The principle of choosing what action to take based on certain criteria.
- Subroutine (AKA Procedure): A named block of code designed to carry out a specific task.
- Function: A subroutine which returns a value.

In python, all of the variables and constants are dynamically typed, meaning you don't have to worry about declaring a data type, as it is taken care of by the programming language.
Within imperative programs the combining of the principles sequencing, iteration and selection are basic to all of them.
Definite iteration is when a process repeats a set amount of time, for example:

```python
for x in  range(10):
    print("hi")
```

This would print `hi` 10 times. Indefinite iteration is a process that repeats until a certain condition is met. This can be done in two ways, *Method 1* with the condition at the beginning, and *Method 2* with the condition at the end. The implications of these two methods is that *Method 2* forces the loop to be done once, whereas the loop in method one may never be done. This can be shown using an example using both methods:

Method 1

```python
x = 5
while x < 5:
    print(x)
```

Method 2

```python
x = 5
while True:
    print(x)
    if not (x<5):
        break
```

Method one results in nothing being printed as `x` starts as `5` so `x<5` `False`, so the loop never runs, so `x` isn't printed. Method two results in `5` being printed because `True` is always `True`, so the while loop runs printing `x` which is equal to `5`, the if statement then resolves to be true (`x<5` is `False` so `not x<5` is true) so the while loop `break`s.
Nesting is placing one set of instructions within another set of instructions, the most common use of nesting is nested selection (`If...Elif...End`) and nested iteration(A for loop within a for loop).
Within a program, the use of meaningful identifier names is encouraged due to the following reasons:

- It's easier to debug (correct) code.
- Easier for others to understand when working on a large project.
- Easier to update the code.

4.1.1.3 The basic arithmetic operations are:

- Addition `a + b`

  `2 + 3 = 5`
- Subtraction `a - b`

  `2 - 3 = -1`
- Multiplication `a * b`

  `2 * 3 = 6`
- Real/Float Division `a / b`

  `2 / 3 = 0.6666666666`
- Integer Division: The result is the truncated integer of the result. `a // b`

  `17 // 3 = 5`
- Modulus `a % b` (AKA remainder)

  `17 % 3 = 2`
- Exponentiation (a^b) `a ** b`

  `2**3 = 8`
- rounding `round(a)`

  `round(0.6666666666)=1`, `round(1.4352534234) = 1`
- truncation `Math.trunc(a)`

  `Math.trunc(0.6666666666)=0`, `Math.trunc(1.4352534234) = 1`

4.1.1.4 Relational operations are expressions that compare two values. Some common Relational Operations are:

- equal to (==)
- not equal to (`!=`)
- less than (<)
- greater than (>)
- less than or equal to (<=)
- greater than or equal to (>=)

4.1.1.5 Boolean operations are expressions that return the result `True` or `False`. Some common Boolean Operations are:

- AND: Returns `True` if both inputs are true.
- OR: Returns `True` if either of its inputs are true.
- NOT: Negates (inverses) the input, `True` → `False`, `False` → `True`
- XOR: Returns `True` if either of its inputs are true but not if both are true.

4.1.1.6 A constant is an item of data whose value does not change whereas a variable is an item of data whose value could change while the program is being run. Named constants are useful because you can easily use them throughout the program, and don't have to worry about the initial value, also you can easily change it by changing the assignment and declaration of the constant.

4.1.1.7 There are several ways to manipulate and convert strings from one data type to another. Some examples of string handling functions are:

- Length: Returns the number of characters within a given string
  `len("Hello World")=11`.
- Position: Returns the position of any character or string within another string
  `"Hello World".index("World")=6`.
- Substring: Returns a string contained within another string.
  `"Hello World"[0:5]="Hello"`.
- Concatenation: returns the result of Adding two strings together
  `"Hello"+"World"="HelloWorld"`.
- Character→Character Codes: Converts a character to a character code (a binary representation if a particular letter, number of special character).
  `ord("A")=65`
- Character Codes→Character: Converts a character code to a character.
  `chr(65)="A"`
- String Conversion Operations
  - String to Integer:
    `int("2")=2`
  - String to Real/Float:
    `float("2.432")=2.432`
  - String to Date/Time:
    ```
    datetime.datetime.strptime('5 May 2016', '%d %b %Y')=
                                datetime.datetime(2016, 5, 5, 0, 0)
    ```
  - Integer to String:
    `str(2)="2"`
  - Float to String:
    `str(2.432)=2.432`
  - Date/Time to String:
    `time.strftime("%d/%m/%Y",time.localtime())="28/05/2016"`

4.1.1.8 In python random number generator functions are all contained within the module `random` and therefore requires us to import it using `import random`. There are several functions within this module, but the three most important functions are:

- `random.random()`: produces a random real number between 0 and 1
- `random.randint(a,b)`: produces a random integer between `a` and `b`.
- `random.sample(population, k)`: Chooses `k` unique random elements from a `population`.

4.1.1.9 Exception Handling is the process of dealing with events that cause the current subroutine/ procedure to stop. In general this is done by:

1. An error is thrown causing the current subroutine to stop.
2. The current state of the subroutine is saved.
3. The exception handling (or catch) block is executed to take care of the error.

4. the normal subroutine can continue from where it left off.

In python, exception handling is done by using `try` and `except`. Here is a relatively simple example:

```python
Age = input("Please Input Your Age: ")

try:
    Age = int(Age)
except:
    print("Age is not an Integer please try again.")
else:
    print("Your age is %i"%(Age))
```

What this code does is it first asks the user to input their age. We then go into the exception handling part where the code tries to make the input an integer. If an error occurs then the program prints `"Age is not an Integer please try again."`, if no errors occur, then the program prints out the age inputted at the start of the program.

A subroutine is self-contained and it carries out one or more related processes, subroutines must be given unique identifiers or names, which means that once they have been written they can be called using their name at any time while the program is being run. Subroutines can be written to handle events (something that happens during runtime).
The benefits of using subroutines are as follows:

- They can be called at any time.
- They allow for an easy overview of the program.
- Can use a top-down approach to develop a project.
- Easier to debug as each subroutine is self-contained.
- Large projects can be developed by multiple programmers

4.1.1.11 A Subroutine often has parameters and Argument. Parameters are pieces of data that represents data to be passed into a subroutine and an argument is a piece of data that is passed into the subroutine. For example if you defined a subroutine `LoadGame(Filename, Board)` Filename and Board are parameters, later when it is called as `LoadGame(TRAININGGAME, Board)` the variables TRAINNINGGAME and Board are the arguments. To pass the arguments into the subroutine a block interface is used, which is code that describes the data being passed into the subroutine.

4.1.1.12 To define a subroutine/ function in python we use the keyword `def` and to add arguments brackets are used after the subroutine name. so if we wanted to define a function named Add_Contact which has the parameters Name, and Address, we would write `def Add_Contact(Name, Address):` for the function to return a value you simply use `return` followed by the data you want the function to return to the calling routine.

4.1.1.13 Within a subroutine (in python) any variable that isn't declared as a global variable, is considered a local variable, meaning that it only exists within the subroutine, and once the subroutine has finished, the variable would no longer exist, so they cannot be accessed outside of the subroutine. There are three main benefits to this which are:

- Can't inadvertently change the value being stored elsewhere in the program.
- Use the same identifier in several places and have them be consider different variables.
- Free up memory as each time a local variable is finished with it is removed from memory.

4.1.1.14 The difference between a local and global variable is that a local variable has a limited existence within a subroutine or function in which it was declared whereas a global variable can be used anywhere in the program.

4.1.1.15 Whenever a program is being executed, it will have to deal with functions from on part of the program returning values to other parts of the program. To deal with this, we use a call stack which takes care of all the times a function is called and where to return the values of the function. The call stack in itself is made up of stack frames, which in themselves are made of three main parts:

- The arguments to be passed
- The address the final output should be returned to (the return address)
- The local variables of the subroutine

There is another part of the stack frame called the frame pointer which points to the part of the just before the function is called, which often means it will be before the local variables of the subroutine, but after the return address, the frame pointer is useful as not all functions will have the same number of local variables, so the stack size can vary, so it is often useful to have this stack pointer to easily be able to restore the stack to the state before the function was called.

4.1.1.16 Recursion is the process of a subroutine calling itself in order to complete a task. The most simple example of recursion is to calculate n factorial ($n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$).

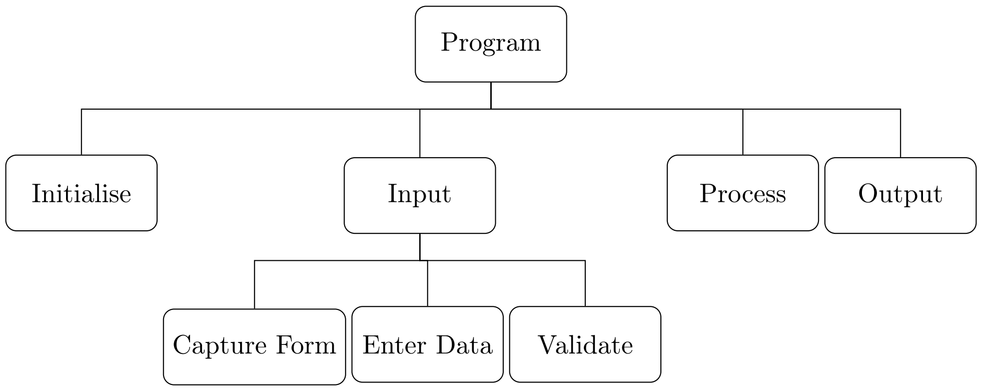
```python
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```

In this example it is clear to see that there's a base case (a case where you return a value that does not call itself), which in this case is return 0 if n equals 0, and a general case (the case that calls upon the function itself) which in this case was return $n \times (n-1)!$

1.2 Programming Paradigms

4.1.2.1 A programming paradigm can be thought of as the way in which we structure our programs. A procedural programming paradigm involves us just writing out step by step instructions for our program to follow, and then executing our program. In object oriented programs, we deal with objects, the way in which they are related to one another and the way in which they are allowed to interact with one another, which is very useful when trying to model groups of things such as the animal kingdom.

4.1.2.2 Hierarchy or Structure Charts use a top-down approach to explain how a program is put together, meaning it starts from the program name and breaks the problem down into smaller pieces. A Structure Chart differs from a Hierarchy Chart as a Structure Chart shows how data flows through a system, whereas a Hierarchy Chart does not. An example of a Hierarchy Chart is as follows:



A flowchart is a diagram using standard symbols that describes a process or system. A system flowchart is a diagram that shows individual processes within a system. It often possible to create just one flowchart that shows the entire system, but this is not always a good idea as modern programs can be very large and putting every process on to one flowchart might make it too complex to be of any real use. This can be fixed by having multiple flowcharts for the multiple systems.
Pseudo-code is a method of writing code that does not require knowledge of a particular programming language without having to worry about syntax or constructs. The only true rule of Pseudo-code is that it has to be internally consistent, for example if you write `print` in one place and then write `output`, this is considered bad practice and also makes it harder to convert it to a programming language later.

Pseudo-code can be used at many levels of detail meaning it is up to the programmer to decide what level of detail is appropriate to the project they are planning to do. One of the major benefits of using Pseudo-code is it allows the programmer to see how his code may eventually be laid out.
Naming Conventions is the process of giving meaningful names to subroutines, functions, variables and other user-defined features in a program. Before coding, a list of all the variables, including their data type and scope (Global or local) should be made. A similar procedure should also be carried out for all functions and subroutines to be featured within the program.
When writing the actual code, you should try and make your program as programmer-friendly as possible with the use of code layout and comments, examples of this would be:

- Comments to show the purpose of an algorithm.
- Comments to show the purpose of each line.
- Sensible variable names.
- Indenting the contents of loops and subroutines.

After the code is initially written, debugging will often have to occur. This can be done using a dry run and a trace table. A dry run is the process of stepping through each line of code to see what will happen before a program is run, a trace table is a method of recording the result of each step that takes place when dry running code.

4.1.2.3 Object-Oriented Programming is a way of programming in which we try and organise objects in a way that reflects the real world. There are some basic concepts that we should define first:

- class
- object
- instantiation

A class is used to define all of the properties (the defining features of the class, the variables) and methods (the things that the class can do, the functions) that are core to a group of similar objects (in the non computing sense). An object is an instance of a class so for example, if we were to say that the human race was a class, you would be an object of that class. Instantiation is the process of creating an object from a class; following our humans example, giving birth would be instantiation. An example of the difference in python would be

