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:

```
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 `breaks`.

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`

  `17 % 3 = 0`
- Exponentiation `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:

```
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
4.1.1.10 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:

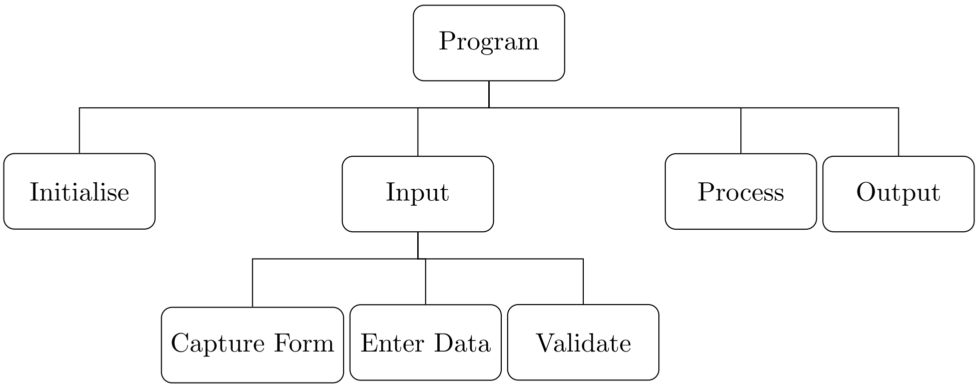
- Can't inadvertently change the value being stored elsewhere in the program.
- Use the same identifier in several places and have them be consider different variables.
- Free up memory as each time a local variable is finished with it is removed from memory.

4.1.1.14 The difference between a local and global variable is that a local variable has a limited existance 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 Role of stack frames in subroutine calls
4.1.1.16 Recursive Techniques

1.2 Programming Paradigms

4.1.2.1 A programming paradigm is...
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

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
4.2.1.2 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 dat of different types)) but can be used in the same way as arrays. Some uses of lists are as follows:

```
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 seperate variable (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:

```
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 arrange of abstract data types, 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, we would add to the end 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, 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 array. 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

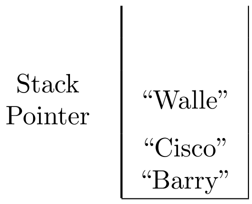
2. Pop

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

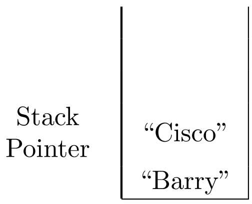
3. Peek/ Top

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

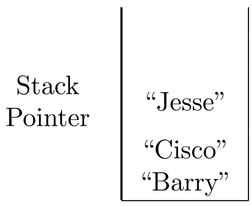
Let's show an example of how these methods would look like on an example stack:



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



And the value "Walle" 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.

