**1 (a) (i)** • lines 10 or 11. [1]

• in those two lines, the function `Fn` calls itself. [1]

**(ii)** line 04 is the stopping condition for the recursion, i.e the base case of the recursion, i.e . [1]

**(b)** • When function `Fn` is called, memory is set aside to store local variables as well as information about the procedure such as its address in memory which will allow the program to return to the proper place after a function call [1]

• Every time a recursive call happens, a new frame for that call will be created and pushed to a stack called the *call stack*. [1]

• As a stack is a LIFO (last in first out) data structure, the last function to be called (the most recent one) will be pushed to the top. When it finishes, its frame is destroyed and removed from the stack, returning control to the frame just below it on the stack (the new top frame). [1]

So a stack is used to keep track of the order of recursive calls and its correct return address.

**(c)** • Case when `A[0]` is not an array, i.e. see `[1] + Fn([[0,1],[2]])` node [1]

• Case when `A[0]` is an array, i.e. see `Fn([0,1])+Fn([[2]])` node [1]

• at least one leaf node at [] [1]

• Correct nodes and all return valuesfile:///C:/Users/Admin/Desktop/prelims/2CZ2_PRELIM_PAPER1.pdf

**(d)** `Fn` 'flatten's a nested array, i.e it takes in an array which may contained subarrays and returns an array which contain all the elements in the subarrays without any of the subarrays. [1]

**(e)**
```
def fn_iter(x):
    if isinstance(x,int):
        return x
    containArray = True
    while containArray:
        almost_flat = []
        inner_array = False
        for i in x:
            if isinstance(i,list):
                for j in i:
                    almost_flat.append(j)
                inner_array = True
            else:
                almost_flat.append(i)
        x = almost_flat
        if not inner_array:
            containArray = False
    return x
```

- handling of integer x input [1]
- create an `almost_flat` list to store the first level of flattening [1]
- check if an element in the array is a subarray, if it is, add the elements of subarray to `almost_flat` [1]
- track if `almost_flat` contains array [1]
- loop until there's no subarrays left [1]

**(f)**
- benefit : Intuitive to use as the flattening is recursive by nature which results is simple, easy to read code. [1]
- drawback : Uses more memory because of the need to store multiple stack frames. Furthermore, if the nesting is very deep, we might hit stack overflow. [1]

**2 (a)**
- `INSTRUMENT` class [1]
- `WOODWIND` class [1]
- `BRASS` class [1]

- Correct inheritance arrows [1]
- 3 properties in INSTRUMENT... [1]
  - ... Another mark for an additional 2 [1]
- 3 methods in INSTRUMENT... [1]
  - ... Another mark for an additional 2 [1]
- property in `WOODWIND...` [1]
  - ...property in `BRASS` [1]
- method in `WOODWIND...` [1]
  - ...method in `BRASS` [1]

**(b)**
- The purpose of a superclass is to allow reusable code by allowing subclasses to inherit its attributes and methods and to extend it further. [1]
- The superclass INSTRUMENT contains attributes that subclasses WOODWINDS and BRASS inherits with each having its own additional attributes such as Type for WOODWINDS and valve_oil for BRASS. [1]

**(c)** Two from the following

- To combine data (attributes) and functions (methods) together in a single class. [1]
- This allows information hiding to be implemented where programmers can only use the given public interface (methods) to access the private attributes. This reduces accidental errors as programmers cannot change the attribute directly which may lead to an inconsistent state. Instead, implementation details are hidden from the programmer using the class. This also promotes usability as encapsulation allows access to a level without revealing the complex details below that level. [1]
- Encapsulation also allows for easier maintenance of code as code changes can be made independently. [1]

**(d)**
- Introduce new class attributes in the `INSTRUMENT` class:
  - `member_loan` which takes a Boolean value with its getter and setter methods to indicate if the loan is made by a member of the CCA. `True` if loan is made by member. `False` otherwise.
  - `daily_loan_fee` with its getter method to indicate the daily fee of the loan, [1]
- a method `get_loan_cost()` should also be implemented in the `INSTRUMENT` class to calculate the loan cost based the available attributes of `INSTRUMENT` objects. In code,

```
def get_loan_cost(self):
    if member_loan:
        return 0.20 //$0.20 is the base fee
    else:
        loan_days = self.get_return_date()-self.get_loan_date()
        return self.get_daily_loan_fee()*loan_days + 0.2
```

**(e)** To allow methods in the subclass to have the same name but possibly different behaviour. , e.g. `get_type()` returns different things for Woodwinds and Brass objects [1]

**3** **(a)** MAC address is used as a unique identifier of a device in a network in the link layer of TCP/IP protocal stack. [1]

**(b)**
```
def hex2dec(s):
    digit_list = '0123456789ABCDEF'
    total = 0
    exponent = 0
    while s:
        total = total + digit_list.index(s[-1])*(16**exponent)
        s = s[:-1]
        exponent = exponent + 1
    return str(total)
```

- getting corresponding decimal value of a hexadecimal digit [1]
- loop to add the values [1]
- correct multiplication of place digits and the corresponding exponents [1]
- correct working algorithm and return string representation [1]

**(c)** 209 : 50 : 19 : 38 : 202 : 75 [1]

**(d)** It uses fewer digits to represent the same value. [1]

**(e)** 8 comparisons [1]

**(f)** Binary search alogirthm requires the array to be sorted first. [1]

**(g)** Any working sorting algorithm [4]

**(h)** Working Binary search algorithm [4]

**(i)** A network switch sends message from a device in the network only to its intended recipient (contrast with hub) based on their MAC addresses [1]

**(j)** The switch builds up a table with the MAC address of every device that is connected to each of its ports. When a signal is received, the data is analysed to determine the destination MAC address. The data is then sent to the port connected to the device with the MAC address. [1]