```python
class Humans:
    def __init__(self,age,height,name):
        self.age = age
        self.height = height
        self.name = name

nathan = Humans(18,1.95,"Nathan")
```

Here we can see that Humans is the class, Nathan is an object from the class humans, and the way that we instantiate Human objects is through a constructor (which constructs an object) like: Humans(18,1.95,"Nathan")
One of the first concepts within Object Oriented Programming we will talk about is encapsulation. This is the idea that the properties and the methods that manipulates these properties should be in the same class, meaning the class is self contained. This means it is less likely that the property of a class will be accidentally affected by other parts of the program. This process may also be called information hiding as the properties are available within the class but not outside it.
Another concept is the concept of inheritance, this is the idea that a parent class can share its properties and its methods with a child/ sub class. With inheritance, you may consider starting off with a base class and then building children off of that. This base class should contain properties and methods that are common to all of its children. With our humans example, we could re-write the code so that it is a child of a class called Animals and let Animals also have other children classes. A coding example of this would be:

```python
class Animals:
    animals = []
    nextID = 0

    def __init__(self,age,height):
        self.id = Animals.nextID
        self._age = age
        self._height = height
        Animals.nextID+=1
        Animals.animals.append(self)

    def getAge(self):
        return self._age

    def getheight(self):
        return self._height

class Humans(Animals):
    def __init__(self,age,height,name):
        Animals.__init__(self,age,height)
        self.__name = name

    def speak(self):
        print("hello")

    def getName(self):
        return self.__name

class Dog(Animals):
    def __init__(self,age,height):
        Animals.__init__(self,age,height)

    def Bark(self):
        print("bark")

```

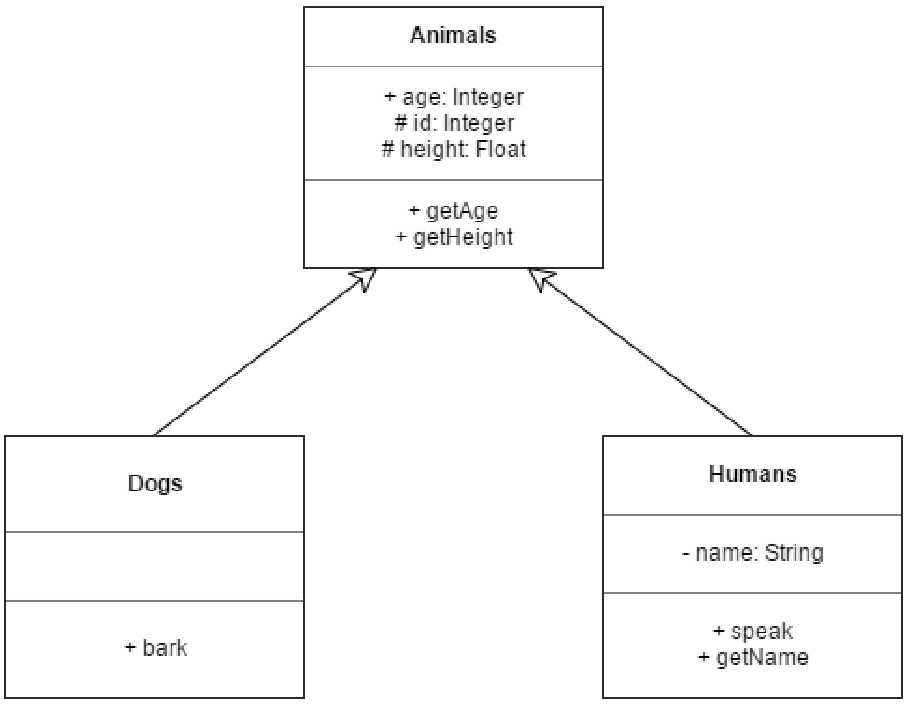
Inheritance can be shown using a diagram. The class diagram follows the following rules.

- It is hierarchical in structure, with the base class on top.
- The arrows indicate direction of inheritance
- Each class is represented by a box containing three sections

  Name

  Properties

  Methods
- If class A inherits from class B, a hollowed triangle arrow drawn from class A to class B.
- If object A is a composite aggregate of class B and C, a filled diamond arrow drawn from class B to class A and from class C to class A.
- If object A is an associate aggregate of class B and C, a hollowed diamond arrow drawn from class B to class A and from class C to class A.
- If a class method/ property is private (can only be accessed within the class), prefix it with a -
- If a class method/ property is public (can be accessed anywhere within the program), prefix it with a +
- If a class method/ property is protected (can be accessed by the class and all of its children), prefix it with a #



The third concept you need to know about is aggregation, this is a way of creating new objects (meaning 2 or more) from objects that already exist. There are two types of aggregation, **association aggregation** and **composition aggregation**. With composition aggregation (AKA composition) this implies that the one "parent" object owns the other "child" objects, an example would be a university and its departments, if the university were to shut down, then all of the departments would also have to shut down as the university owns the departments. With association aggregation if the parent object is destroyed, this does not necessarily mean the child objects would be destroyed. An example would be a department and its professors, if the department is shut down, the professors still exist, they just have to find work else where.
The final concept is polymorphism, this is the idea that a method defined in a base class can be redefined in classes that inherit from it, and thus be used in different ways. When the method defined within the sub class takes precedence over the same method defined in the base class, this is called overriding.
There are several advantages to using an object-oriented programming paradigm:

- It is easier to follow a modular approach to programming, thus making editing and appending programs easier.
- Easy to add functionality to a module.
- Modular design allows teams of programmers to easily work on self-contained modules.
- Inheritance means code is made more easily reusable throughout the program.
- Less likely to cause bugs as only edit something within one class.
- Libraries of classes can be created for easy code reuse.

There are 3 design principles within object oriented programming:

- Encapsulate what varies

  This is basically saying to break the problem down into as many classes as necessary to represent the problem in a way that adequately models the world around us. If you find a class should be broken up further, then break it up.

- Favour composition over inheritance

  For inheritance you would need to create a new class that inherits from other classes, whereas with composition or association aggregation, if you know the original classes work, you just need to instantiate the classes and combine the objects, which is less error prone.

- Program to interfaces not to implementations

  Program classes so that if they are related in some way they share methods that will carry out similar functions on similar pieces of data (if you want some experience with interfaces, I would suggest looking into learning some basic java) instead of creating many classes with lots of different classes which carry out the same function, but with different method names. (An example of where I broke this above is with the speak method in Humans and the bark method in Dogs, I could have easily moved the speak method into the Animals class and then overridden them in the sub classes, which would make the classes more logically coherent).

For the exam, you should be able to write object oriented programs and have experience in coding the following:

- Abstract methods

  A method that is declared but has no implementation

- Virtual methods

  A method defined in the base class which can be overridden. Its purpose is to allow the program on run time to decide at run-time what version of the method to run based on the instance used.

- Static methods
- Inheritance
- Aggregation
- Polymorphism
- Public, private and protected specifiers

Below I'm going to show a quick example of how to do each of the above in python
Abstract methods

```
#Abstract Base Class
import abc

Class A:
    @abc.abstractMethod
    def methodToDefine(self, input):
        return
```

Virtual methods - thankfully in python all methods are *technically* virtual due to duck typing (Google it).
Static methods

```
Class A:
    @staticmethod
    def printHi():
        print("Hi")
```

Inheritance, where class B inherits from class A

```
class A:
    pass

class B(A):
    pass
```

Association Aggregation, where class A and class B make class C

```
class A:
    pass

class B:
    pass

class C:
    def __init__(self, a1, b1)
        self.a = a1
        self.b = b1

```

```python
obj_a = A()
obj_b = B()
obj_c = C(a,b)
```

Composition Aggregation, where class A and class B make class C

```python
class A:
    pass

class B:
    pass

class C:
    def __init__(self):
        self.a = A()
        self.b = B()
```

Polymorphism

```python
class A:
    def printWelcoming(self):
        print("Hi")

class B(A):
    def printWelcoming(self):
        print("Hello")
```

Public, Private, and Protected specifiers

```python
class A:
    def __init__(self,a1,b1,c1)
        """self.a is public"""
        self.a = a1
        """self.b is protected"""
        self._b = b1
        """self.c is private"""
        self.__c = c1
```

2 Fundamentals of Data Structure

2.1 Data Structures and Abstract Data Types

4.2.1.1 A data structure is any method used to store data in an organised and accessible format, they normally contain data that are related, different data structures allow for different data manipulations which means different data structures are use for different types of applications. For example an array may be useful to store a list of names whereas a textfile may be used to store information for a database.

An array is a set of related data items stored under a single identifier, they can have one or more dimensions, all elements are often of the same type(homogeneous). An array most commonly has either one dimension (which can be useful to represent vectors) which can be visualised using a list, or two dimensions (which is useful for representing a matrix) which can be visualised using a two-dimensional table. In python, instead of arrays we use lists, which have a few minute differences to standard arrays (python lists are heterogeneous - they can store data of different types) but can be used in the same way as arrays. Some uses of lists are as follows:

```python
Studentname = ["Derrick","Gill","Jamal","Lois"]
Studentname[1]
'Gill'

ArrayAdd=[[0,1,2],[1,2,3],[2,3,4]]
ArrayAdd[1][2]
3
```

4.2.1.3 Files are used to store many different types of data meaning that many different file types are needed to store all of these different types of data. Many file types are portable, meaning that they can be used on many different platforms, the two most common portable file types when programming are text files (which is a file that contains human readable characters) and binary files (which stores data as 1s and 0s). One line on a text file may be referred to as a record, and the different items of data stored within the record are called the fields.
All files have an internal structure which allows them to store data efficiently, there are two

common structures that are used to store data These are:
Tab-delimited text (txt) file:

```
Sara     Phillips     sphillips0@google.co.jp Female   117.135.192.97
Laura    Harvey   lharvey1@utexas.edu Female   62.114.62.185
Eugene   Wells    ewells2@weibo.com    Male     119.176.45.229
Helen    Jordan   hjordan3@geocities.jp    Female   81.49.64.62
Shirley  Weaver   sweaver4@pbs.org     Female   218.20.41.34
```

Comma separated values (csv):

```
Sara,Phillips,sphillips0@google.co.jp,Female,117.135.192.97
Laura,Harvey,lharvey1@utexas.edu,Female,62.114.62.185
Eugene,Wells,ewells2@weibo.com,Male,119.176.45.229
Helen,Jordan,hjordan3@geocities.jp,Female,81.49.64.62
Shirley,Weaver,sweaver4@pbs.org,Female,218.20.41.34
```

To read and write to csv, we can use the python module csv:

```python
import csv
file= open("Contacts.csv","a+",newline='')
Reader = csv.reader(file) # This reads the contents of the file
Writer = csv.writer(file) # This creates an object which allows us to write to the
    file.
file.write("\n")
Writer.writerow(['Joe', 'Shmuck', 'JShmuck3D@hotmail.com', 'Male','162.148.10.205'
    ])
file.close()
```

Binary files contain binary codes and usually contain some header information that describes what these represent, binary files are not easily readable by a human, but can quickly be interpreted by a program. For example, the PNG image file is a binary file, can be used in a range of applications and requires less memory than some other image formats. Many program files (executables) are binary files so they can be used on other platforms. The two main actions you might want to perform on binary files are to read and write data from and to it.

4.2.1.4 An abstract data is the conceptual model of how data should be stored and the operations that can be done on this data. Data structures are the physical implementations of these abstract data types within a programming language. There are a large range of abstract data types, and the ones needed for the exam are as follows:

- Queue
- Stack
- List
- Graph
- Tree
- Hash Table
- Dictionary
- Vector

When considering data structures, they can be split up into two groups: static, and dynamic. Static data structures can only hold use a certain amount of memory, usually defined by the programmer, whereas dynamic data structures can change in size, using more or less memory as needed. Dynamic and data structures have their independant advantages and disadvantages:

Static Data Structures	**Dynamic Data structures**
Inefficient as memory is allocated that may not be needed	Efficient as the amount of memory used varies as needed
Fast access to each element of data as the memory locations are fixed when the program is written, thus they will be contiguous	Slower access to each element as the memory is allocated at run time so may be fragmented.
Structures are a fized size, making them more predictable to work with.	Structures vary in size so there needs to be a mechanism for knowing the size of the current structure.

2.2 Queues

4.2.2.1 The queue is a FIFO (First in First Out) structure (meaning the first value into the structure, will be the first out). A queue acts like a queue in a shopping market, the first person into the queue, will be the first to be serviced.
To implement a queue, we use a front and rear pointer to represent the front and back of the queue respectively. To explain the general operations of a queue we will use an example on a small scale with a queue (capable of storing a max of 6 items) with some values already in it:

Front Pointer			Rear Pointer		
“Nathan”	“Tashy”	“Giorgia”	“Ryan”		

If we wanted to add to this queue (AKA enqueue), we would add to the end of the queue, and then move the rear pointer to the address of the new item. So in this example if we were to add the name “Helen” to the queue, the structure would become:

Front Pointer				Rear Pointer	
“Nathan”	“Tashy”	“Giorgia”	“Ryan”	“Helen”	

If we wanted to delete from the queue (AKA dequeue), we always delete from the front, and the front pointer moves on to the next item, so in this case if we were to delete an item, the queue would become:

	Front Pointer			Rear Pointer	
	“Tashy”	“Giorgia”	“Ryan”	“Helen”	

The example above shows an implementation of a queue called a linear queue, where the queue can be visualised as a straight line. Other implementations of a queue include the circular queue and priority queue (although this structure varies slightly from a normal queue in that it adds a priority attribute to each element).
With the linear queue, it is possible that if we were to implement it using a static data structure such as an array, it is possible that the queue has no elements and is thus empty, or the queue has used all the elements in the array, thus being full. tests are needed for both of these scenarios, as well as the name, maximum size, and the position of the pointers when the queue is initialised.
You may have also noticed that in our example, if two more names were added to the array, then the rear pointer would be out of the range of the array and cause an error. A fix for this could be to shift all the elements of the array back when an element is removed, however this could be a long operation if they are using a longer queue. Another way to solve this problem is to use a circular queue.
A solution to the problem that is caused by linear queues is to implement a circular queue, this queue uses the same underlying concept of a linear queue, however whenever the pointer would go out of bounds, it instead wraps around to the beginning of the array, causing it the array to act as if it is a circle. If we were to use an example of adding two names to the example queue, then delete a name, here is what it would look like:
Add the name Anik:

	Front Pointer				Rear Pointer
	“Tashy”	“Giorgia”	“Ryan”	“Helen”	“Anik”

Add the Name Bilal:

Rear Pointer	Front Pointer				
“Bilal”	“Tashy”	“Giorgia”	“Ryan”	“Helen”	“Anik”

Delete a name:

Rear Pointer		Front Pointer			
“Bilal”		“Giorgia”	“Ryan”	“Helen”	“Anik”

A priority queue acts like other queues but removal is now based on priority as well as position. Elements with a higher priority are removed first and if two elements have the same priority, whichever element was added first is removed first (keeping the FIFO aspect of a queue). The priority of an object is often shown via subscript and assumes that 1 is the highest priority. For example, in the following priority queue:

Front Pointer			Rear Pointer		
“Nathan”$_1$	“Giorgia”$_2$	“Ryan”$_1$	“Helen”$_3$		

The order of deletion would be:

1. Nathan
2. Ryan
3. Giorgia
4. Helen

There are two main ways to set up a priority queue:

- Add new elements at the end of the queue (after the rear pointer)

  This makes addition of new elements easier as all it has to do is increment the rear pointer and place the data to where the rear pointer points however, removal becomes more difficult as a search through the data structure must be done to find the first element with the highest priority in the list.

- Add elements in position dependant on their assigned priority, this would mean our example priority queue would be set up as:

Front Pointer			Rear Pointer		
“Nathan”$_1$	“Ryan”$_1$	“Giorgia”$_2$	“Helen”$_3$		

  This makes removal easy as all it has to do is remove the data at the front pointer and increment the front pointer however, addition becomes more complicated as a search of the data structure must be done to be able to insert the data into the correct location.

2.3 Stacks

4.2.3.1 Stacks are Last In First Out (LIFO) data structures, they are often implemented using an array and use a stack pointer to point to the top element of the stack. There are 3 main methods that are used on a stack:

1. Push

   This adds a data value to the top of the stack

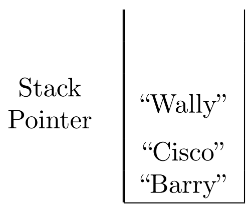
2. Pop

   This removes the top data value from the stack and return this data value

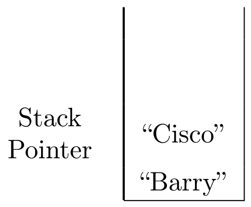
3. Peek/ Top

   This returns the top data value without removing it from the queue.

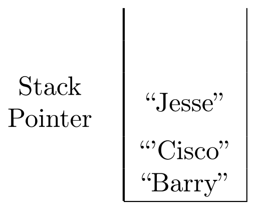
Let’s show an example of how these methods would look like on an example stack:



Now if we were to pop a value from the stack, it would become:



And the value “Wally” would be returned, next we’re gonna push the name “Jesse” onto the stack it would become:



And now if we were to perform Peek/ Top on the stack, the value “Jesse” Would be returned, and the stack would remain the same.
There are two main errors which may occur when dealing with stacks, stack overflow and stack underflow. Stack-overflow occurs when you try and add more data to a stack which is already full, whereas stack underflow occurs then you try and remove data from an empty stack.

2.4 Graphs

4.2.4.1 Graphs are a data structure which is used to represent more complex relationships.
There are many uses of a graph, such as:

- Human Networks

  Showing the relationship between different people

- Transport Networks

  For example train maps, which allows for organisation of staff and timetabling

- The internet and web

  Internet: the devices and connections between them

  Web: Sites and links between them

- Computer Science

  Find shortest path between two processor components to minimise latency

- Medical research

  Can be used to investigate the spread of viruses

- Project Management

  Can be modelled using the task a nodes and the dependencies between them as nodes

- Game theory

  Nodes represent actions and the edges are the outcomes.

Key Terms:

- Graph

  A mathematical structure that models the relationship between pairs of objects
- Weighted Graph

  A graph that has a data value labelled on each edge
- Vertex/ Node

  An object in a graph
- Edge/ Arc

  A join or relationship between two nodes
- Undirected Graph

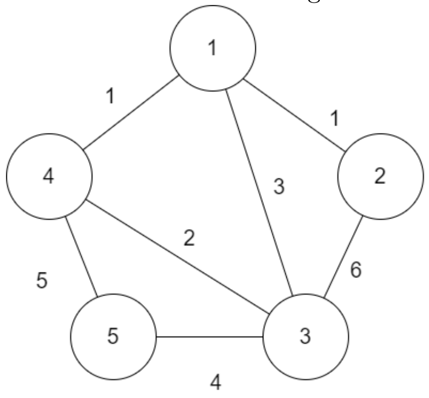
  A graph where the relationship between vertices is two-way
- Directed Graph

  A graph where the relationship between vertices is one-way

There are 2 ways of representing a graph so that it can be processed by a computer

- Adjacency Matrix
- Adjacency List

A graph can be represented using a two-dimensional matrix; this is called an adjacency matrix. Visually, this is like a table which records information about which vertices have an edge connecting them i.e which vertices are adjacent. Each vertex has a row in the table and each vertex has a column in the table. A "1" is placed in the intersection if a vertex's row with another vertex's column, if there is an edge between them. A "0" is placed in all of the other cells to denote that there is no edge connecting the two vertices.



Vertex	1	2	3	4	5
1	0	1	3	1	0
2	1	0	6	0	0
3	3	6	0	2	4
4	1	0	2	0	5
5	0	0	4	5	0

Table 1: Undirected Graph

An alternative to an adjacency matrix is an adjacency list, which can be used to indicate which vertices are next to each other.

Table 2: Undirected Graph

Vertex	Adjacent Vertices
1	2,3,4
2	1,3
3	1,2,4,5
4	1,3,5
5	3,4

Directed graphs can also be represented as an adjacency matrix or list. The method is very similar to that for an undirected graph, there are slight differences so that the matrix/ list reflect the direction of each stage.

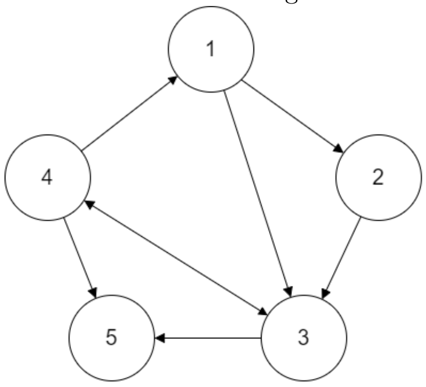


Table 3: Directed Graph

Vertex	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	1	0	1	0	1
5	0	0	0	0	0

Table 4: Directed Graph

Vertex	Adjacent Vertices
1	2,3
2	3
3	4,5
4	1,3,5
5	

It is quicker to find out if there is an edge between 2 vertices using an adjacency matrix (you only have to look at one item, in an list you may have to look at all items in a vertex's list to see if there is an edge to another vertex)
However, an adjacency list can require significantly less space as there is no wastage - it only stores information about which edges do exist in the graph, an adjacency matrix stores information about each possible edge (whether it exists or not).
For large graphs which have many vertices but few edges an adjacency list is normally preferable, if a graph has many edges an adjacency matrix is normally the best way of representing the graph.

2.5 Trees

4.2.5.1 Trees are a connected, undirected graph with no cycles, it is called the tree because of the fact that this type of data structure can easily be visualised as a hierarchical structure with branches. A rooted tree is a tree in which one of the nodes are designated from the roots, and all other edges are branching away from the root node. A binary tree is a rooted tree in which each of the node has at most two children nodes. A common use for a binary tree is as a binary search tree, a structure used to store data input in a random order in such a way that it is already sorted. It follows the following algorithm (insertion into a binary tree):

```
Store first data item in root node;
While not in empty branch:
      If the value of the new data item is less than the value in the current node, branch left;
      Else branch right;
Put value in node at end of branch;
```

2.6 Hash Tables

4.2.6.1 Hash Tables are a data structure that stores key/value pairs based on an index calculated from an algorithm. It is made up of two parts, the data and the keys. A hashing algorithm is used on the key which generates the index into which the data will be stored, this hashing algorithm will always produce the same output for a given input. Whenever you want to search for a certain key, you can perform the hashing algorithm on the key to find the index the data should be held in and thus retrieve the data in one step. Hashing algorithms are used in: Databases, Memory Addresses, Operating Systems, Encryption, Checksums, Programming.
There are a few factors to be considered when choosing a hashing algorithm:

- A numeric value must be produced from this hashing algorithm
- It must generate unique indices
- It needs to create a uniform spread of indices
- There must be enough space to store the data volumes
- Has to balance speed and complexity

The reason why we have to chose a suitable hashing algorithm is so that we can avoid collisions, this occurs when a hashing algorithm produces the same output (index) for multiple different inputs (keys). Whenever a collision occurs, there must be some sort of way to assign this key a unique index. Two main methods are used for this:

- Chaining

  If this scenario occurs, the key/value pair is added to a list at the corresponding index. If a unique address is found, the key/ value pair is simply added to that index.

- Rehashing/ Probing

  If a collision occurs one of two things is done: another hashing algorithm is used, or it looks for the next empty space and puts the data into this space.

2.7 Dictionaries

4.2.7.1 A dictionary is a data structure that is used to associate keys with values, similar to a hash table. It can also be called an associative array as it associates two sets of data.
Dictionaries can be useful for information retrieval (the tracing and recovery of specific information from stored data) for example, the sentence “To be, or not to be: that is the question:” could be represented as the following {’the’: 1, ’or’: 1, ’that’: 1, ’question’: 1, ’is’: 1, ’not’: 1, ’be’: 2, ’to’: 2}
The main python dictionary methods
instantiation

```python
dict = {}

```

insertion

```python
dict["a"] = 1
dict["b"] = 2
dict["c"] = 3
# dict = {"a"=1,"b"=2,"c"=3}
```

retrieval