Table 1: Undirected Graph

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

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

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

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

2.6 Hash Tables

4.2.6.1 Hash Tables

2.7 Dictionaries

4.2.7.1 Dictionaries

2.8 Vectors

4.2.8.1 Vectors

3 Fundamentals of Algorithms

3.1 Graph-traversal

4.3.1.1 Simple Graph-Traversal Algorithms

3.2 Tree-traversal

4.3.2.1 Simple Tree-Traversal Algorithms

3.3 Reverse Polish

4.3.3.1 Reverse Polish - Infix Transformations

Table 4: Directed Graph

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

3.4 Searching Algorithms

4.3.4.1 Linear Search
4.3.4.2 Binary Search
4.3.4.3 Binary Tree Search

3.5 Sorting Algorithms

4.3.5.1 Bubble Sort
4.3.5.2 Merge Sort

3.6 Optimisation Algorithms

4.3.6.1 Dijkstra's shortest path algorithm

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
```

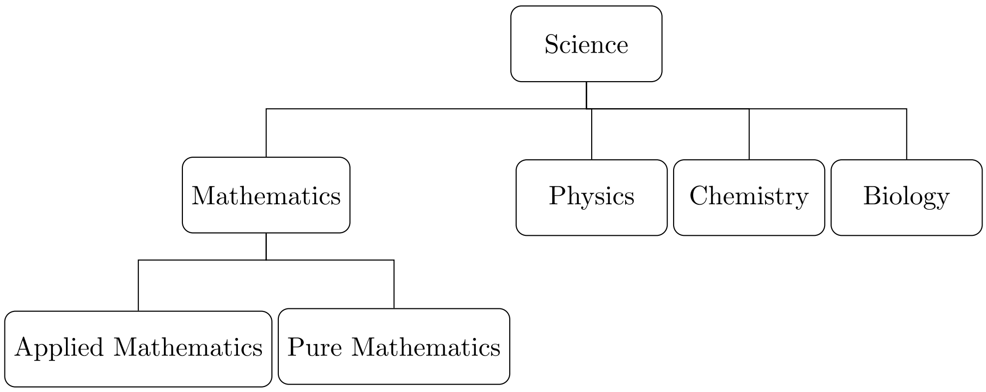
We can use a technique called hand-tracing/ dry running to work through the code and see if it works as intended. An example of a dry run is as follows:

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

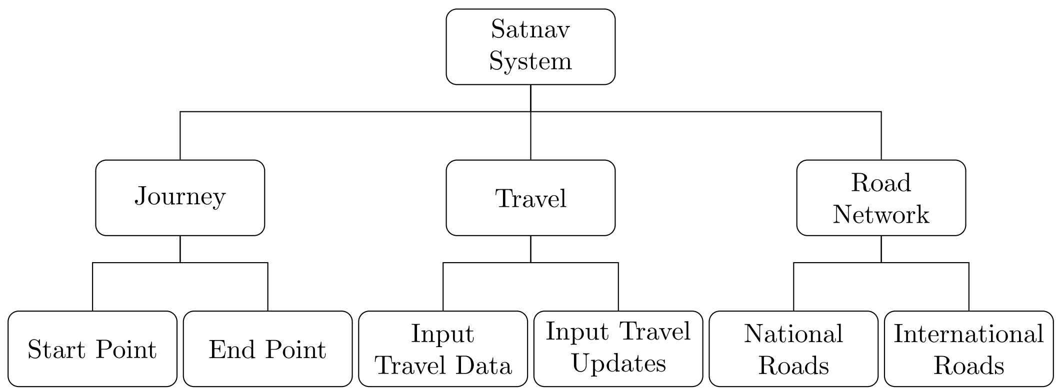
This is the concept of hiding how a data type is represented, making it easier to construct new data objects (called compound data objects). It also involves separating implementations of data objects and the user interface.

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.

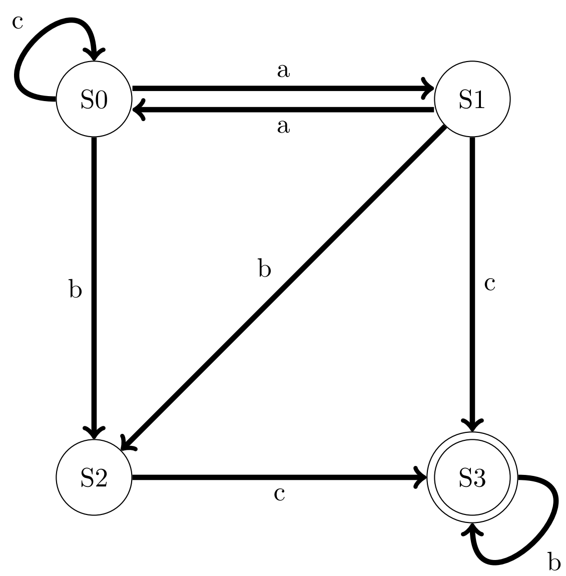


3.4.1.11 Automation is the process of creating computer models (abstraction of real world objects/ phenomena) of real-life situations and putting them into action to solve problems. This is done by:

- 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

4.4.2.2 Maths for Regular Expressions
4.4.2.3 Regular Expressions
4.4.2.4 Regular Languages

4.3 Context-free Languages

4.4.3.1 Backus-Naur Form(BNF)/Syntax Diagrams

4.4 Classification of Algorithms

4.4.4.1 Comparing algorithms
4.4.4.2 Maths For Understanding Big-O Notation
4.4.4.3 Order of Complexity
4.4.4.4 Limits of Computation
4.4.4.5 Classification of Algorithmic Problems
4.4.4.6 Computable and non-computable problems
4.4.4.7 Halting Problem

4.5 A model of computation

4.4.5.1 Turing Machine