**(k)**
- format check : using ':' vs using '-' [1]
- Range check : the 2 digit blocks are hexadecimal digits. [1]
- Description [1]
- Examples [1]

**4 (a)** The client–server model is a paradigm where providers of resources are designated as servers, and resource requesters are designated as clients. In this context, a resource is something that is accessed using the internet, such as a web page, an email, etc. [1]

**(b)** Peer-to-peer network is an alternative way to setup the network. [1]

**(c)**
- Benefit: In a client–server model, network security and maintenance, such as data backups, can be managed centrally. [1]
- Drawback: Network performance can be negatively impacted by multiple client requests occurring upon the server at the same time, e.g., in DDoS attacks [1]
- Alternative drawback: If central server malfunctions in client-server model, the requests of clients in the network will not be able to be fulfilled. This is unlike in peer-to-peer model where another peer will be able to serve the request.

**(d)** When services are requested by clients, the requests are put in a queue based on their time of request. Queue works on first-in-first-out basis and the client that has been served will be dequeued from the queue. [1]

**(e)** The integer variables are `head` and `tail`, which is to track the position of the first and last element of the queue respectively. [1]

**(f)** The array serves as a container where each element is an object that stores information about the clients' requests, like time of request and data that is needed to process the request. [1]

**(g)**
```
FUNCTION dequeue(arr, head, tail)
    IF tail = -1 THEN
        PRINT("Queue is empty")
        dequeued_item <- Null
    ELSE
        dequeued_item <- queue[head]
        IF head == tail THEN // Nothing left in queue
            head = -1
            tail = -1
        ELSE
            head = (head + 1) MOD Size(arr) // arr is where we hold the queue objects
        ENDIF
    ENDIF
    RETURN dequeued_item
ENDFUNCTION
```

- empty queue case specifically handled [1]

- moving the head pointer correctly [1]

```
class Client:
    def __init__(self,data,paying=False):
        self.data = data
        self.paying = paying
        self.next = None
    def set_next(self,client):
        self.next = client
        def get_next(self):
        return self.next
    def is_paying(self):
        return self.paying
    def display(self):
        return f'Client({self.data},{self.is_paying()})'
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
    def enqueue(self,client_data,paying):
        client_node = Client(client_data,paying)
        if self.head == None:
            self.head = client_node
            self.tail = self.head
        else:
            if not client_node.is_paying():
                self.tail.set_next(client_node)
                self.tail = self.tail.get_next()
            else:
                current_node = self.head
                if not self.head.is_paying():
                    client_node.set_next(self.head)
                    self.head = client_node
                while current_node:
                    prev_node = current_node
                    current_node = current_node.get_next()
                if prev_node.is_paying():
                    if current_node == None:
                        prev_node.set_next(client_node)
```

**(h)**
- case when queue is empty is handled [1]
- check for paying condition of the client before insertion [1]
- correct insertion for non paying customer [1]
- correct insertion for paying customer, both cases where there is and isn't paying customer in the queue [1]
- implementable pseducode with correct result [1]

**5 (a)**
- Non atomic. [1]
- Repeating groups of attributes (`Student ID`, `First Name`, `Last Name`) for each student [1]

**(b) (i)** `Student ID` [1]

**(ii)** `Subject ID` and `Teacher ID` (Composite Key) [1]

**(iii)** `Student ID` and `Subject ID` (Composite Key) [1]

**(c)** {}

**(i)** Counter example: SELECT Subject.SubjectID FROM Subject INNER JOIN Score ON Subject.SubjectID = Score.SubjectID WHERE StudentID = 9107

**(ii)** SELECT Subject.SubjectID FROM Subject INNER JOIN Score ON Subject.SubjectID = Score.SubjectID WHERE StudentID = 9107

**(d)**
- Not 3NF [1]
- There is a non-key dependency [1]
- The grade depends on the score as given by the question [1]

**(e)**
- `Student(Student ID, Student Name, Class Name)` [1]
- `StudentTeacher(Student ID*, Subject ID*,Teacher ID*)` [1]
- `Teacher(Teacher ID, Teacher Name)` [1]
- `Subject(Subject ID, Subject)` [1]
- `Score(Student ID*, Subject ID*, Score*)` [1]
- `Grade(Score,Grade)` [1]
- All primary keys and foreign keys indicated correctly. [1]

**(f)** Data redundancy is when there is unnecessary data repetition in a database. [1]

In the context given, `GP1` SubjectID together with its corresponding `H1 General Paper Subject` is repeated. [1]

- To free the collection of relations from undesirable insertion, update and deletion dependencies. An example update anomaly in this context, interim tables could have been created where `Subject` was recorded down as `SubjectID` instead before the merge into the `Score` table. [1]
- To reduce the need for restructuring the collection of relations, as new types of data are introduced, and thus increase the lifespan of application programs; In this context, an example would be a situation where more subjects, including their respective teachers, are being offered to the students. [1]
- Make better use of the storage space.

**(g)**

```
• SELECT Subject.Subject, Teacher.TeacherName, Score.Score, Grade.Grade
• FROM Score                                                                    [1]
• INNER JOIN StudentTeacher ON Score.StudentID = StudentTeacher.StudentID and Score.SubjectID
  = StudentTeacher.SubjectID                                                    [1]
• INNER JOIN Teacher ON StudentTeacher.TeacherId = Teacher.TeacherID
• INNER JOIN Grade ON Score.Score = Grade.Score
• INNER JOIN Subject ON Score.SubjectID = Subject.SubjectID                     [1]
• WHERE Score.StudentID = 9107                                                  [1]
• ORDER BY Grade DESC                                                           [1]
```