```
dict["a"]
# return 1
```

modification

```
dict["a"] = 4
# dict = {"a"=4,"b"=2,"c"=3}
```

deletion

```
del dict["a"]
# dict = {"b"=2,"c"=3}
```

2.8 Vectors

4.2.8.1 In mathematics, a vector is generally defined as a quantity with direction and magnitude. There are many ways which a vector can be represented:

- a List/ 1D array

  [1,-3,2]

- a Dictionary

  {0:1,1:-3,2:2}

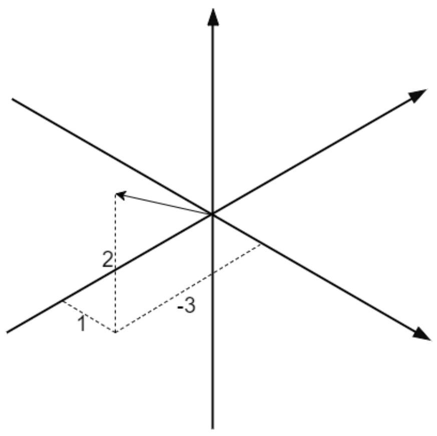
- a function

  $f : 0 \mapsto 1$

  $f : 1 \mapsto -3$

  $f : 2 \mapsto 2$

- an arrow



Each of the different parts that make up a vector are called its components, so for the examples above, they would have components 1,3, and 2. In maths notation these vectors can be written as $\begin{pmatrix} 1 \\ -3 \\ 2 \end{pmatrix}$ or $i - 3j + 2k$. From the arrow representation, you can see that you can calculate magnitude (size) of the vector using Pythagoras's Theorem so the magnitude of $i + 3j + 2k$ (which can be represented as $|i - 3j + 2k|) = \sqrt{1^2 + (-3)^2 + 2^2} = \sqrt{14}$

There are 3 simple operations that you need to be able to do with vectors:

- Vector Addition

  $$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

  $$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

  $$\mathbf{a} + \mathbf{b} = \begin{pmatrix} a_1+b_1 \\ a_2+b_2 \\ \vdots \\ a_n+b_n \end{pmatrix}$$

- Vector scalar multiplication

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

$$x \times \mathbf{a} = \begin{pmatrix} x\times a_1 \\ x\times a_2 \\ \vdots \\ x\times a_n \end{pmatrix}$$

- Dot/ Scalar Product

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$\mathbf{a} \bullet \mathbf{b} = \begin{pmatrix} a_1\times b_1 \\ a_2\times b_2 \\ \vdots \\ a_n\times b_n \end{pmatrix}$$

Another concept you need to be aware of is the convex combination, This can most easily be thought of as the space between two vectors. It can be more rigorously defined on the following way. Given 2 vectors **a** and **b** and two scalars (AKA real numbers) α and β, the convex combinations is the set of vectors in the form $\alpha\mathbf{a} + \beta\mathbf{b}$ given $\alpha \geq 0$, $\beta \geq 0$, and $\alpha + \beta = 1$.
The parity bit can be generated given two vectors over GF(2)[1] The parity bit for even parity is calculated by doing the dot product of the two vectors, one of the vectors will always be filled with 1s and the other vector is the vector you want to calculate the parity of. For example if $\mathbf{u} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$ and $\mathbf{v} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$, then $\mathbf{u} \bullet \mathbf{v} = 1 \times 1 + 1 \times 0 + 1 \times 0 + 1 \times 1 = 1 + 1 = 0$ therefore the parity bit is 0.

3 Fundamentals of Algorithms

3.1 Graph-traversal

4.3.1.1 There are two ways to traverse a graph:

- Depth First Search

  In a depth first traversal you first try and go as deep into the graph as possible and once you reach a dead end, you backtrack till you reach a node where one of its neighbours hasn't been visited. It is often called recursively. This is useful for generating and solving mazes.

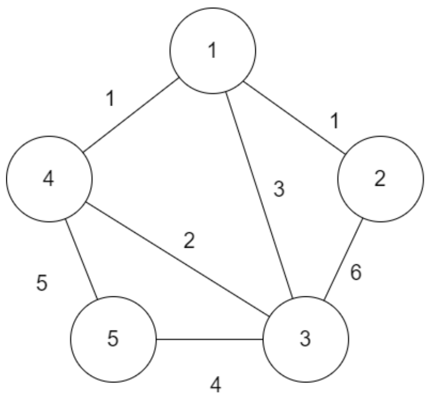
- Breadth first Search

  In a breadth first traversal, you always try and visit the nodes closest to the previously visited nodes. A queue is often used to keep track of previously visited nodes. This is useful for finding the shortest distance between two nodes for an unweighted graph.

[1]to briefly go into this, a Galois field of two elements, GF(2), is simply a system where only the numbers 1 and two are used, this is useful as it makes it easier to represent bitwise operations as arithmetic operations instead, XOR is addition/subtraction, AND is multiplication, there tables would be as follows

$\times$	0	1
0	0	0
1	0	1

$+$	0	1
0	0	1
1	1	0



Using the graph above, we will demonstrate both algorithms:

Depth First

Explanation	Current Node	Visited Nodes
Select the node to start from 1.	1	
Mark 1 as visited, choose a node connected to 1 and has not been visited (2) and recursively call the traversal routine to explore from this node.	1	1
Mark 2 as visited, choose a node connected to 2 and has not been visited (3) and recursively call the traversal routine to explore from this node.	2	1 2
Mark 3 as visited, choose a node connected to 3 and has not been visited (5) and recursively call the traversal routine to explore from this node.	3	1 2 3
Mark 5 as visited, choose a node connected to 5 and has not been visited (4) and recursively call the traversal routine to explore from this node.	5	1 2 3 5
Mark 4 as visited, all nodes connected to 4 have been visited so backtrack until you find a node that has unvisited neighbours. All the nodes have been visited so terminate.	4	1 2 3 5 4

There are two main ways to define a depth first traversal:
Recursively

```
function Depth-First-Traversal(graph, nodeV)
    mark the nodeV as discovered
    FOR all nodeW adjacent to nodeV in graph
        IF nodeW has not been visited
            Depth-First-Traversal(graph, nodeW)
        ENDIF
    ENDFOR
```

and **Using a Stack**

```
function Depth-First-Traversal(graph, nodeV)
    let Stack be a stack
    push nodeV onto Stack
    WHILE Stack is not empty
        pop an element from Stack and store it as nodeW
        IF nodeW is not marked as visited:
            mark nodeW as visited
            FOR all nodeX adjacent to nodeW in graph
                push nodeC onto Stack
            ENDFOR
        ENDIF
    ENDWHILE
```

Breadth First

Explanation	Current Node	Queue
Select the node to start from 1.	1	1
Dequeue node from the queue (1) as it is about to be fully explored, mark node (1) as fully explored	1	
Add all the nodes that are adjacent to the current node 1 and are not fully explored, and add them to the queue	1	2 3 4
Dequeue node from the queue (2) as it is about to be fully explored, mark node (2) as fully explored	1	3 4
Add all the nodes that are adjacent to the current node 2 and are not fully explored, and add them to the queue	2	3 4
Dequeue node from the queue (3) as it is about to be fully explored, mark node (3) as fully explored	3	4
Add all the nodes that are adjacent to the current node 3 and are not fully explored, and add them to the queue	3	4 5
Dequeue node from the queue (4) as it is about to be fully explored, mark node (4) as fully explored	4	5
Add all the nodes that are adjacent to the current node 4 and are not fully explored, and add them to the queue	4	5
Dequeue node from the queue (5) as it is about to be fully explored, mark node (5) as fully explored	5	
Add all the nodes that are adjacent to the current node 5 and are not fully explored, and add them to the queue	5	
The queue is now empty after checking for adjacent nodes, therefore the algorithm has finished		

The pseudo-code for this algorithm is as follows:

```
function Breadth-First-Traversal(graph, root)
    let Queue be a queue
    enqueue root onto Queue
    mark root as considered
    WHILE Queue is not empty
        dequeue an node from Queue and store it as current
        FOR all node adjacent to current in graph
            IF node is not marked as considered:
            mark node as considered
            enqueue node to Queue
        ENDIF
    ENDFOR
ENDWHILE
```

3.2 Tree-traversal

4.3.2.1 There are three ways of traversing a binary tree:

- pre-order

  In pre-order, you first visit the root node, you then traverse the left sub-tree, and then traverse the right sub-tree. This method is the same as depth first traversal. (note you visit the root node first)

  This can be used to get a prefix expression (an expression where the operators are before the values to be evaluated) from an expression tree. It can also be used to copy a tree

- in-order

  In in-order, you first traverse the left sub-tree, then you visit the root node, then you traverse the right sub-tree. (note you visit the root node second, in the middle)

This is used on binary search trees as it returns the values according to the comparator used to set up the binary tree.

- post order

  In post-order, you first traverse the left sub-tree, then traverse the right sub-tree, then visit the root node, This method is the same as the Breadth first traversal. (note you visit the root node last)

  This can be used to get a postfix expression (an expression where the operators are after the values to be evaluated) from an expression tree (converts infix to RPN). It is also used to delete nodes/ emptying a tree (as it deletes from the bottom and works its way up)

Whenever I say visit, I mean extract the data from that node. You can see that prefix used for each corresponds to when the root-node is visited, and that you always traverse the left sub-tree before you traverse the right.
The algorithms for each using python would be as follows, which returns a list in the order in which the nodes were traversed.

```python
def pre_order(root):
    traverse = [root]
    if root.getLeftChild():
        traverse += pre_order(root.getLeftChild())
    if root.getRightChild():
        traverse += pre_order(root.getRightChild())
    return traverse

def in_order(root):
    traverse = []
    if root.getLeftChild():
        traverse += in_order(root.getLeftChild())
    traverse += [root]
    if root.getRightChild():
        traverse += in_order(root.getRightChild())
    return traverse

def post_order(root):
    traverse = []
    if root.getLeftChild():
        traverse += post_order(root.getLeftChild())
    if root.getRightChild():
        traverse += post_order(root.getRightChild())
    traverse += [root]
    return traverse
```

3.3 Reverse Polish

4.3.3.1 Reverse polish notation is another name for postfix notation which is where the operators are after the values on which they operate. This is in contrast to what we normally use which is infix notation, where the operators are in between the values on which they operate on. An operator is a process within an expression, e.g + - * / etc. an operand is the value that operators work on, a value within the expression.
A simple way to do this algorithm is to first write out all the operands in the order in which they were originally in the infix expression as that order doesn't change. Then consider step by step, what calculations you would need to do to represent the expression, then for each write it out make sure the operator is after the operands, for example:
5 + ((1 + 2) * 4) -ĹŠ 3
First you add 1 and 2 together (1 2 +), then you times that by 4(1 2 + 4 *), then you add 5 to it (remember, 5 appeared first, so it has to be at the beginning of the expression, so 5 1 2 + 4 * +), then you subtract 3 from it (5 1 2 + 4 * + 3 -). The reason that the operands keep their order is due to the algorithm used for the conversion called the shunting yard algorithm, which goes as follows (tokens refer to both operands and operators):

1. While there are tokens to be read:

   Read a token.

   - If the token is a number, then push it to the output queue.
   - If the token is a function token, then push it onto the stack.

- If the token is a function argument separator (e.g., a comma):

  Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue. If no left parentheses are encountered, either the separator was misplaced or parentheses were mismatched.

- If the token is an operator, o1, then:

  while there is an operator token o2, at the top of the operator stack and either o1 is left-associative and its precedence is less than or equal to that of o2, or o1 is right associative, and has precedence less than that of o2,

  pop o2 off the operator stack, onto the output queue; at the end of iteration push o1 onto the operator stack.

- If the token is a left parenthesis (i.e. "("), then push it onto the stack.
- If the token is a right parenthesis (i.e. ")"):

  Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.

  Pop the left parenthesis from the stack, but not onto the output queue.

  If the token at the top of the stack is a function token, pop it onto the output queue.

  If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.

2. When there are no more tokens to read:

   While there are still operator tokens in the stack:

   If the operator token on the top of the stack is a parenthesis, then there are mismatched parentheses.

   Pop the operator onto the output queue.

You don't need to know the algorithm off by heart, you just need to be able to convert from infix to postfix and vice versa. To convert from postfix to infix (or just evaluate a postfix expression we use the following algorithm):

1. While there are input tokens left Read the next token from input. If the token is a value Push it onto the stack. Otherwise, the token is an operator (operator here includes both operators and functions). It is already known that the operator takes n arguments. If there are fewer than n values on the stack (Error) The user has not input sufficient values in the expression. Else, Pop the top n values from the stack. Evaluate the operator, with the values as arguments. Push the returned results, if any, back onto the stack.
2. If there is only one value in the stack That value is the result of the calculation.

It may look complicated, but it basically boils down to, read the expression from left to right, where ever you see an operator, evaluate it with the 2 preceding values, and replace them with the new value, for example, consider 5 + ((1 + 2) * 4) - 3 was converted to its postfix notation 5 1 2 + 4 * + 3 - The steps to evaluate it would be (left shows conversion to infix, right shows evaluation, both go through the same algorithm, you just decide whether or not to simplify the expression on the way down):

5 (1+2) 4 * + 3 -	5 3 4 * + 3 -
5 ((1+2)*4) + 3 -	5 12 + 3 -
5 + ((1+2)*4) 3 -	17 3 -
5 + ((1+2)*4) - 3	14

The main upside of postfix notation is that it eliminates the need for brackets within the expression, as the expression is evaluated from left to right, with operators acting on previous two values. It also means that expressions can easily be evaluated via the use of a stack. It is also used in interpreters which are based on a stack, such as with Postscript (a programming language used to describe a page's text and graphical content) and bytecode.

3.4 Searching Algorithms

4.3.4.1 In a linear search, you just check each item in a list sequentially to see if the item you are looking for is in the list. This approach isn't especially efficient, but must be used if you are dealing with a list that has no inherent order to it. It has a time complexity of $O(n)$. Sample python code would be as follows:

```python
def linear_search(lis, item):
    for ele in lis:
        if ele == item:
            return True
    return False
```

4.3.4.2 A binary search can be used on a list that has some sort of logical ordering. It works off of the premise that in an ordered list, if you were to play a game of higher or lower, always choosing the middle of the remaining list, will allow you to eliminate half of the remaining list, allowing you to quickly deduce where (or if) an element is in the list. It has a time complexity of $O(\lg(n))$. An algorithm in python would be as follows:

```python
def binary_search(lis, item):
    low_pointer = 0
    high_pointer = len(lis)-1
    while low_pointer<=high_pointer:
        mid_pointer = (low_pointer + high_pointer)//2
        if item < lis[mid_pointer]:
            high_pointer = mid_pointer-1
        if item > lis[mid_pointer]:
            low_pointer = mid_pointer+1
        if item == lis[mid_pointer]:
            return True
    return False
```

4.3.4.3 A Binary Tree is often used in programs where data is very dynamic, data is constantly being added and taken away from the system. To search through a binary tree is similar to searching an ordered list, except instead you are searching through a tree, thus you must traverse the tree and look at the items in each node. It has a time complexity of $O(\lg(n))$. Example python code would be:

```python
def binary_search(root,item):
    current_node = root
    while True:
        if current_node.getLabel() > item:
            current_node = current_node.getLeftChild()
        elif current_node.getLabel() < item:
            current_node = current_node.getRightChild()
        else:
            return True
        if current_node == None:
            return False
```

3.5 Sorting Algorithms

4.3.5.1 The bubble sort is a simple way of ordering a list. The algorithm can easily be described as follows: Until the list is ordered, go along the list comparing each element to its next element, if the current element is greater than the next element, swap the elements, else continue.

```python
def bubblesort(lis):
    for i in range(len(lis)):
        for j in range(len(lis)-1):
            if lis[j] > lis[j + 1]:
                temp = lis[j+1]
                lis[j+1]=lis[j]
                lis[j]=temp
    return lis
```

This algorithm can be written in a more sophisticated manner (so that it ends if no swaps occur, as this implies it is sorted, thus not going through unneeded iterations) as follows:

```python
def bubblesort(lis):
    while True:
        completedFlag = True
            for j in range(len(lis)-1):
                if lis[j] > lis[j + 1]:
                    temp = lis[j+1]
                    lis[j+1]=lis[j]
                    lis[j]=temp
                    completedFlag = False
        if completedFlag:
            return lis
```

A quick trace through the algorithm after every while loop for the list [3,5,1,6,2,4,7,8] would be as follows:

Pass	list[0]	list[1]	list[2]	list[3]	list[4]	list[5]	list[6]	list[7]	completedFlag
0	3	5	1	6	2	4	7	8	True (no passes have been done yet)
1	3	1	5	2	4	6	7	8	False
2	1	3	2	4	5	6	7	8	False
3	1	2	3	4	5	6	7	8	False
4	1	2	3	4	5	6	7	8	True

The bubble sort algorithm is one of the least efficient sorting algorithms and has a time complexity of $O(n^2)$, and attempts should be made to try and use better sorting algorithms.

4.3.5.2 Merge Sort is an algorithm which works by first splitting the original list into 2 halves, merge sorting each of the halves, then merging the two half lists so that it creates an ordered list, this is an example of a divide and conquer strategy to problem solving. As you can tell, this is a recursive sort, and has the base case of a single element, because a single element is already sorted. Python code for the merge sort (and a separate function for the merging) are as follows:

```
def mergesort(lis):
    if len(lis)==1:
        return lis
    else:
        lis1 = mergesort(lis[0:len(lis)//2])
        lis2 = mergesort(lis[len(lis)//2:len(lis)])
        return merge(lis1,lis2)

def merge(lis1,lis2):
    res = []
    while (len(lis1)!=0 and len(lis2)!=0):
        if lis1[0] < lis2[0]:
            res.append(lis1[0])
            del lis1[0]
        else:
            res.append(lis2[0])
            del lis2[0]
    while len(lis1)!=0:
        res.append(lis1[0])
        del lis1[0]
    while len(lis2)!=0:
        res.append(lis2[0])
        del lis2[0]
    return res
```

This is a much more efficient sorting algorithm than the bubble sort and has time complexity $O(n \lg(n))$. A brief trace of the algorithm for the list [3,5,1,6,2,4,7,8] would be as follows:

1. [3,5,1,6,2,4,7,8]
2. [3,5,1,6] [2,4,7,8]
3. [3,5] [1,6] [2,4] [7,8]
4. [3] [5] [1] [6] [2] [4] [7] [8]
5. [3,5] [1,6] [2,4] [7,8]
6. [1,3,5,6] [2,4,7,8]
7. [1,2,3,4,5,6,7,8]

3.6 Optimisation Algorithms

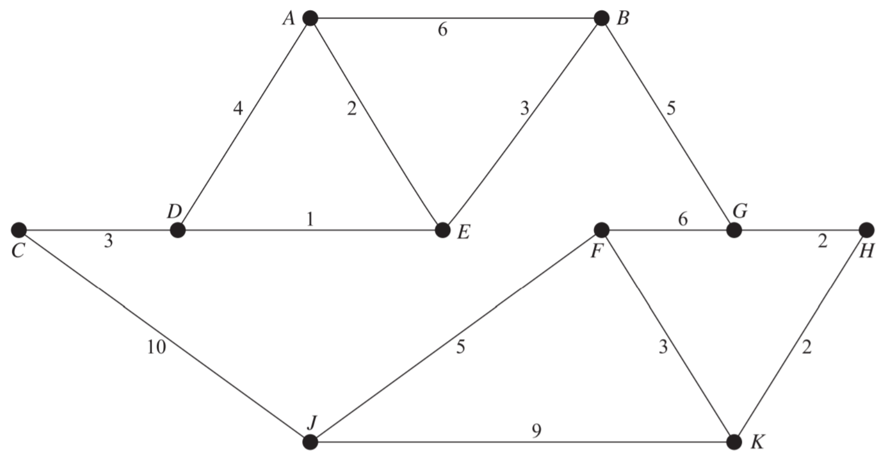
4.3.6.1 Dijkstra's shortest path algorithm is an algorithm used to find the shortest path between two nodes in a graph (mainly weighted). This algorithm has several useful applications:

- Geographic Information systems (GIS)
  - Satellite Navigation
  - Mapping software
- Telephone and Computer Network Planning
- Network Routing/ Packet Switching
- Logistics and Scheduling

Dijkstra's Algorithm goes as follows:

1. Give all of the existing nodes a value corresponding to their current perceived distance according to the following rules: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances. Compare the newly calculated perceived distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest perceived distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

For the following graph, the Trace of the algorithm from node A to node K would be (the subscript next to numbers shows the node to which the shortest path would come from):



Step	Vertex	A	B	C	D	E	F	G	H	J	K
0	A	0_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A	∞_A
1	A	0_A	6_A	∞_A	4_A	2_A	∞_A	∞_A	∞_A	∞_A	∞_A
2	E	0_A	5_E	∞_A	3_E	2_A	∞_A	∞_A	∞_A	∞_A	∞_A
3	D	0_A	5_E	6_D	3_E	2_A	∞_A	∞_A	∞_A	∞_A	∞_A
4	B	0_A	5_E	6_D	3_E	2_A	∞_A	10_B	∞_A	∞_A	∞_A
5	C	0_A	5_E	6_D	3_E	2_A	∞_A	10_B	∞_A	16_C	∞_A
6	G	0_A	5_E	6_D	3_E	2_A	16_G	10_B	12_G	16_C	∞_A
7	H	0_A	5_E	6_D	3_E	2_A	16_G	10_B	12_G	16_C	14_H
8	K	0_A	5_E	6_D	3_E	2_A	16_G	10_B	12_G	16_C	14_H

We can use this to trace back and find that the shortest route is A-E-B-G-H-K

4 Theory of Computation

4.1 Abstraction and Automation

3.4.1.1 Problem solving is the process of finding solutions to a certain problem. One of the main tools used when problem solving is the application of Logical Reasoning which is the process of using a given set of facts to determine whether new facts are true or false. To solve a problem another important step is to identify what the problem is.

3.4.1.2 An Algorithm is a sequence of steps that can be followed to complete a task and that always terminate. An example of an algorithm (written in pseudo-code) is as follows:

```
func bubblesort( var a as array )
for i from 1 to N
    for j from 0 to N - 1
      if a[j] > a[j + 1]
        swap( a[j], a[j + 1] )
end func

```

We can use a technique called hand-tracing/ dry running to work through the code and see if it works as intended. An example of a dry run is as follows:

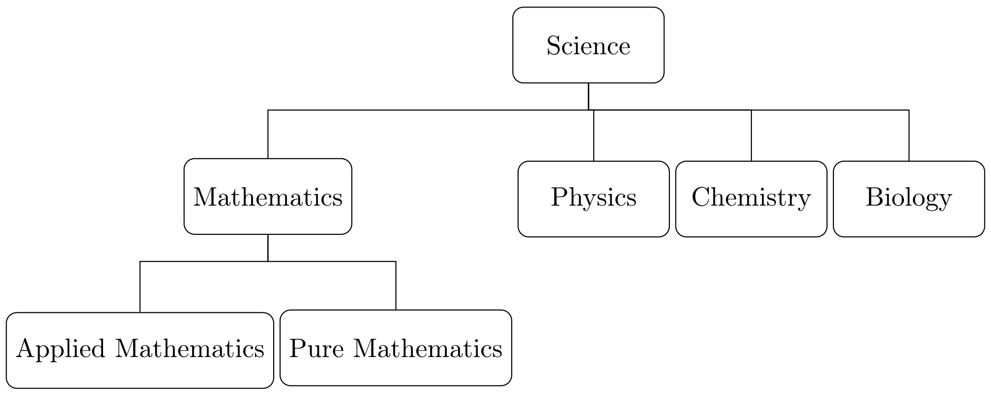
i	j	a			
0	0	3	2	1	4
1	0	2	3	1	4
1	1	2	1	3	4
1	2	2	1	3	4
2	0	1	2	3	4
2	1	1	2	3	4
2	2	1	2	3	4
3	0	1	2	3	4
3	1	1	2	3	4
3	2	1	2	3	4

This shows what happens to the data throughout the process at which the algorithm is being run.

3.4.1.3 The main concept behind abstraction is to reduce a problem to its most basic parts, its essential features. This is useful as it allows a programmer to solve the problem without having to fuss too much about the details, abstraction is also useful because it allows the solution to one problem to be implemented in other similar problems. In general, there are two main types of abstraction:

- Representational Abstraction
  - This is the process of removing unnecessary details so that only information that is required to solve the problem remains. An example of this is a train map, as this shows how the different stations are connected, but doesn't really care about actual distance or time taken.

- Abstraction by generalisation/categorisation
  - This is the concept of reducing problems by putting aspects of a problem into hierarchical categories using an "is a kind of" relationship. An example of this is could be a tree showing the different sciences, e.g:



3.4.1.4 The process of information hiding involves hiding all details about an object that do not contribute to its essential characteristics. A simple example of this is using a car, as you can control it by using the steering wheel, gearbox, pedals, etc. and don't need to know the mechanics behind it.
There are many different types of abstractions, for example:

3.4.1.5 • Procedural Abstraction

This is the concept that all solutions can be broken down into a series of procedures, an example would be a recipe.

3.4.1.6 • Functional Abstraction

This is the concept that all solutions can be broken down into reusable functions. Functions can be thought of as abstracted procedures as one can call a function without completely knows how it works.

3.4.1.7 • Data Abstraction

This is the concept of hiding how a data type is represented, making it easier to construct new data objects (called compound data objects). It also involves separating implementations of data objects and the user interface.

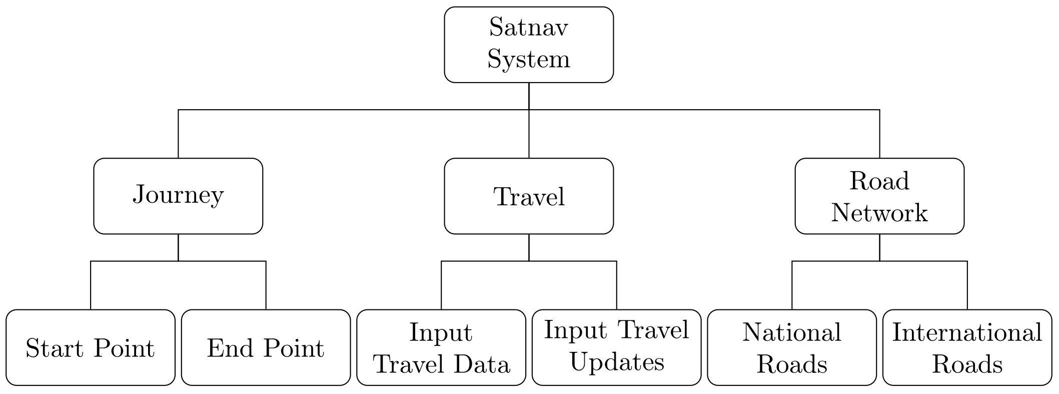
3.4.1.8 • Problem Abstraction/ Reduction

This is the process of removing unnecessary details in a problem until the underlying problem is identified to see if this is the same as a problem that has already been solved.

3.4.1.9 Decomposition is the process of breaking a large task into a series of subtasks. Procedural decomposition is the process of breaking down a task into procedures and subroutines.

3.4.1.10 Composition is the building up of a whole system from smaller units. The opposite of decomposition. This involves:

- Writing all the procedures and linking them together to create compound procedures.
- Creating data structures and combining them to form compound structures.

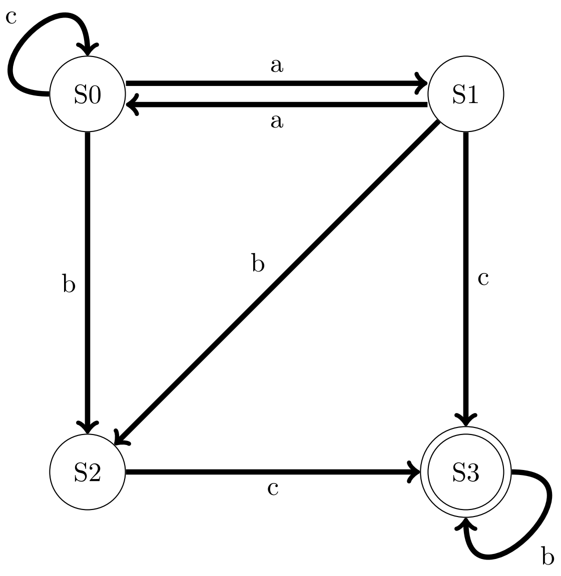


3.4.1.11 Automation is the process of creating computer models (abstraction of real world objects/ phenomena) of real-life situations and putting them into action to solve problems. This is done by:

- Understanding the problem.
- Creating Algorithm.
- Implementing the algorithms in program code.
- Implementing the models in data structures.
- Executing the code.

4.2 Regular Languages

3.4.2.1 A finite state machine is any device device that stores its current state and whose status can change as the result of an input. Mainly used as a conceptual model for designing and describing systems. A state transition diagram is a visual representation of an FSM using circles and arrows, whereas a state transition table is a tabular representation of an FSM showing input, current state, and next state. Within a state transition diagram, there may be an accepting state, represented by two concentric circles, which shows whether an input has been accepted. An FSM doesn't necessarily need an accepting state. Below is an example of both a state transition diagram and a state transition table.



State Transition Diagram

Input	Current State	Next State
c	S0	S0
a	S0	S1
b	S0	S2
a	S1	S0
b	S1	S2
c	S1	S3
c	S2	S3
b	S3	S3

State Transition Table

A Mealy Machine is a type of finite state machine which also produces an output, this output can be shown on a diagram by instead of labelling each transition with "input", you label them with "input/output", and that output is outputted when the transition it is on occurs. This can also be represented in a state transition Table by simply adding a column.

4.4.2.2 There is a lot of random maths that is part of this course that isn't taught as part of even standard A level maths, but thankfully it only covers basic concepts so. The first thing we need to do is define a set, a set is "a well-defined collection of distinct objects", as we are only likely to deal with real numbers, you can just think of it as a collection of different numbers (they don't need to be in any specific order). A set is often defined using curly brackets for example, a set called A which is the set of numbers 1,2,3,4,5 would be shown as $A = 1, 2, 3, 4, 5$. There are a few standard sets that you should know

- Real - The set of all numbers on the number line, represented as $\mathbb{R}$
- Rational - the set of all numbers that can be expressed as $\frac{a}{b}$ where a and b are both integers, represented as $\mathbb{Q}$
- Integers - The set of all whole numbers, positive and negative, represented as $\mathbb{Z}$

  $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$
- Naturals - The set of all positive, whole numbers, represented as $\mathbb{N}$

  $\mathbb{N} = \{0, 1, 2, 3, 4, 5, 6, \dots\}$

There are other ways to define a set instead of just listing the numbers, you can instead define how the set should be built, this is called set comprehension/ set building. for example set $B = \{x^2 \mid x \in \mathbb{N} \wedge x \geq 2\}$ Could be broken down as follows:

- B - the name of the set
- $\{\}$ - indicates the contents of the set
- x^2 - This indicates the values which you put into the set
- $|$ - Such that, the part after this defines the values which x can take
- $\in$ - is a member of
- $\mathbb{N}$ - The set of natural numbers
- $\wedge$ - and
- ≥ 2 - greater than or equal to 2

So to write it as one sentence, the set B is the set of all values of x^2 such that x is a member of the set of natural numbers, and x is greater than or equal to 2. ($B = \{4, 9, 16, 25, 36, 49, 64, \dots\}$). One of the more unique and important sets is the empty set ($\{\} = \varnothing$) which contains no elements. Often is is easier to use a set comprehension to define a set, especially if it is infinite, for example the set $\{0^n1^n \mid n \geq 1\} = 01, 0011, 000111, 00001111, 0000011111, \dots$. In python, sets are also defined using curly brackets.
When talking about a set, it can be either finite or infinite. A finite set is a set that has a finite number of elements, it has an upper limit (less than infinity), whereas an infinite set has an infinite number of elements, it is not finite. A set is said to be countable if it has the same cardinality as some subset of the natural numbers, or more simply put, if it can be counted using the natural numbers. This implies that all finite sets are countable, as well as sets with the same cardinality (size) as the natural numbers. This brings us onto countably infinite sets, these are infinite sets that are countable, this means the set has the same cardinality as natural numbers, implying that the set and natural numbers can me mapped in a way to form a one to one correspondence between each elements of the two sets (for an example, look up Cantor's diagonal argument). The cardinality of a set is the number of elements within the set.
There are 5 set operations that you need to know:

- Cartesian Product - Combines 2 sets to create a set of ordered pairs, for example, if $A = \{a, b, c\}$ and $B = \{1, 2, 3\}$ then $A \times B = \{(a, 1)\,(a, 2)\,(a, 3)\,(b, 1)\,(b, 2)\,(b, 3)\,(c, 1)\,(c, 2)\,(c, 3)\}$ to define more generally, $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$.

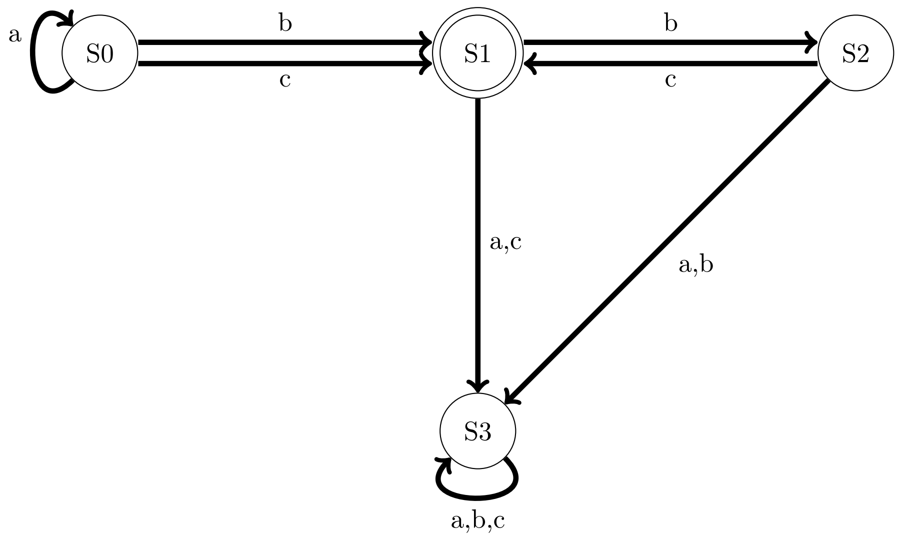
- Membership - This is when an element belongs to a set so for example if we had a set $A = \{1, 2, 3, 4\}$ then we can say that 2 is a member of the set A or in set notation $2 \in A$
- Union - This joins two sets together so that it contains all of the elements that are in each of the sets. For example, the union of the set $A = \{1, 2, 3, 4\}$ and the set $B = \{3, 4, 5, 6\}$ is $\{1, 2, 3, 4, 5, 6\}$ or in set notation $A \cup B = \{1, 2, 3, 4, 5, 6\}$. It can be thought of as an OR.
- Intersection - This joins two sets together so that it contains all of the elements that are in both sets. For example, the intersection of the set $A = \{1, 2, 3, 4\}$ and the set $B = \{3, 4, 5, 6\}$ is $\{3, 4\}$ or in set notation $A \cap B = \{3, 4\}$. It can be though of as an AND.
- Difference - This joins two sets together so that it contains all of the elements that are in one of the sets, but not in both sets. For example, the difference of the set $A = \{1, 2, 3, 4\}$ and the set $B = \{3, 4, 5, 6\}$ is $\{1, 2\}$ or in set notation $A - B = \{1, 2\}$ also $B - A = \{5, 6\}$. The symmetric difference is $(A-B) \cup (B-A) = A \ominus B = A \Delta B = 1, 2, 5, 6$ (in our example).

A set A is a subset of another set B if all elements of the set A are elements of the set B. In other words, the set A is contained inside the set B. The subset relationship is denoted as $A \subseteq B$. For example if $A = \{2, 3, 4\}$ and $B = \{1, 2, 3, 4, 5\}$ then $A \subseteq B$, this would also be true if $A = \{1, 2, 3, 4, 5\}$. A set is A proper subset of B if A is a subset of B and $A \neq B$, so for example, if $A = \{2, 3, 4\}$ and $B = \{1, 2, 3, 4, 5\}$ then $A \subset B$, this would not also be true if $A = \{1, 2, 3, 4, 5\}$ as $A = B$ therefore A can't be a proper subset of B.

4.4.2.3 A regular expression is a simple way of describing a set and allow convenient shorthand description of certain languages. Here are common regular expressions:

Regular Expression	Meaning	Strings Produced
abc	a then b then c	abc
a|b|c	a or b or c	a b c
a*bc	0 or more a then b then c	bc abc aabc aaabc
a+bc	1 or more a then b then c	abc aabc aaabc
a?bc	0 or 1 a then b then c	bc abc
a{2,5}bc	2 to 5 a then b then c	aabc aaabc aaaabc aaaaabc
(a|b)c	a or b, then c	ac bc
.bc	any character then b then c	abc bbc cbc dbc 1bc
[abc]bc	a or b or c then b then c	abc bbc cbc
[âbc]bc	(not a or b or c) then b then c	dbc ebc 1bc

Regular expressions can also be represented using a finite state machine, for example, the regular expression a*(b|c)(bc)* could be represented by the following finite state machine.



State Transition Diagram

4.4.2.4 A regular language is a language that can be represented by a regular expression, thus will also be accepted by a finite state machine

4.3 Context-free Languages

4.4.3.1 Context-free languages are languages that are defined by context-free grammars, which in turn are simply a set of rules which defines what strings are possible in a given language. We often find that languages are too complex to be represented by a regular expression or a finite state machine, thus context-free grammars are useful in these situations (e.g. try formulating a regular expression for a palindrome).
Backus-Naur form is a notation we can use to describe context free grammars, thus describing the syntax of a program. There a few characters that are important within Backus-Naur Form:

- <>

  The name within the curly bracket defines a non-terminal expression, that is an expression that can be broken down further. For example in the expression <digits>::=0|1|2|3|4|5|6|7|8|9 , <digits> would be a non-terminal, and the digits from 0 to 9 would be terminal as they can't be broken down any further.

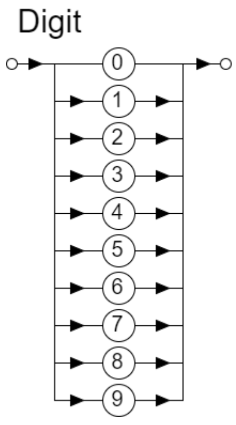
- ::=

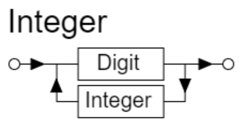
  This is used to say how non-terminal expressions are defined

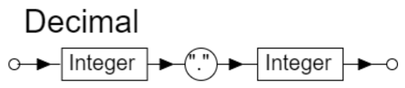
- |

  This is a way of breaking up several parts of an expression, an 'or' of the different parts.

To give a simple example which defines decimals would be as follows <decimal> ::= <integer> "." <integer> <integer> ::= <digit> | <digit> <integer> <digit> ::= 0|1|2|3|4|5|6|7|8|9
This shows that a decimal is 2 integers separated by a decimal point, an integer is either just a digit, or a digit followed by an integer (showing how context free grammars can be recursive), and a digit is defined as any natural number from 0 to 9.
This can also be described using a syntax diagram, arrows show possible choices, ellipses show terminal expressions, and rectangles show non-terminal expressions. For example, the syntax diagrams for the above BNFs would be:





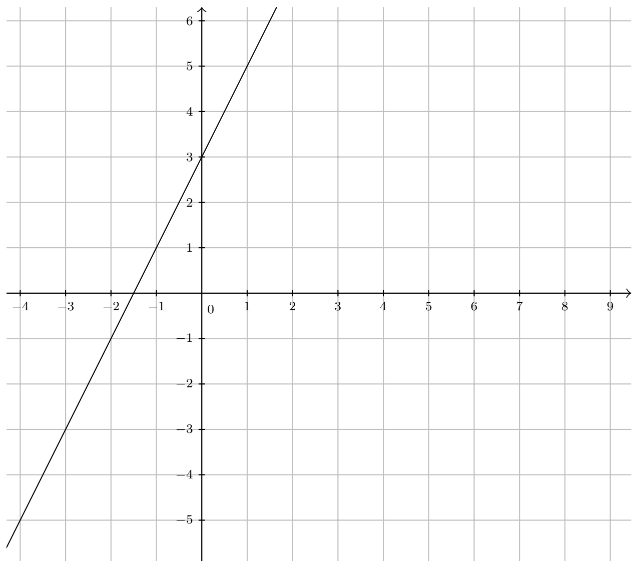


4.4 Classification of Algorithms

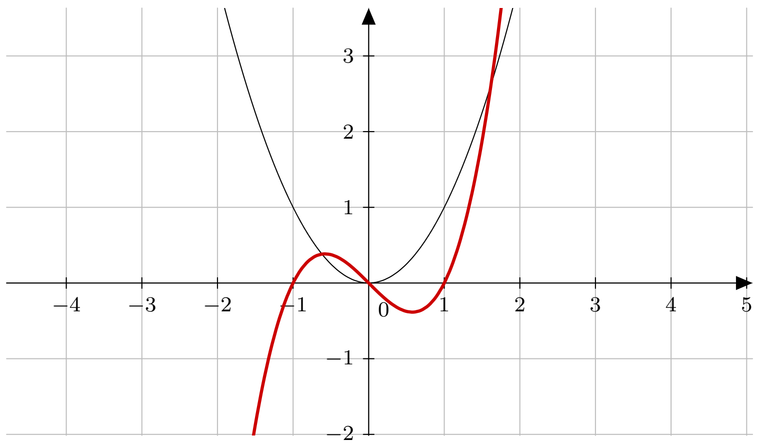
4.4.4.1 In computer science we often derive several algorithms in order to solve the same problem. It is useful to have a way for us to compare these algorithms either time-wise (how ling the algorithm takes) or space-wise (how much memory does the algorithm use).

4.4.4.2 In order to understand Big O-notation, you need to understand some basic functions (as well as some random related facts), the first thing is domain and co-domain. The domain is the set from which all the values that you can input into a function are taken and the co-domain is the set from which all of the output values can be taken. If you do A-level maths, it is similar to the concept of the range, however the co-domain can be larger than the range (the range is a subset of the co-domain). For example, take the function $f(x) = x^2$ (or $f : x \mapsto x^2$ written slightly differently) it has the domain $\mathbb{R}$ as you can put in any real number, and it can have co-domain $\mathbb{R}$ as all of its outputs are real numbers, you can see how this can be different to range, as the range would be $\{x \mid x \in \mathbb{R} \land x \geq 0\} = \mathbb{R}_0^+$. Often to fully define a function, they will state the domain and co-domain of a function in the following way, $f : \mathbb{R} \mapsto \mathbb{R}$, so if we wanted to define a new function $g(x)$ that produces the same output as $f(x)$ but has a different co-domain, we may say $g : \mathbb{R} \mapsto \mathbb{R}_0^+$
$g : x \mapsto x^2$ and this has co-domain $\mathbb{R}_0^+$
For this section, you also need to know some general graph functions:

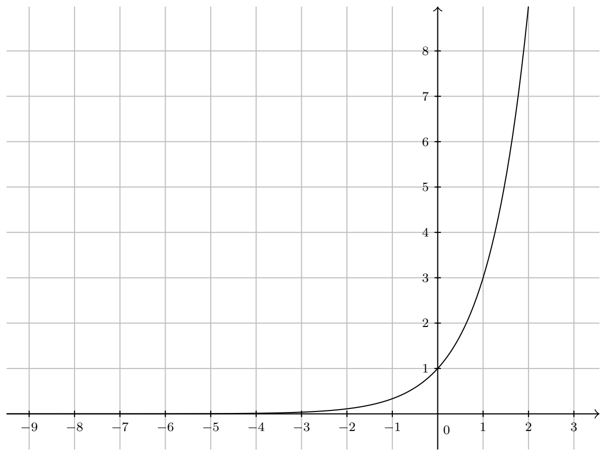
- Linear, this would be a graph of the form y = ax + b, for example $y = 2x + 3$



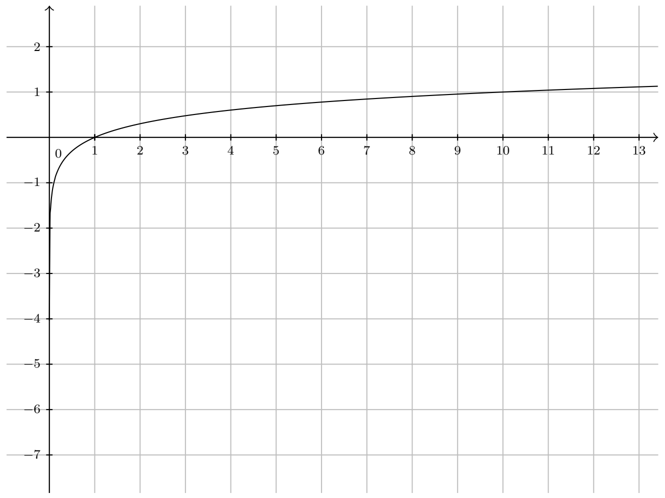
- Polynomial - An expression in terms of variable with integer powers greater than or equal to 1. If the largest power is 0 then it is constant (a), 1 then it is linear ($ax + b$), 2 then it is quadratic ($ax^2 + bx + c$), 3 then it is cubic ($ax^3 + bx^2 + cx + d$), 4 quartic, 5 quintic, etc. Below are examples of a quadratic and cubic expressions with forms $y = x^2$ and $y = x^3 - x$ respectively.



- Exponential - This an expression of the form $a \times b^x$ where a and b are constants. Below is the function 3^x:



- Logarithmic, this is an expression of the form $y = \log_a(x)$ where a is a constant. The definition of a log is the solution y to $b^y = x$, which means if $b^y = x$ then $y = \log_b(x)$. In other words its the power you have to raise a number to, to achieve another number. For example, if $2^x = 128$, then we can see that $x = 7$, therefore $x = \log_2(128) = 7$. The graph below shows $y = \log_{10}(x)$



Another thing you need to know is n factorial ($n!$). n factorial is equal to the product of all positive integers less than or equal to n, so $n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$. factorial is used for when you want to find the number of ways a set can be ranged, the number of permutations of a set. To show you how, if we want to find all of the permutations of the word "dog" we would find $6 = 3!$, to list them all: dog, dgo, odg, ogd, gdo, god. To get the logic behind it, think about it this way, there are 3 ways to pick the first letter, then 2 to pick the second, then only one option left for the last, we get $3 \times 2 \times 1 = 3! = 6$ permutations, for a set with 4 elements, you have 4 choices, then 3 choices, then 2 choices, then 1 choice, giving you $4 \times 3 \times 2 \times 1 = 4! = 24$ permutations, for a set with 5 elements $5 \times 4 \times 3 \times 2 \times 1 = 5!120$ permutations. This pattern goes on, so a set of n elements has $n!$ permutations.

4.4.4.3 Big O notation is a way to describe an algorithm time-wise or space-wise. Big O notation has 5 main classifications in terms of time complexity:

- Constant Time ($O(1)$)

  This is an algorithm that will always run in the same amount of time, no matter the size of the input. An example is accessing an array, as you get the value by referring directly to its address, thus no matter the size of the array, it will always take the same amount of time to carry out this process.

- Linear Time ($O(N)$)

  This is an algorithm that grows proportionally to the size of the input. E.g. if the size of the input doubles, the amount of time the algorithm may also doubles. If you were to graph the length of time the algorithm takes against the size of input, you would end up with a linear function, so the above example may not necessarily be true if the graph made was $y = 2x$, then is the size of the input doubled, the time taken would quadruple. A for loop over every element once will create an algorithm of linear time.

- Polynomial Time ($O(N^k)$ where k is a positive integer constant greater than 2)

  This is an algorithm that's time complexity can be upper bounded by a polynomial expression in terms of the size of the input, for the example, if the algorithm's time complexity can be expressed as $4n^2 + 6n - 4$, then in big O notation it can be written as $O(N^2)$. This can occur when iterative techniques are used, or when nested loops are used to iterate over elements, e.g. bubble sort.

- Exponential Time ($O(k^N)$ where k is a positive constant)

  This is an algorithm that's time complexity can be upper bounded by an exponential expression. Problems that require Algorithms of this time complexity to solve are called intractable problems as they become quite unwieldy as the size of the input increases.

- Logarithmic Time ($O(\log(N))$)

  This is an algorithm that's time complexity can be upper bounded by an logarithmic expression. These time complexities often show up when working on a binary tree, or when performing a binary search.

The order of time complexity from best to worst is as follows (if you want to convince yourself of this, plot graphs of each on top of each other)

1. Constant Time
2. Logarithmic Time
3. Linear Time
4. Polynomial Time
5. Exponential time

When trying to derive the time complexity of the algorithm, consider the following:

- Algorithms that require no data, and are simple assignment or comparisons, are linear time
- Algorithms that feature a loop over all of the inputs, are linear.
- Algorithms that feature multiple nested loops, all iterating over the inputs are polynomial time. The number of nested loops, indicate the order of the polynomial. e.g. bubble sort has 2 nested loops therefore it is $O(N^2)$, if we were to add a nested loop that printed the list after every stage, it would become $O(N^3)$.
- Algorithms that use recursion may have exponential time complexity (not always true, as shown with the merge sort)

- Splitting data into multiple pats, then performing actions on these parts, then repeating this, is often of logarithmic time complexity

4.4.4.4 In reality, we have limits to what we can do with an algorithm, such as the hardware (limits to RAM, and ROM), as well as the complexity of the algorithm, which limit what we can compute.

4.4.4.5 There are two ways to classify a problem:

- Tractable

  Problems that have algorithmic solutions that are of polynomial time complexity or less

- Intractable

  Problems that do not have algorithmic solutions that are of polynomial time complexity or less. To solve these problems, often a heuristic is used, this is when you use some sort of measure to guess a solution, then use an algorithm to improve your initial guess. An example is the travelling salesman problem, which is to find the shortest route to visit all of the nodes in a weighted connected graph.

4.4.4.6 There are problems that exist that cannot be solved algorithmically, meaning no matter how powerful your computer is, the problem has no algorithm solutions.

4.4.4.7 The Halting Problem is an example of a problem that cannot be solved algorithmically, as it would create a paradox. The halting problem is about trying to find an algorithm that can determine for all other algorithms whether it will end by feeding in this other algorithm. We can show that such an algorithm doesn't exist via a proof by contradiction. Let's assume that such an algorithm does exist and returns `True` if the algorithm terminates, `false` otherwise, we are going to feed in the following algorithm (let's call it the Anti-Halting Algorithm)

```
IF the output of the halting algorithm is True:
    WHILE True:
        pass
    ENDWHILE
ELSE
    HALT
ENDIF
```

1. The halting algorithm evaluates `True`, meaning the anti-halting algorithm will halt

   Then the anti-halting algorithm will go into an infinite while loop, never halting

2. The halting algorithm evaluates `False`, meaning the anti-halting algorithm will not halt

   Then the anti-halting algorithm will halt

Either way you will get some kind of contradiction, thus showing a halting algorithm cannot exist. Now you don't need to know the proof, but it is just good to understand why the halting problem is the way it is. We can gather 2 things from this:

1. There are some problems that cannot be solved algorithmically (unsolvable problems)
2. There are some problems that cannot be solved in a reasonable time frame (intractable problems)

4.5 A model of computation

4.4.5.1 A Turing Machine is a model of computation that is used to identify whether a problem is computable. Here are the basics of the turing maching:

- It is made up of a tape and a read/write head
- the tape is made up of multiple cells that act as memory
- Each cell will contain a symbol, often 0, 1 and the blank symbol (this can be many different things, including $\square$, $\diamondsuit$, or B)

- The read/ write head can read from or write to what is currently underneath it (writing to it basically overwrites its current value), and starts at the left most not blank cell.
- The tape can move left (L, $\leftarrow$) or right(R, $\rightarrow$), one cell at a time, meaning all cells are accessible to the read/write head.
- The machine can halt at any point if it reaches a halting state (a state with no outgoing transitions) or has processed all of the input

The order in which the read/ write head performs its operations are as follows:

1. **Read** the square currently underneath the read/write head
2. **Write** to the square currently underneath the read/write head
3. **Move** the read/write head left or right

To define a Turing machine, you need a start state, a halting state, ability to move the read/ write head, and a transition function, this defines what's to be written and what state to move to, given the current state and what has been read. There are 3 main ways to do this:

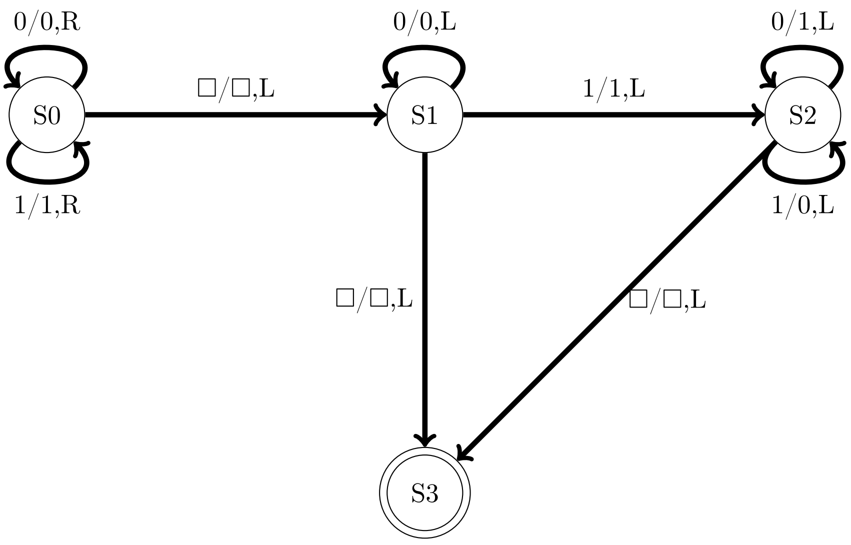
- An instruction table

  this is a tabular way to show all of the transitions within a Turing machine, for example:

State	Read	Write	Move	Next State
S0	0	0	R	S0
S0	1	1	R	S0
S0	□	□	L	S1
S1	0	0	L	S1
S1	1	1	L	S2
S1	□	□	L	S3
S2	0	1	L	S2
S2	1	0	L	S2
S2	□	□	L	S3

- A finite state machine

  A Turing machine can be represented as a finite state machine in the following way:



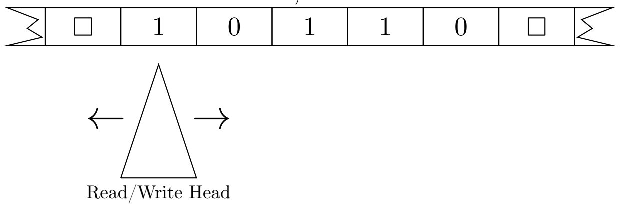
- A list of transition rules

  A Turing machine can also be defined as a list of transition rules of the following form δ(Current State, Input Symbol) = (Next State, Output Symbol, Movement). For example:

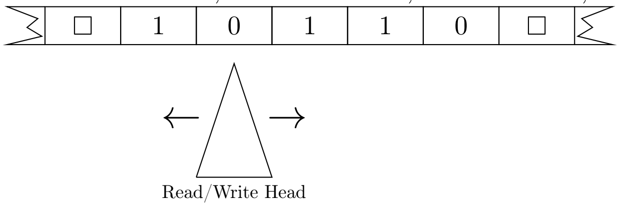
  – δ(S0,0) = (S0,0,R)
  – δ(S0,1) = (S0,1,R)

- δ(S0,□) = (S1,□,L)
- δ(S1,0) = (S1,0,L)
- δ(S1,1) = (S2,1,L)
- δ(S1,□) = (S3,□,L)
- δ(S2,0) = (S2,1,L)
- δ(S2,1) = (S2,0,L)
- δ(S2,□) = (S3,□,L)

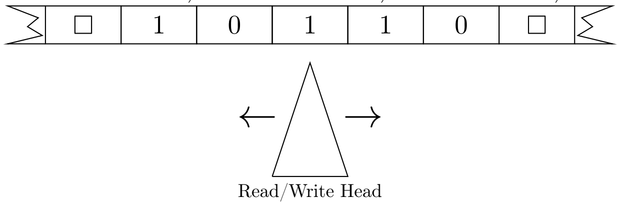
All of the above examples define the same Turing machines, so let's see what the output is for the input of 10110.
So it starts off as follows, in the state S0:



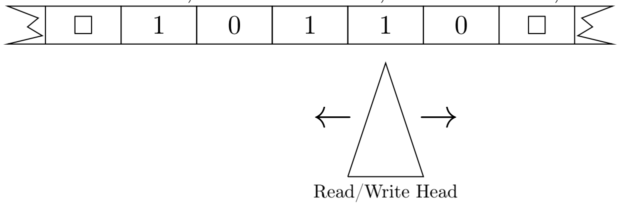
So then it reads 1, thus it writes 1, remains in S0, and moves right



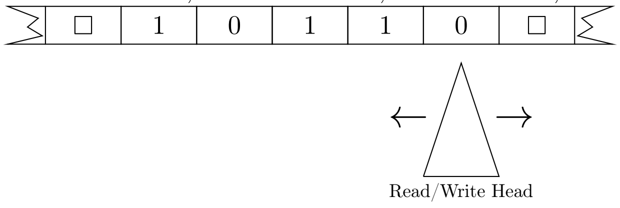
Then it reads 0, thus it writes 0, remains in S0, and moves right



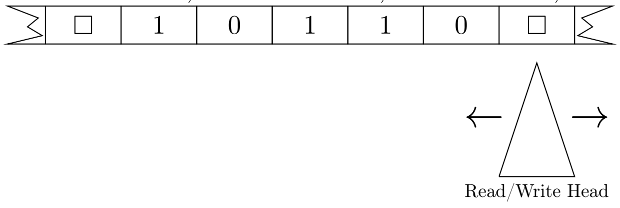
Then it reads 1, thus it writes 1, remains in S0, and moves right



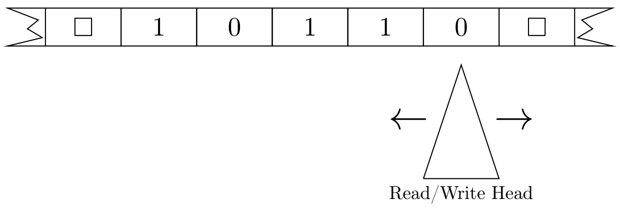
Then it reads 1, thus it writes 1, remains in S0, and moves right



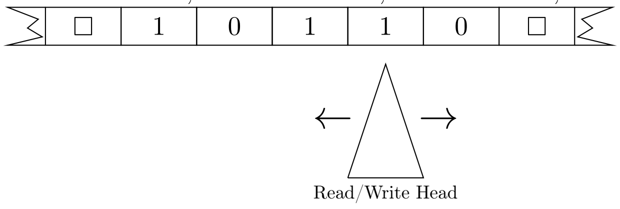
Then it reads 0, thus it writes 0, remains in S0, and moves right



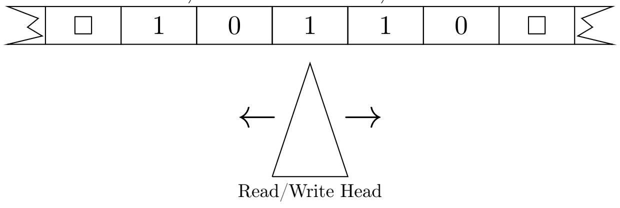
Then it reads □, thus it writes □, Moves to the S1 state, and moves left



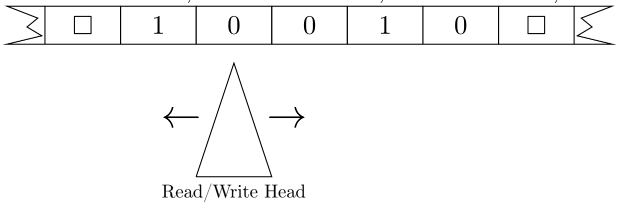
Then it reads 0, thus it writes 0, remains in S1, and moves left



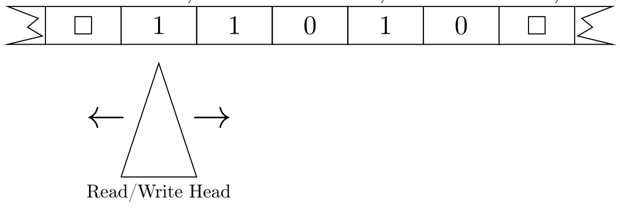
Then it reads 1, thus it writes 1, Moves to the S2 state, and moves left



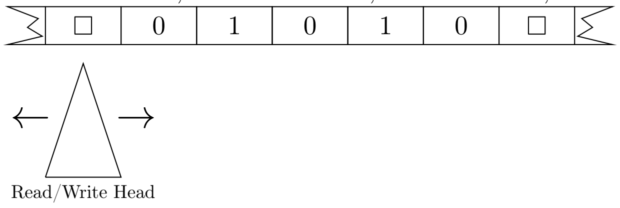
Then it reads 1, thus it writes 0, remains in S2, and moves left



Then it reads 0, thus it writes 1, remains in S2, and moves left



Then it reads 1, thus it writes 0, remains in S2, and moves left



Then it reads □, thus it writes □, Moves to the S3 state, and moves left. S3 is a halting state, therefore at this point the program stops, and the tape now reads 01010. The point of this Turing machine is to find the two's complement of the inputted string.

A universal Turing machine is a machine that can simulate a Turing machine by reading a description of the machine along with the inputs of its tape. To put it another way, what it does is it takes in the description of another machine encoded into some sort of string, and from this is able to emulate the working of this machine onto an input, which would also be provided. Both the program and the data are stored on the tape, which is similar to the stored program concept, where the program and the data the program operates on are stored in the same location in memory.

Turing machines and Universal Turing machines are useful as they provide a model of computation which allows us to clearly define if a problem is computable, that is to say that if it can be solved using a universal Turing machine, then it is computable. Universal Turing machines are useful as you don't need to make a new machine for each problem, you just need the description of the machine, and the inputs you want it to work on.